Constructing a Hybrid Taint Analysis Framework for Diagnosing Attacks on Binary Programs

Erzhou Zhu, Xuejun Li, Feng Liu, Xuejian Li
School of Computer Science and Technology, Anhui University, Hefei, China
Email: {ezzhu, xjli, fengliu, xjl}@ahu.edu.cn

Zhujuan Ma*
School of Economic and Technical, Anhui Agricultural University, Hefei, China
Email: ezzhusjtu@gmail.com

Abstract—**For the purpose of discovering security flaws in software, many dynamic and static taint analyzing techniques have been proposed. By analyzing information flow at runtime, dynamic taint analysis can precisely find security flaws of software. However, on one hand, it suffers from substantial runtime overhead and is incapable of discovering the potential threats. On the other hand, static taint analysis analyzes program's code without actually executing it which incurs no runtime overhead, and can cover all the code, but it is often not accurate enough. In addition, since the source code of most software is hard to acquire and intruders simply do not attach target program's source code in practice, software flaw tracking becomes rather complicated. In order to cope with these issues, this paper proposes HYBit, a novel hybrid framework which integrates dynamic and static taint analysis to diagnose the flaws or vulnerabilities for binary programs. In the framework, the source binary is first analyzed by the dynamic taint analyzer. Then, with the runtime information provided by its dynamic counterpart, the static taint analyzer can process the unexecuted part of the target program easily. Furthermore, a taint behavior filtration mechanism is proposed to optimize the performance of the framework. We evaluate our framework from three perspectives: efficiency, coverage, and effectiveness. The results are encouraging.**

Index Terms—**Binary Taint Analysis, Dynamic Analysis, Static Analysis, Software Vulnerability, Security**

I. INTRODUCTION

Malware is a collective term for **mal**icious soft**ware** which enters a system without authorization of the user. With increasing popularity of the Internet, increasing amount of vulnerable software, and rising sophistication of malicious code itself, malware is a big threat to today's computing world. Malicious attackers are able to gain access to confidential information inside the target platform, even to take control of it by taking advantage of design flaws.

In recent years, many techniques have been developed for detecting malicious software. Taint analysis is a form of information-flow analysis which establishes whether values from unauthenticated methods and parameters may flow into security-sensitive operations [1]. As taint analysis can detect many common vulnerabilities in applications, it has attracted much attention from both research and industry communities. Based on the concept that some data (such as input from a user) is not trustworthy, taint analysis tracks where the data may be used to harm the software, and monitors suspicious actions.

Generally speaking, there are two taint analyzing techniques: dynamic analysis and static analysis [2]. Static analysis is a process of analyzing program's code without actually executing it. It relies only on the information available at compile time. In this process (taking binary executable as an example), the binary code is usually disassembled into a form of assembly instructions first, then both control flow and data flow analyzing techniques can be employed to draw conclusions about the functionalities of the program. It is useful in providing a view of the overall behavior of a program without focusing on any particular execution. The technique has low overhead with respect to the utilization of system resources. However, it has the limitation of imprecision when it handles the dynamic structures (pointers, aliases and conditional statements) of the target program. Meanwhile, many interesting questions that can be asked about a program are undecidable in general cases [3].

Dynamic analysis analyzes the program at runtime. It is more precise than its static counterpart since it takes the runtime information into consideration. Since the dynamic technique only focuses on a particular execution of the target program, the amount of analysis is sharply decreased. However, it suffers from large runtime overhead, and can only detect software vulnerabilities when the attacks have been launched. So, it is impossible to locate the latent weak spots, which is very desirable in many cases. In the field of dynamic taint tracking, many testing-based techniques, attempting to detect the potential security threats by improving the code coverage [4], have been developed. However, high code coverage

* Corresponding Author

© 2014 ACADEMY PUBLISHER
doi:10.4304/jcp.9.3.566-575

is difficult to achieve, and the testing incurs too much runtime overhead.

In order to take advantage of the merits of both dynamic and static analysis and avoid the defects of both, it is necessary to combine the two approaches. In this paper, we propose HYBit, a hybrid framework which integrates dynamic and static taint analysis to discover software flaws or vulnerabilities. Since the source code of most software is hard to acquire and intruders simply would not attach target program's source code with their attacks [5], our framework is designed to handle the binary code. In order to achieve this goal, we employ CrossBit [6, 7]**,** a dynamic binary translator, as the basic tool. In the framework, the target program is first analyzed by the dynamic analyzer. Meanwhile, runtime information is collected for further analysis. Then, with the help of the runtime information, the static analyzer can work on the unexecuted part of the program. Finally, the information about both executed and unexecuted parts are reported to the end users. In order to address the low efficiency problem, we also propose a taint behavior filtration mechanism to optimize our framework. In summary, our research has the following contributions: we 1) propose HYBit, a novel hybrid framework which integrates the dynamic and static taint analysis to track the flaws or vulnerabilities for binary programs; 2) design a dynamic technique to track the tainted data and their propagation behaviors; 3) present a static analysis technique to complete the target program by adding the unexecuted part of the target program; and 4) design a novel taint behavior filtration mechanism to further optimize our framework.

This paper is a continuation of the work in [8]. On the basis of the paper, we have made tremendous improvements. The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces CrossBit, a dynamic binary translator that our framework (HYBit) is based on. In Section 4, we present an overview of our framework. Then, in Section 5, we discuss the implementation. Section 6 provides the experimental evaluation. Finally, Section 7 briefly concludes the paper and outlines our future work.

II. Related Work

Software security has drawn much attention for many years. As an effective way to detect software vulnerabilities and improve software security, taint analysis has been an important approach. Since the source code (written in a high-level language) is often hard to acquire in practice, binary taint analysis is widely used in software vulnerability detection. In this field, many static and dynamic binary analyzing techniques have been proposed.

A. Static Binary Analysis

Static binary analysis is the process of analyzing a binary executable without actually executing it. IntScope [4], with the goal of statically eliminating vulnerabilities in x86 binaries, first translates the source binary code into its own designed intermediate representation (IR), and then performs a path-sensitive data flow analysis on the IR by leveraging symbolic execution and taint analysis. UQBTng [9] is a tool capable of automatically finding integer overflows in win32 binaries. As a project for automatically analyzing vulnerabilities on SPARC binary, Chevarista [10] employs an interval analysis technique to detect buffer overflows and integer overflows. Its target is achieved during the process of statically translating the binary code into the SSA (Static Single Assignment) form. When the symbol table and debugging information are either entirely absent, or cannot be relied upon, value-set analysis, a static analysis algorithm proposed in [11], first recovers the contents of the memory locations and how they are manipulated from the x86 executables. Then, it translates the x86 binary codes onto an IR which can facilitate the work of vulnerability detection and prevention. M. Christodorescu [12] presents a static analysis mechanism to detect malicious patterns from binary executables. It presents a unique viewpoint on malware detection, i.e. malicious code detection as an obfuscation-deobfuscation game between malicious code writers and researchers working on malicious code detection.

Static analysis has the advantage that it can cover entire program code and it is usually faster than its dynamic counterpart. However, it has the limitation of imprecision when it handles the dynamic structures (pointers, aliases and conditional statements) of the target program. Meanwhile, many interesting questions that can be asked about a program are indeterminate in the general case [3].

B. Dynamic Binary Analysis

Much attention has been drawn to suspicious data tracking with dynamic taint analysis. As an intuitive way, dynamic binary instrumentation has been employed to facilitate taint tracking. Valgrind [13] is a dynamic binary instrumentation framework. It uses a dynamic binary re-compilation technique to build heavyweight dynamic binary analysis tools, such as Cachegrind, Callgrind, etc. Tools of Valgrind are created as a plug-ins, written in C, added to the Valgrind core. The basic view is "Valgrind core + tool plug-in = Valgrind tool". When the named tool starts up, the core disassembles the binary code into an intermediate representation which is instrumented with analysis code by the tool plug-in, and then converted back into machine code. Valgrind's core spends most of its time making, finding, and running translations. None of the client's original code is run. Pin [14] uses dynamic compilation to instrument executables while they are running. It is designed to provide an easy-to-use, portable, transparent, and efficient instrumentation platform for building Pintools. Pintools are written in C/C++ using Pin's rich API; and the API is designed to be architecture independent whenever possible, making Pintools source compatible across different architectures. DynamoRIO [15] is a runtime code manipulation system that supports code transformations on any part of a program. DynamoRIO provides unmodified manipulation mechanisms for applications running on stock operating systems (Windows or Linux) and commodity IA-32 and

© 2014 ACADEMY PUBLISHER

AMD64 hardware. Like Pin, DynamoRIO also provides powerful APIs. These APIs abstract away the detail of the underlying infrastructure and allow the tool builder to concentrate on analyzing or modifying application's runtime code stream. DIOTA [16] is a method for instrumenting binaries on the fly; it uses a number of backends to check programs for faulty memory accesses, data races, deadlocks, etc. DIOTA performs basic tracing operations to deploy coverage analysis, such as tracing all memory accesses or all executed codes.

Besides dynamic instrumentation, another approach to protect software is to monitor the input from the user as tainted data [2, 18]. Taintcheck [19] describes a dynamic-taint based approach to prevent overwrite attacks. Their approach taints any data read from a network socket which receives data from users. During execution, the approach monitors the binary program and guarantees that tainted data is not used as the destination of a control transfer instruction (such as jmp), a format string, or a system-call argument. Dytan [17] is a generic dynamic taint analysis framework which can handle the data flow and control flow in its taint analysis. Dytan is also flexible and does not require any specific support from the runtime system. LIFT [20], with a similar goal, provides a way to facilitate monitoring the tainted data during software execution. DYBS [21] is a Lightweight Dynamic Slicing Framework for Diagnosing Attacks on x86 Binary Programs.

Ding [22] proposes a behavior-based dynamic heuristic analysis approach for proactive detection of unknown malicious code. The behavior of malicious code is identified by the system calling through virtual emulation and the changes in the system. In their research, a statistical detection model and a mixture of export (MoE) model are designed to analyze the behavior of malicious code.

In contrast to the static binary analysis technique, dynamic technique analyzes the source executable at runtime. However, vulnerabilities cannot be detected by the technique until the target program is under attack.

C. Dynamic-Static Combined Analysis

Dynamic-static combined analysis is a methodology which integrates the two approaches in a complementary manner. It adopts the strengths of the two and eliminates their weaknesses. Rawat [23] presents a hybrid approach for buffer over-flow detection in C code. The approach makes use of static and dynamic analysis of the application under investigation. The static part consists of calculating taint dependency sequences (TDS) between user controlled inputs and vulnerable statements. The dynamic part consists of executing the program along TDSs to trigger the vulnerability by generating suitable inputs. Halfond [24] proposes a new approach of penetration testing. The approach incorporates two recently developed analysis techniques (static and dynamic analysis) to improve input vector identification and detect when attacks have been successful against a Web application. To apply their analysis work in industry security applications, Wu [25] investigates semantic metadata and structural syntax analysis. The paper explains how their approaches achieve the goal in terms of static and dynamic analysis by using industry scenarios.

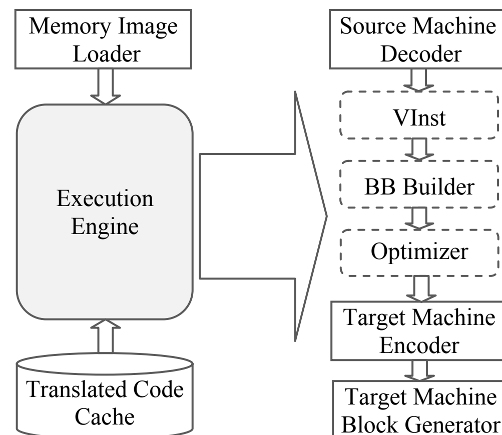


Figure 1. High-level Overview of CrossBit.

However, these methods are designed to analyze the programs written in high-level languages.

Zhang [5] describes a novel approach to overcome the limitation of traditional dynamic taint analysis by integrating static analysis into the system and presents framework SDCF to detect software vulnerabilities with high code coverage. SDCF works on DynamoRIO, a runtime code manipulation system that supports code transformations on any part of a target program. However, DynamoRIO only supports the x86 binary programs. This feature limits the usage of SDCF.

III. Introduction to CrossBit

CrossBit is designed and implemented as a dynamic binary translator, which aims at quickly migrating existing executable code from one platform to another at low cost. It supports multiple source architectures and multiple target architectures. It fully or partially supports source platforms including SimpleScalar, IA32, MIPS, SPARC and target platforms such as IA-32, PowerPC and SPARC. The operating system that CrossBit supports is Linux. In order to support code translation among multiple sources and targets, a new intermediate instruction set—VInst [6], which is independent of any specific machine instructions, has been introduced. CrossBit first converts source binary code to VInst instructions and then transforms them into target platform code, using a granularity of basic block (BB) as the basic unit of translation.

Fig.1 shows a high-level overview of CrossBit. In the figure, the rectangular boxes are related to front_end and back_end, the dotted boxes belong to the intermediate _layer. Generally speaking, the framework can be divided into three parts: the Front_end, the Intermediate_layer, and the Back_end. The Front_end is responsible for loading the binary executable code into memory and transforming the source binary code into VInst instructions. The Intermediate_layer is responsible for forming the VInst instructions into basic blocks and performing optimizations. The back_end transforms

© 2014 ACADEMY PUBLISHER

intermediate instructions (in intermediate blocks) to target instructions, and executes them immediately.

The workflow of the entire framework can be described as follows: 1) the Binary Code Execution Monitor first inserts the user-defined analysis code into the target program with the help of CrossBit. Then it switches back to continue the execution of the instrumented target program and meanwhile gets the profile information of this execution. 2) With the taint sources provided by the Taint Source Recognizer, the Dynamic Taint Analyzer tracks the taint propagation in the target program and gets the runtime analysis result. 3) According to the definition of the basic block, the CFG (control flow graph) Builder merges the current instruction into a single basic block, and organizes basic blocks that belong to the same function as a CFG. With the function call hierarchy information, the CG (call graph) Builder generates the CG of the target program. 4) With the result provided by its dynamic counterpart, the Static Taint Analyzer performs the static analysis on the unexecuted part of the target program. Besides, it is also responsible for synthesizing the result provided by its dynamic counterpart and providing a complete result to the end users.

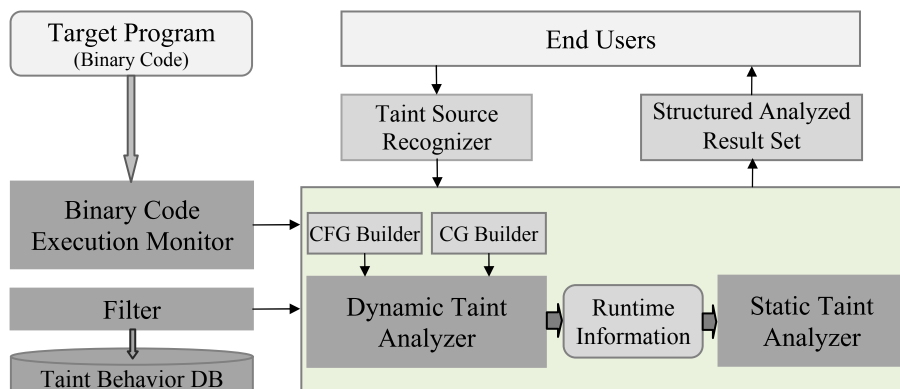


Figure 2. Overall Architecture of HYBit.

IV. Framework Overview

As Fig.2 shows, the hybrid taint analyzing framework, HYBit that we implemented contains the following main components: Binary Code Execution Monitor, Taint Source Recognizer, Dynamic Taint Analyzer, Static Taint Analyzer, and Filter.

The Binary Code Execution Monitor (based on the dynamic binary translator---CrossBit) is built to monitor the target program during its execution. As mentioned previously, CrossBit manipulates the target program at the intermediate instruction level. By inserting user-defined analyzing code to any part of the target program, we can observe and potentially manipulate the target program prior to its execution. The Taint Source Recognizer is responsible for defining suspicious taint sources. In the framework, we allow two means to define taint sources: user-defined or system default. By default, all data which is introduced to the target program from the outside is marked as untrusted. Actually, all clients that built on the framework can also define the points (memory space or registers) they may be interested in as taint sources, and track their propagation behaviors. The Dynamic Taint Analyzer is used to analyze the target program dynamically. It marks the input data from unsafe channels as tainted, and tracks their information flow and dynamic taint propagation during execution. In our framework, another important work for the Dynamic Taint Analyzer is to collect and deliver the required information to the Static Taint Analyzer. Given the runtime information provided by its dynamic counterpart, and with the help of the CrossBit, the Static Taint Analyzer can analyze the unexecuted part of the target program easily. Finally, the information about both executed and unexecuted parts of the target program is provided to the end users. The Filter is a component to optimize our framework. Through the Filter, many “safety instructions” and APIs in the system library can be ignored. This will substantially reduce the overhead of the framework. Meanwhile, the taint features of filtered API functions and “safety instructions” will be stored in a Taint Behavior Database for further use during the process of taint analysis.

V. Implementation

This section presents the key techniques for implementing our framework, including dynamic taint analysis, static taint analysis, and optimization mechanisms of the framework.

A. Dynamic Taint Analysis

Dynamic taint analysis is an effective method for detecting software vulnerabilities. This approach is based on the concept that any variable that can be influenced by the outside is a potential threat to the target program and should be monitored. Like the workflow of traditional mechanisms, dynamic taint analysis is completed by tracking the propagation behaviors of tainted data and then mining the vulnerabilities of the target program. By monitoring some system APIs (such as NtCreateFile()), we can mark the data from dangerous sources (such as opening files and network packages) as tainted. During the execution of the target program, the Dynamic Taint Analyzer traces the taint propagation by marking every memory byte or register which was influenced by the tainted sources. The framework will closely monitor all

© 2014 ACADEMY PUBLISHER

the marked data, record their dangerous behaviors (such as changing the normal execution path or changing the system stack content which should be retained), and finally, report them as potential vulnerabilities.

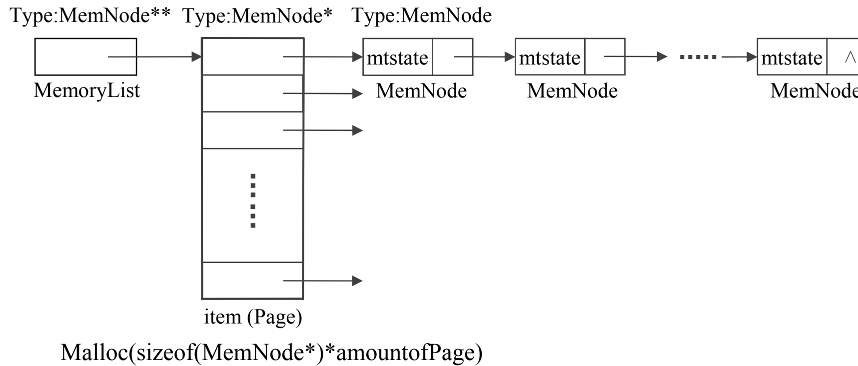


Figure 3. Data Structure of Memory Model.

A) Taint Source Locating

At the beginning of the dynamic taint analysis, it should locate the taint source in the memory space and registers, which are most likely contaminated by the input data from unsafe channels. In the framework, there are two means to identify taint sources, user-defined or system default. In the user-defined model, end users can specify some execution points (memory area or register) they may be interested in as taint sources, and then track their taint propagation behaviors. In the system default model, the framework will automatically identify the input from the outside users, and take it as suspicious. Actually, in a general Windows program, we can use three system calls (CreateFile(), CreateFileMappping(), and MapViewOfFile()) to load external files. By tracking the loading processes and the target addresses of these files, we can locate the taint sources easily.

B) Taint Propagation Tracking

After the tainted data is identified, the Dynamic Taint Analyzer monitors each binary instruction which refers to the tainted data to track the taint propagation. The Dynamic Taint Analyzer works with the underlying Binary Code Execution Monitor. Given taint sources, dynamic analyzing roles, and the instrumented target program, the Dynamic Taint Analyzer performs taint analysis during the run, and, returns the results to the following component of the framework. Since the underlying DBT system CrossBit of our framework is built as a process virtual machine, we cannot track instructions that reside in the kernel space.

Actually, this type of information flow tracking is implemented on the intermediate instruction level, and we mark any data that is derived from tainted data as tainted. Meanwhile, we just need to take care of tainted or untainted states of operands instead of their concrete values. Take the mov instruction (mov r32, r32/m32/imm32) as an example; the source operand (on the right side of the instruction) is classified as general-purpose registers, memory spaces, or immediate values. Any tainted general-purpose registers, memory locations, or immediate values in the source operand will make the destination operand (on the left side of the instruction) tainted. Otherwise, the destination operand is marked as untainted.

During the process of dynamic taint analysis, the system has to build memory and register models to record the taint states of each memory byte and register.

- *Memory Model*

Since not all the memory locations are tainted during the analysis process, we only need to record the memory addresses that are affected by the tainted sources. In the framework, a chaining hash table is used to record the tainted memory bytes. Fig.3 shows the data structure of this table.

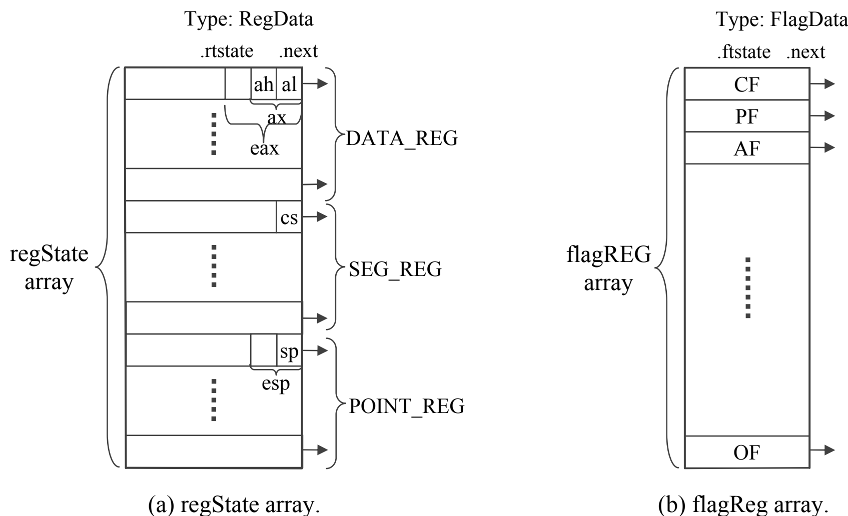
In the figure, an item corresponds to one page of the target program's virtual space, and the number of memory pages determines the number of items. Each page (the size of a page is multiple of 32 bytes) that corresponds to an item is split into several 32-byte sub-pages, and a sub-page is accommodated by a MemNode. Consequently, a page is saved by several MemNodes that belong to a single chain. There are two fields in the MemNode data structure, a pointer field (next) that identifies the next item, and a data field (unsigned int mtstate) that is used to record the taint states of 32 memory bytes. In the mtstate data field, one bit corresponds to a memory byte. The value of this bit is 1 if and only if the corresponding memory unit is tainted. Since the mtstate is a 32-bit variable, a MemNode can record the taint states of 32 memory bytes. In the model, a MemNode is added to the hash table if and only if at least one of the 32 memory bytes is tainted.

If a byte of the memory space is tainted, our tainting propagation mechanism works as follows: 1) finds which item it belongs to by its memory address; 2) creates a node in the item, and record its memory state in the node; 3) if all the taint states in a MemNode are cleaned, the corresponding nodes in the table will also be deleted.

- *Register Model*

In register taint propagation management, normally one bit is used to represent the state of a register. However, there is a special relationship among registers: some registers are part of others. That means, when the state of a register is changed, the state of other registers may be affected. For accuracy, this relationship must be taken into consideration. In the register model, as shows

© 2014 ACADEMY PUBLISHER



(a) regState array. (b) flagReg array.

Figure 4. Data Structure of Register Model.

in Fig.4, we use two arrays to record the taint states of registers: regState and flagState.

In regState, as shows in Fig.4 (a), each element (RegData) incorporates two fields: a pointer field (next) that is used to track the changes of the registers except EFLAGS, and a data field (unsigned int rtstate) that is used to record the taint states of the corresponding register. For the data registers (DATA_REG), the last three bits of rtstate are used to record their taint states. Take the eax register for example, the last and penultimate bits denote the states of *al* and *ah* respectively, the last two bits record the states of ax, and the whole three bits specify the state of eax. For the segment registers (SEG_REG), only one bit (the last bit of rtstate) is used to record their taint states. For the pointer registers, the last two bits of rtstate are used to record their states. Take the esp register for example, the last bit denotes the state of sp, and the last two bits record the state of esp.

In flagState, as shows in Fig.4 (b), we use the ftstate (Boolean variable) data field to record the taint states of the flag bits (1 denotes the register is tainted and 0 means untainted). Meanwhile, a pointer field (next) is used to track the changes of each flag.

C) Runtime Information Collection

Besides the task of taint propagation tracking, another important work of the Dynamic Taint Analyzer is to collect the runtime information of this execution. Meanwhile, it also records and structures the function calling relationships of the target program. All this work is convenient for the Static Taint Analyzer. Fig.5 shows a call graph that was generated by the framework. In the figure, every node stands for a function and the number in each node represents the entry address of the corresponding function. The arrow line is the calling relationship between each caller-callee function pair and the red ones represent the functions that are contaminated by the tainted data.

B. Static Taint Analysis

Dynamic analysis is accurate. But it can only analyze part of the target program that is actually executed. Static analysis is just the opposite. In our framework, the static analyzer is used to analyze part of target program that has not been executed or analyzed by its dynamic counterpart.

In order to analyze the unexecuted part of the target program, we have to acquire the “whole picture” of it. As mentioned in Section 4, the dynamic analyzer also saves the runtime information that is required for analysis by the static analyzer. Actually, during the dynamic analysis, the CG (Call Graph) of the target program has been built and for each function in the CG, the CFG (Control Flow Graph) of the executed code has also been constructed.

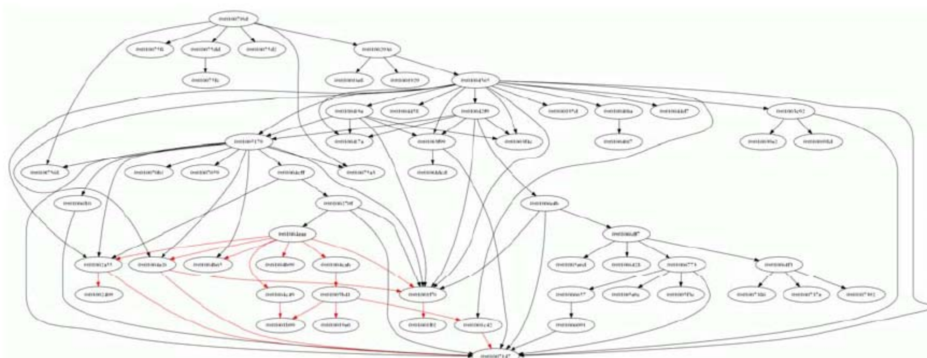


Figure 5. Call Graph of a Target Program.

© 2014 ACADEMY PUBLISHER

Meanwhile, branch points like conditional jumps will be added to the branch list of the function for use in static completion analysis. This runtime information makes the task of static taint analysis easier.

For each function call of the target program, the static analyzing module uses the following steps to traverse the branch point list (a kind of runtime information, which initially stores the branch points of the executed part of the function), and meanwhile supplement the unexecuted part of the function: 1) collects the branch points of the executed part of the function, and stores them in a branch point list P; 2) for each branch point p in P, utilizes b=GetNextBB(p) to get the next unexecuted basic blocks; 3) if the newly derived basic block b is not in B (the basic block set used to accommodate the unexecuted basic blocks), goes to step 4), else goes to step 2); 4) if b's last instruction is a conditional branch p', adds p' into P, else goes to step 5); 5) adds b into B; 6) sets a new edge e, which starts from p and ends with b, and stores it into E (the edge set used to accommodate the unexecuted edges); 7) goes to step 2).

After executing the above steps, information (branch points, edges, and basic blocks) about the structure of the unexecuted code is available, and subsequently, the visualized CFG can be generated for the user for manual analysis. By integrating the information about individual functions, we can get the whole structure of the target program. Given such information, the static analyzer is able to apply taint analysis on the unexecuted part of the target program which cannot be reached by the dynamic analyzer. The static analyzing module analyzes the whole target program by traversing each unexecuted path. As the experimental results shown in Section 5, this part of the code is usually not very large. Moreover, with the help from dynamic analysis, much of the code can be skipped because it is irrelevant to the tainted data.

It is notable that, given the runtime information, the static taint analysis in our framework is more accurate than most other static approaches. However, this type of information also limits the scope of the static analysis, because it may be dissimilar in different instances.

C. Framework Optimization

In this subsection two approaches, instruction-level filtering and function-level filtering, are employed to optimize the performance of the proposed framework. The two approaches first summarize and determine the taint propagation behaviors of certain parts of the target program. Then, given this information, they can optimize the analyzing processes by avoiding dealing with every instruction of the target program.

A) Instruction-Level Filtering

Actually, in the process of taint propagation tracking, we divided the instructions into two categories. 1) Instructions which can propagate tainted data, such as LOAD, PUSH, MOV, etc. When dealing with this type of instructions, we mark any data that is derived from tainted data as tainted. 2) Instructions which cannot propagate tainted data, such as NOP, JMP, CMP, etc. This type of instruction does not affect taint propagation, and we can skip them to reduce the analysis work load.

There are also some special instructions whose results do not depend on the input. The behaviors of these instructions should be summarized and determined before the task of code analyzing takes place. Take the XOR instruction (xor eax, eax) for example, the value of the destination operand eax (on the left side of this instruction) is always set to zero regardless the original value in the register. In this case, eax is set clean (untainted) after execution of the instruction. Another example is the binary instructions which are used to initialize the stack of function calls. The functionalities of these instructions are quite similar and their behaviors are predictable.

B) Function-Level Filtering

Since the behaviors of the most APIs in the system library can be determined and need not be analyzed in every instance, we first examine the code of these APIs and gather their taint propagation information. Then, when the APIs are called, the framework does not have to analyze every instruction of them.

According to a common observation, a large part of the binary code in software is loaded from system libraries such as Kernel32.dll, MSVCRT.dll and USER32.dll. Since the behaviors of these modules are predictable, the filtration mechanism summarizes the taint effects of these API functions and skips them. In the first place, it has to examine the source code and documents of these APIs, gather their taint propagation information and store all these kinds of information into the Taint Behavior DB.

After the taint effects of API functions, some principles are designed to provide the decision of the code filter: 1) if a function does not do anything in the taint propagation, the taint status of the program does not change; 2) if a function can propagate the tainted data, the taint status of the program will remove the source parameter from the set of tainted data; and 3) if a function can propagate the tainted data, the destination of the tainting should be marked as tainted and brought into the monitoring of the system. With these principles, irrelevant API functions can be free from instrumentation of information tracking. However, some API functions cannot be simply skipped, such as strcpy, which has a tight relationship with some software vulnerabilities. Therefore, this kind of API function must still be analyzed in the process of information flow tracking.

VI. Experimental Evaluation

Our experimental evaluation has three main goals:

1) Assessing the efficiency of HYBit by comparing the performance with the native platform;
2) Studying the coverage of HYBit by counting how much code of the target program can be analyzed by the framework;
3) Evaluating the effectiveness of HYBit by applying it to software vulnerability detection.

© 2014 ACADEMY PUBLISHER

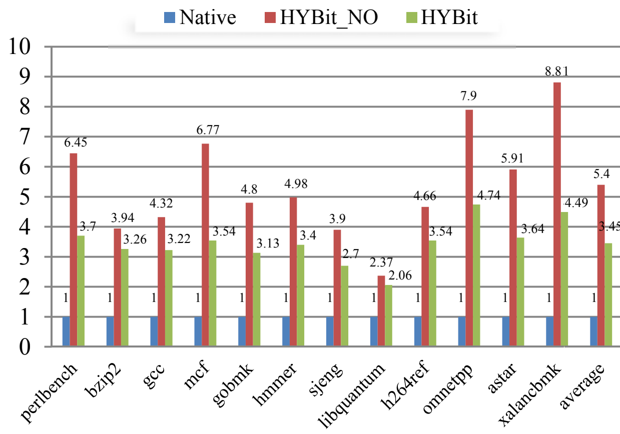


Figure 6. The Performance Evaluation of the HYBit.

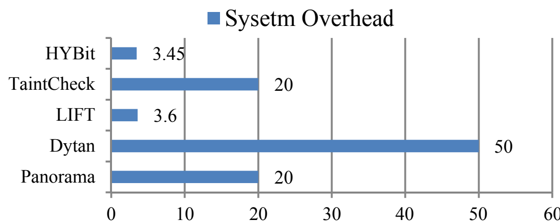


Figure 7. System Overhead Comparison Between HYBit and Other Attack Diagnosis Tools.

A. *Efficiency*

This subsection provides a performance evaluation of HYBit. The experiment is based on the SPEC CINT2006 benchmarks on Windows. Fig.6 shows normalized execution time (the ratio of our time to native execution time, as described in Equation (1) below) of HYBit. In the figure, the results of NATIVE, HYBit, and HYBit_NO refer to the execution time of the binary code running on the underlying platform CrossBit directly, running with analysis by HYBit, and with analysis but without the taint behavior filtration optimization.

$$\text{Performane} = \frac{\text{HYBit (or HYBit_No)}}{\text{Native}} \tag{1}$$

Because of facilitation of the underlying binary translation platform CrossBit as well as the static analysis of HYBit, we achieve the overhead of 5.4 times on average to the native platform without the optimization of taint behavior filtration. However, when the taint behavior filtration optimization is introduced to the system, the time cost reduces remarkably, and the average runtime overhead is 3.45 times to that of native execution.

It can be observed from Fig.6 that the performance of HYBit differs among the target programs. One reason is the structural distinction among these programs. Little time is spent on analyzing library functions that can be summarized and filtered by the optimization component of the framework so the structure of the .exe module of the target program is an important factor in the performance of our system. It is also worth noting that we

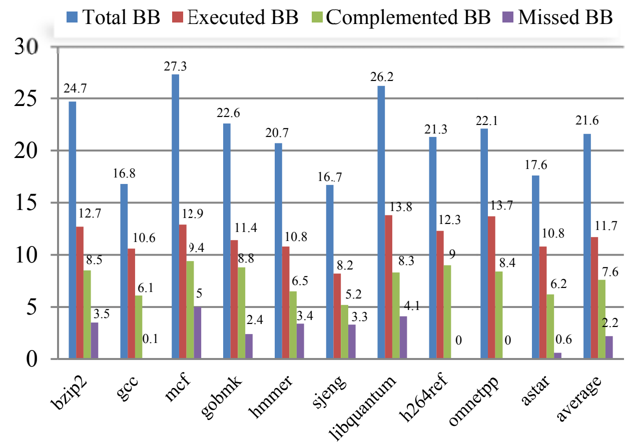


Figure 8. Coverage Evaluation of HYBit.

have not summarized the system library entirely. So, when the programs call functions which have not been included in our taint behavior database, the overhead will increase.

Fig.7 shows the average overhead comparisons between HYBit and other popularly used binary-level attack diagnosing tools: Dytan [17], Panorama [18], TaintCheck [19], and LIFT [20]. Compared to tools such as Panorama (slowed down the target programs by 20x on average), Dytan (50x), LIFT (3.6x) and TaintCheck (20x), HYBit incurs much lower runtime overhead (3.45x).

B. *Coverage*

In this subsection, we evaluate the coverage of HYBit by counting how much code of the target program can be covered by the framework. In order to achieve this goal, we also introduce the SPEC CINT2006 benchmarks. As shown in Fig.8, we are concerned with four kinds of results: Total BB (the number of total basic blocks in the target program), Executed BB (the number of basic blocks which are analyzed by the dynamic taint analyzer), Complemented BB (the number of basic blocks that are analyzed by the static taint analyzer), Missed BB (the number of basic blocks which are not recognized by our framework). The Coverage Rate (Executed BB and Complemented BB as a percentage of all basic blocks in the target program) can be calculated as Equation (2).

$$\text{Coverage Rate} = \frac{\text{Executed BB} + \text{Complemented BB}}{\text{Total BB}} \tag{2}$$

Total BB is the sum of executed BB, Complemented BB and Missed BB, as shown in Equation (3).

$$\text{Total BB} = \text{Executed BB} + \text{Complemented BB} + \text{Missed BB} \tag{3}$$

From Fig.8 and Equation (2), we can derive that the Coverage Rate of HYBit reaches 90% on average. There are two reasons why these indices are not 100%: (1) HYBit cannot handle an indirect control transfer which depends on the context; (2) the information that is used for the algorithm of static completion is acquired from the

© 2014 ACADEMY PUBLISHER

TABLE I.
EFFECTIVENESS VERIFICATION ON GENERAL SOFTWARE

Target Program	Attacks	Discovered Attacks	Potential Vulnerabilities
Kingsoft WPS	2	2	14
Justsystem Ichitaro	3	3	2
Hangul HWP	3	3	42
MS Word 2003	0	0	3
Foxit Reader	22	22	12
IrfanView	32	32	4
Total	62	62	77

dynamic analyzer, however, this kind of information only holds for a specific instance.

C. *Effectiveness*

In this part of the evaluation, we test the effectiveness of HYBit. As mentioned previously, through dynamic and static analysis of our framework, information on both executed and unexecuted parts of the target program is reported to the end users for further analysis. By making full use of this information, we can discover not only the possible weak points in the executed part of the target program, but also the latent software vulnerabilities which has not been executed.

Table 1 provides the results of running Kingsoft WPS (a word processing software for Chinese), JustSystems Ichitaro (a word processing software for Japanese), Hangul HWP (a word processing software for Korean), MS Word 2003 (a widely used word processor), Foxit Reader (another widely used document processor for Chinese), and IrfanView (a free graphic viewer for Windows) on HYBit.

In the table, the column "Attacks" contains the number of attacks incorporated in the tested target programs. The "Discovered Attacks" and "Potential Vulnerabilities" columns show the numbers of attacks and potential vulnerabilities been discovered by HYBit. From the data that shown in the table, we see that all the predefined (by CVE library) attacks are discovered by HYBit, so the recognition ratio is 100%. Furthermore, HYBit can also detect 77 potential vulnerabilities that may pose potential threat to the target programs.

VII. CONCLUSIONS AND FUTURE WORK

This paper proposes HYBit, a novel hybrid framework which integrates dynamic and static taint analysis to track the flaws or vulnerabilities for binary programs. Dynamic analysis and static analysis techniques offer two complementary approaches for checking vulnerabilities. HYBit exploits the strengths of both, and eliminates the drawbacks. Based on the dynamic binary translator CrossBit, HYBit first employs a dynamic analyzer to check the binary code of the target program; tainted data and its propagation are thereby tracked in one part (the executed part) of the target program. Then, with the runtime information provided by the dynamic part, static analysis of the unexecuted part of the target program becomes easier. The static analyzer employs a static completion algorithm to construct the "whole picture" of the target program. Then the unexecuted part of the target program is analyzed. In order to further improve the performance, a taint behavior filtration optimization mechanism is proposed. With this mechanism, many APIs functions from the library and "safety instructions" are skipped which greatly reduces the overhead of the framework. The results of the experiments on benchmarks show that the system is efficient and effective and covers most part of the target program.

In the future, more optimization methods, such as program slicing, both in the dynamic analysis and static analysis will be proposed to further improve the performance of the framework. We are grateful for Professor Yun Yang from Swinburne University of Technology in Australia for English proofreading.

ACKNOWLEDGMENT

This work was supported by the academic and technical leader recruiting Foundation of Anhui University, the National Natural Science Foundation of China (Grant No. 61300169, 61003131), the National High Technology Research and Development Program (863Program) of China (Grant No. 2012AA010905), the National Basic Research Program (973 Program) of China (Grant No. 2012CB723401).

REFERENCES

[1] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09), June 2009, pp.87-97.

[2] Christoph Csallner, Yannis Smaragdakis, Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. ACM Transactions on Software Engineering and Methodology, 17(2):1-37, 2008.

[3] Paolo Zuliani, André Platzer, Edmund M. Clarke. Bayesian statistical model checking with application to simulink/stateflow verification. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '10), April 2010, pp.243-252.

[4] Tielei Wang, Tao Wei, Zhiqiang Lin, Wei Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: Proceedings of Network and Distributed System Security Symposium (NDSS '09), February 2009.

[5] Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, Haibing Guan: Static program analysis assisted dynamic taint tracking for software vulnerability discovery. Computers & Mathematics with Applications, 63(2): 469-480, 2012.

[6] Erzhou Zhu, Haibing Guan, Guoxing Dong, Yindong Yang, Hongbo Yang. A Translation Framework for Executing the Sequential Binary Code on CPU/GPU Based Architectures. Journal of Software, 6(12):2331-2340, 2011.

[7] Haibing Guan, Erzhou Zhu, Kai Chen, Ruhui Ma, Yunchao He, Haipeng Deng, Hongbo Yang. A Dynamic-Static Combined Code Layout Reorganization Approach for Dynamic Binary Translation. Journal of Software, 6(12):2341-2349, 2011.

[8] Erzhou Zhu, Haibing Guan, Rongbin Xu, Feng Liu. HYBit: A Hybrid Taint Analyzing Framework for Binary Programs.

© 2014 ACADEMY PUBLISHER

Lecture Notes in Computer Science, 2013(7929):232-239, 2013.

[9] R.Wojtczuk. UQBTng: a tool capable of automatically finding integer overflows in win32 binaries. In 22nd Chaos Communication Congress, November 2005.

[10] Tyler Durden. Automated Vulnerability Auditing in Machine Code. http://www.phrack.com/issues.html?issue=64&id=8. Version of May 22, 2007.

[11] Gogul Balakrishnan,Thomas Reps. Analyzing Memory Accesses in x86 Executables. Lecture Notes in Computer Science, 2004(2985): 5-23, 2004.

[12] Mihai Christodorescu, Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. Technical Report # 1467 at the Computer Sciences Department of the University of Wisconsin, Madison, US.2003.

[13] Nicholas Nethercote, Julian Seward. Valgrind: A program supervision framework. Electronic Notes in Theoretical Computer Science, 89 (2):89-100, 2003.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05), June 2005, pp.190-200.

[15] Derek Bruening, Qin Zhao, Saman Amarasinghe. Transparent Dynamic Instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12), March 2012, pp.133-144.

[16] Jonas Maebe, Michiel Ronsse, Koen De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT '02), September 2002.

[17] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA '07), July 2007, pp.196-206.

[18] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07), October 2007, pp.116-127.

[19] James Newsome, Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of 2005 Network and Distributed System Security Symposium (NDSS '05), 2005.

[20] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, Youfeng Wu. Lift: A lowoverhead practical information flow tracking system for detecting security attacks. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06), December 2006, pp.135-148.

[21] Erzhou Zhu, Feng Liu, Xianyong Fang, Xuejun Li, Yindong Yang, Alei Liang. DYBS: a Lightweight Dynamic Slicing Framework for Diagnosing Attacks on x86 Binary Programs. Journal of Software, Accepted in August 2013.

[22] Jianguo Ding, Jian Jin, Pascal Bouvry, Yongtao Hu, Haibing Guan. Behavior-Based Proactive Detection of Unknown Malicious Codes. In: Proceedings of the 2009 Fourth International Conference on Internet Monitoring and Protection (ICIMP '09), May 2009, pp.72-77.

[23] Sanjay Rawat, Dumitru Ceara, Laurent Mounier, Marie-Laure Potet. Combining Static and Dynamic Analysis for Vulnerability Detection. CoRR abs/1305.3883 (2013).

[24] William G. J. Halfond, Shauvik Roy Choudhary, Alessandro Orso. Improving penetration testing through static and dynamic analysis. In Proceedings of Software Testing, Verification and Reliability, 2011, pp.195-214.

[25] Raymond Wu, Masayuki Hisada. Static and dynamic analysis for web security in industry applications. International Journal of Electronic Security and Digital Forensics, 3(2):138-150, 2010.

Erzhou Zhu is currently a lecturer with School of Computer Science and Technology, Anhui University (Hefei, China). He received his Ph.D. degree in computer science from Shanghai Jiao Tong University (Shanghai, China) in 2012. His current research interests include, but are not limited to, program analysis, computer architecture, compiling technology, virtualization and cloud computing.

Xuejun Li is an associate professor with School of Computer Science and Technology, Anhui University (Hefei, China). He received his Ph.D. degree from Anhui University in 2005. His research interests are program analysis and embedded systems.

Feng Liu is currently a professor with School of Computer Science and Technology, Anhui University (Hefei, China). He received his Ph.D. degree in computer science from University of Science and Technology of China (Hefei, China) in 2003. His current research interests include computer architecture, parallel computing, and cloud computing.

Xuejian Li received his Master degree in computer science from Anhui (Hefei, China) in 2008. His current research interests include program analysis, software security, and compiling.

Zhujuan Ma received his Master degree in computer science from Anhui Agricultural University (Hefei, China) in 2011. His current research interests include virtual machines, computer architecture, and compiling.

© 2014 ACADEMY PUBLISHER