DYBS: A Lightweight Dynamic Slicing Framework for Diagnosing Attacks on x86 Binary Programs

Erzhou Zhu, Feng Liu, Xianyong Fang, Xuejun Li*
Key Laboratory of Intelligent Computing and Signal Processing of Ministry of Education & School of Computer Science and Technology, Anhui University, Hefei, China
Email: {ezzhu, fengliu, fangxianyong, xjli}@ahu.edu.cn

Yindong Yang, Alei Liang
Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: {yasaka, aleiliang}@sjtu.edu.cn

***Abstract*—Nowadays, applications are usually large-scale, this making tasks of comprehending and debugging software rather complicated. As a dynamic reduction technique for simplifying programs, dynamic program slicing is an effective and important approach for locating and diagnosing software attacks. However, most of the existing dynamic slicing tools perform slicing at the source code level, but the source code of most software is hard to acquire in practice. In order to cope with this issue, a novel lightweight dynamic slicing framework---DYBS, is proposed for diagnosing attacks on x86 binary programs. During the execution, DYBS first gathers the runtime profile information of the target program. Once the attack is encountered and set as the slicing criterion, the normal execution terminates, and a backward program slicing is started to locate the vulnerabilities. Furthermore, a Function Call Filtration optimization mechanism is proposed to improve the performance of the framework. It is proved in the experiments that DYBS can diagnose software attacks with much lower overhead than many other similar analyzing systems.**

***Index Terms*—Dynamic Program Slicing, Dynamic Binary Analysis, Attack Diagnosis, Software Security**

I. INTRODUCTION

Nowadays, computers are affecting people more and more deeply in their work and daily life. With the increasing popularity of the Internet, the increasing number of available vulnerable software, and the elevating sophistication of the malicious code itself, malware, a collection term for malicious software which enters system without authorization of user of the system, is a big threat to today's computing world [1-3]. Malicious users are able to gain access to confidential information inside the target platform, even take control of it by taking advantage of the designing flaws [4].

Take notorious buffer overflow as an instance, as shown in Fig.1, attackers can exploit this software vulnerability by manipulating the software input, and cause overwrite in the stack to take control of the execution stream of the program [5]. For clarity, the situation is described with C code while our framework is implemented to handle binaries. The code in the example gets data from input, and loads it into the buffer. The input contains a flag to indicate its characteristics. The program handles inputs with invalid data, ANSI format and UNICODE format differently. As Fig.1 presents, invalid and ANSI format data are taken good care of, but the procedure for dealing with UNICODE contains a design flaw which may cause buffer overflow.

```
buffer[MAXLINELEN];
InputList *il;
//code to handle input.
if (il->flag == FLAG_INVALID)
   Input_invalid_procedure(il);
else if(il->flag == FLAG_UNICODE)
   Input_unicode_procedure(buffer, il);
else if(il->flag == FLAG_ANSI)
   Input_ansi_procedure(buffer, il);
else
   ...
void Input_unicode_procedure(buffer, il)
   {
      ...
      sprintf(buffer, "%s", il->usr_str); //buffer overflow
      ...
   }
```

Figure 1. A typical form of software vulnerability.

However, modern applications are usually large-scale, which makes understanding software and diagnosing attacks rather complicated. Therefore, when attacks occur in large-scale software, method of locating and diagnosing them at low cost is an interesting research direction. Program slicing, a program reduction technique that simplifies programs by removing parts labeled non-relevant with respect to a slicing criterion, is gaining more and more attention in computer societies [6][7]. It is a promising technique for providing automatic support for various important software engineering activities, including software maintenance, software measurement,

* Corresponding Author

program comprehension, and program optimization. It is also been proved to be an effective approach to enhance security and diagnose attacks in software, and there are many applications have been proposed in this field, such as preserving data confidentiality for a target program [8], locating bugs in parallel programs [9], identifying attacks and bugs dynamically [10].

Generally, there are two approaches to implement this technique, static program slicing and dynamic program slicing. As the traditional form, static program slicing [6] relies purely on the information which is available at compile time. 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. Static slicing also helps in comprehending the overall dependencies of the selected slicing criterion. However, static slicing has the limitation of imprecision when it handles the dynamic structures (pointers, aliases and conditional statements) of the target program. Meanwhile, since static slices are computed for all possible executions rather than a specific execution, the generated slices are large.

Dynamic program slicing [11] is a technique that does the program slicing in the actual process of running the target program. By using dynamic program slicing, users can extract the instructions and basic blocks (sequences of instructions ending with a single control transfer instruction) with respect to slicing criterion during the execution of the target program. The extracted instructions and basic blocks will constitute a subset of the total code. Compared with the static method, the dynamic approach has at least two advantages. Firstly, dynamic program slicing is more precise than its static counterpart since it takes the runtime information into consideration. Secondly, since dynamic slicing only focuses on a particular execution of the target program, the size of the slicing result is sharply decreased. However, the construction of a dynamic program slice is expensive since it requires tracing of the program's execution [12]. In order to cope with this issue, much research [13][14] on dynamic program slicing is concerned with improving the algorithms for slicing both in terms of reducing the size of the slices and improving the time efficiency.

Actually, most of the current dynamic slicing methods perform dynamic program slicing at the source code level. However the source code of many programs is not easy to acquire; this makes the task of dynamic slicing rather complicated or even impractical. DYBS, a new lightweight dynamic binary program slicing framework, is proposed in this paper to locate and diagnose attacks on x86 binary programs. DYBS first gathers the profile information during the execution of the target program. However, once the attack is encountered and set as the slicing criterion, the normal execution terminates and a backward program slicing is employed to locate the vulnerabilities of the target program. DYBS is built with the help of DynamoRIO [15], a code manipulating system to monitor the target binary program dynamically.

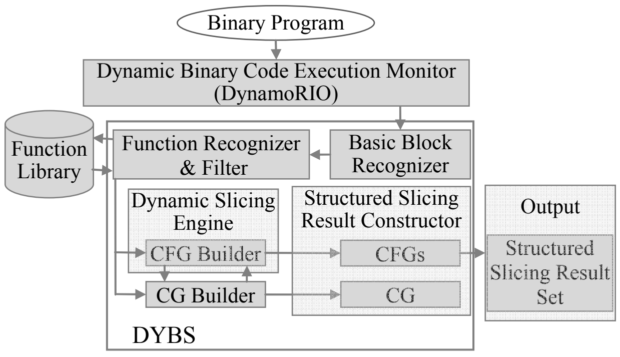
Binary Program
Dynamic Binary Code Execution Monitor (DynamoRIO)
Function Library
Function Recognizer & Filter
Basic Block Recognizer
Dynamic Slicing Engine
Structured Slicing Result Constructor
CFG Builder
CFGs
CG Builder
CG
Output
Structured Slicing Result Set
DYBS

Figure 2. The overall framework of DYBS.

Furthermore, a Function Call Filtration optimizing mechanism is proposed to improve the performance of the framework. In summary, this paper makes the following contributions: (1) proposes a novel dynamic binary program slicing framework---DYBS. It combines dynamic program slicing with binary analysis, and performs dynamic slicing analysis directly on the binary executables of target programs; this greatly expands the scope of the dynamic program slicing technology. (2) Applies this technique to diagnose vulnerabilities in and attacks on the software, and demonstrates the feasibility of this idea in experiments. (3) Organizes the slicing results by function CG (Call Graph) and CFG (Control Flow Graph) to make them more concise for the users. (4) Designs a Function Call Filtration optimization approach in the proposed framework to simplify the slicing process and further improve the performance.

The remainder of the paper is organized as follows: In Section Ⅱ, an overview of the framework is presented. In Section Ⅲ, DYBS is described and its implementation discussed. Section Ⅳ provides an experimental evaluation of the proposed framework. In Section Ⅴ, the related work is introduced. Finally, section Ⅵ briefly concludes this paper and outlines some future work.

II. FRAMEWORK OVERVIEW

As Fig.2 shows, the dynamic binary program slicing framework that we implemented, DYBS, contains the following main components: Dynamic Binary Code Execution Monitor, Basic Block Recognizer, Function Recognizer and Filter, Dynamic Slicing Engine, CFG (Control Flow Graph) Builder, CG (Call Graph) Builder, and Structural Slicing Result Constructor.

The proposed DYBS is built on the DynamoRIO platform, a runtime code manipulation system that supports code transformation and instrumentation on any part of a target program. DynamoRIO is the IA-32 version of Dynamo [31], and it is implemented for both IA-32 Windows and Linux operating systems. The goals of DynamoRIO are to run large desktop applications and to observe or potentially manipulate every single application prior to its execution. It takes basic blocks as the basic execution unit, copies these blocks into a code cache, and executes them natively. DynamoRIO capable of intercepting a variety of system calls, which greatly facilitates the tasks of attack diagnosis in DYBS.

The workflow of the entire DYBS system can be described as follows: (1) the Dynamic Binary Code Execution Monitor first inserts the user-defined analysis code into the target program with the help of DynamoRIO. Then DYBS switches back to continue the execution of the instrumented target program, and meanwhile gets the profile information of this execution. (2) According to the definition of basic blocks, the Basic Block Recognizer merges the current instructions into a single basic block, and delivers it to the following components of DYBS. (3) The Function Recognizer deals with every function call during the execution of the target program. It combines basic blocks that belong to the same function. (4) During the execution of the target program, if a fatal error is encountered, the current execution terminates and an error is reported to the system. At this time, a backward slicing algorithm is started with respect to the slicing criterion (memory crash point address). (5) The Function Filter is used to optimize the performance of DYBS. It compares the current function with the functions that are contained in the user defined Function Library. If the current function already existed in the library, this function can be directly skipped in the process of code analysis; otherwise, it should be further analyzed by DYBS. (6) The Dynamic Slicing Engine executes the dynamic program slicing algorithm on the basic block set of the current function. It slices out the ones that are related to the slicing criterion. In the Slicing Engine, the CFG Builder analyzes the basic blocks in every function, organizes the basic block sequence and constructs the CFG of this function. (7) The CG Builder imitates the stack operations like an operation system to manage the call relationships of the function and utilizes the profile information to generate the CG of the whole program. (8) Once the CG of the whole program and the CFG of each function are available, the Structural Slicing Result Constructor is capable of organize the slicing results concisely.

III. Implementation

This section presents the key techniques for the implementation of DYBS, which includes dynamic binary program slicing, inter-procedure analyzing and real-time data processing, slicing result structuring, and system optimizing.

A. *Dynamic Binary Program Slice*

Generally, program slicing can be divided into two categories: forward program slicing and backward program slicing. The dynamic binary program slicing in DYBS refers to the technique that analyzes and extracts instructions that affect or are affected by the slicing criterion. As mentioned previously, the primary goal of DYBS is to dynamically locate and diagnose software vulnerabilities when the attacks take place. This means it has to set the attack point, i.e. the memory address of the crash point, as the slicing criterion. Then it uses the backward program slicing technique to slice out basic blocks that affect this slicing criterion during the execution of the target program. Actually, the set of extracted basic blocks is a subset of the whole original program, and it is indeed executable. It is worth mentioning that DYBS is also capable of using forward program slicing. In the rest of this paper, we mainly discuss the backward program slicing mechanism of DYBS.

```
Worklist Algorithm (Backward)

Input
      W: basic block set of the target program;
       n: memory address of the crash point.
Output
       S: set of the sliced basic blocks
1.  set $V = \Phi$;  // V stores the slicing criterions in each iteration step
    set $S = \Phi$; //S stores basic blocks that selected during the slicing
                        process
2.  $i, j \in N$;  // i, j are the loop iterative variables
      $v_i$ is an arbitrary instruction of a basic block in V;
      $w_j$ is an arbitrary instruction of a basic block in W;
3.  $V \leftarrow I_n$;  //n is the slicing start point
    //$I_n$ is the instruction set corresponding to the memory address n
4.  while ($V \neq \Phi$) do
5.      for (an arbitrary intermediate instruction $v_i$ in V) do
             for ($j = i \rightarrow 0$) do  // backward slicing
                 if $v_i$'s source operand is the destination operand of $w_j$
                    then $V \leftarrow w_j$;
                         $S \leftarrow w_j$;
                 endif
             endfor
             delete $v_i$ from V;
        endfor
    endwhile
  return S;
```

Figure 3. Worklist backward program slicing algorithm.

In DYBS, the backward slicing algorithm it employs is called the Worklist Algorithm. As described in Fig.3, there are two inputs to the algorithm, the basic block set of the target program (W), and the memory address where the attack takes place (n). The output of the algorithm is a set of sliced basic blocks (S). In the algorithm, V is the intermediate instruction set to accommodate the slicing criteria in each iteration step, i and j are the loop iteration variables, v_i is an arbitrary instruction of a basic block in V, and w_j is an arbitrary instruction in W.

The core idea of the Worklist Algorithm is this: for an arbitrary instruction v_i in the slicing source set V, the algorithm judges whether each instruction w_j (in W) before v_i affects v_i (i.e. the source operand of v_i is the destination operand of w_j). If it does, w_j will be added to the sets V and S. Each time the judgment of v_i finished, v_i will be removed from the slicing source set V. The algorithm repeats the above steps and continuously updates the slicing source set V and slicing result set S, until the slicing source set V is empty.

At the end of the execution of the backward Worklist Algorithm, the set S is the result that is composed of the extracted basic blocks. By this algorithm, users just need to analyze the sliced result rather than the whole program. This greatly narrows the scope of code analysis, and can improve the efficiency of locating and diagnosing attacks.

B. *Inter-procedural Analysis*

During the process of dynamic program slicing, DYBS will automatically construct the CFGs and CG of the target program. With the two data structures, the scenarios of function calls, stack operation and basic block executing sequence can be imitated. DYBS collects the function call relationships and stores them in a tree where the callers are regarded as parent nodes and the callees as children nodes. With the call graph, users can acquire the function call hierarchy of the target program, and it is very convenient for them to locate the vulnerabilities. A CFG describes the basic block execution sequence of a single function. From the CFG, users can obtain the basic block dependencies of a function. Meanwhile, the CFG properly ensures the correctness of a slicing algorithm when it executes on a program.

In DYBS, a data structure (called a Bitmap) is employed to record the slicing states of all basic blocks. For an arbitrary bit in the Bitmap, a 1 indicates that the basic block is included in the slicing result set, and 0 that is not. This method is efficient and requires less storage space. With the help of stack operation information from CG and CFGs, DYBS is capable of breaking through the function field limitation and perform the inter-procedural analysis easily.

C. *Real-time Data Proces*

Since DYBS performs its analysis during the running of the target program, it can acquire a lot of runtime information, such as the actual values of pointer variables and the values of array indices. So, it is more precise and effective than static program slicing. Since all variables are assigned a specific value during the runtime, DYBS can deal with pointer variables easily. Actually, a pointer variable is just a variable that stores the addresses of data or instructions, so, it can be treated as a common variable during the dynamic binary analysis.

The loop structure is another tough problem for static binary analyzers, since they cannot statically discover how many iterations of a loop will take place, and this problem becomes worse when they encounter endless loops. To address the problem, many static solutions set an upper limit N for the maximum iteration number of a loop, so a loop will automatically terminate when the number of iterations is greater than N. For a dynamic analyzer, the objects analyzed are the real executing basic blocks, and the number of iterations for a loop depends on the specific execution of the target program. So, the number of iterations in DYBS is a specific value, and we do not need any special treatment for loop structures.

D. *Slicing Results Organization*

As mentioned previously, the CFG describes the execution order of basic blocks of a single function, and the CG describes the function calling hierarchy of the whole target program. The two data structures can clearly display the slicing results. However, since the slicing algorithm that is implemented in DYBS is a kind of dynamic inter-procedural analysis, the analyzed basic blocks are not from a single function, but from different functions on the execution path (that is, the extracted basic blocks are the dynamic executed blocks from the same execution path).

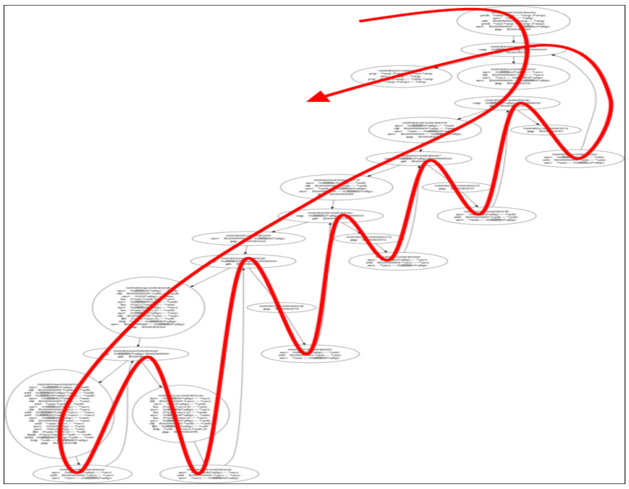

Figure 4. The CFG generated by inter-procedural slicing analysis.

For the purpose of reducing the workload of diagnosing attacks, DYBS takes the execution path as the main line to organize the slicing results. As an example, Fig.4 shows part of the result of inter-procedural dynamic program slicing for a matrix multiplication target program. The oval nodes represent basic blocks, and each block is identified by the address of its first instruction. The red line in the figure shows the execution sequence.

F. *System Optimization*

Like many other dynamic analyzing systems, DYBS also has tremendous overhead during its execution. This section provides a Function Call Filtration mechanism to optimize the proposed framework. The Function Call Filtration mechanism first inspects the slicing propagation behaviors of certain parts of the target program, and determines whether these parts will affect the slicing criterion or not. If it is known in advance that some parts will not affect the slicing criterion and they are stored in the library, then, the slicing process can skip them to avoid processing them in the target program.

A) *API Checking*

In most cases, systems have to use a large percentage of their resources (space and time) to process API functions. However, the features of most APIs in the system library can be determined, and it is not necessary to analyze them in every instance. So, the API inspecting is a key part of this optimization mechanism.

Actually, according to a common observation, a large part of the binary code in software is directly loaded from the system library, such as kernel32.dll, USER32.dll and ntdll.dll. The behaviors of these modules are predictable and it is not necessary to check every instruction in them. The Function Call Filtration mechanism tries to check the propagation behaviors of these API functions and then skip them in the analyzing process. In the first place, it has to examine the source code and the propagation behaviors of these APIs and store them in the user defined Function Library.

Table 1 shows the sample of propagation behaviors of some APIs. In the table, column *RetValNum* and *ParaNum* represent the number of return values and the

TABLE 1.
THE RELATED INFORMATION OF A PART OF THE USER-DEFINED FUNCTION (BEHAVIORS) LIBRARY

FunctionName	**ModuleName**	**RetValNum**	**RetValLen**	**ParaNum**	**SlicingPro**
GetModuleHandleA	Kernel32.dll	1	32	1	<1,ret>
SetLastError	Kernel32.dll	0	0	1	<1,null>
LstrlenW	Kernel32.dll	1	32	1	<1,ret>
CloseHandle	Kernel32.dll	1	32	1	<1,null>
RemoveDirectoryW	Kernel32.dll	1	32	1	<1,null>
UnmapViewOfFile	Kernel32.dll	1	32	1	<1,null>
CreateFileW	Kernel32.dll	1	32	7	<1,ret>
LoadStringW	User32.dll	1	32	4	<1,ret>,<1,2>,<1,3>
CharUpper	User32.dll	1	32	1	<1,ret>
--initterm	MSVCRT.dll	1	32	2	<1,ret>
wcsncpy	MSVCRT.dll	1	32	2	<2,1>,<2,ret>

number of parameters of each API function. *RetValLen* represents the length of the return value. *SlicingPro*, formally represented as <slicing source, slicing destination>, shows the slicing propagation features of each API function. The angle bracket means the slicing destination will affect the slicing source, i.e. the source operand of the instruction of the slicing source is the destination operand of the instruction of the slicing destination. For example, <1, ret> means the 1st parameter will be affected by the return value, and <1, 2> means parameter 2 will affect parameter 1.

B) Function Call Filtrating

After the propagation behaviors of APIs are summarized, some principles are used to guide the decision of the Function Filter:

(1) If the code of a function does not propagate the slicing features, i.e. it does not affect and is not affected by the code of other functions, then the slicing result set does not change after the slicing algorithm is performed on this function. So this kind of function should be skipped.

(2) If a function performs some operations to eliminate the slicing propagation features of its parameters, then the slicing propagation features of these parameters should be removed from the slicing result set.

(3) If a function can propagate the slicing features, then the affected destination data should be incorporated in the slicing results.

With these principles, irrelevant API functions can be free from instrumenting and analyzing.

IV. EXPERIMENTAL RESULTS

This section describes experiments to evaluate the efficiency of DYBS. Two compare the overall performance among DYBS, Native, and other attack analyzing tools. Two examine effectiveness of DYBS.

TABLE 2.
CONFIGURATION DETAILS OF THE EXPERIMENTAL ENVIRONMENT

Hardware/Software	**Configurations**
CPU	Intel Core2 (3.0GHz)
RAM	DDR2-667 (2GB)
HD Disk	Seagate SATA (250GB 5400RPM)
OS	WindowsXP SP3

Finally, we use several widely used applications to test the accuracy and practicality of DYBS. Table 2 describes the hardware and software configuration of the experimental environment.

A. Efficiency Evaluation

The two experiments used the SPEC CINT2006 benchmarks (on Windows) to evaluate the efficiency of DYBS. Fig.5 shows the normalized execution time (the ratio of execution time of DYBS to the time of Native ones, as described in equation (1)) of DYBS.

$$\text{Efficiency}=\frac{\text{DYBS(or DYBS\_No\_Opt)}}{\text{Native}} \qquad (1)$$

Since DYBS was constructed on DynamoRIO, the execution time shown as *Native* in the figure is the time of the tested benchmarks running directly on the underlying DynamoRIO platform. The results for *DYBS_No_Opt* are the time on DYBS without any optimizations. The results shown for *DYBS* are the time on the complete DYBS system, with Function Call Filtration optimization. Results are normalized, with Native performance treated as 1, so the results for DYBS and *DYBS_No_Opt* show how much more execution time they require than Native. From Fig.5, we see that the non-optimized version of DYBS needs 3.15x the time of the Native program. However, when the Function Call Filtration optimization mechanism is introduced to the system, the time cost is reduced remarkably, and the average time overhead is only 1.41x that of Native execution.

Fig.6 shows the average overhead comparisons between DYBS and other popularly used binary-level attack diagnosing tools: TaintCheck [24], LIFT [26], Dytan [28], Panorama [29]. Compared to tools such as

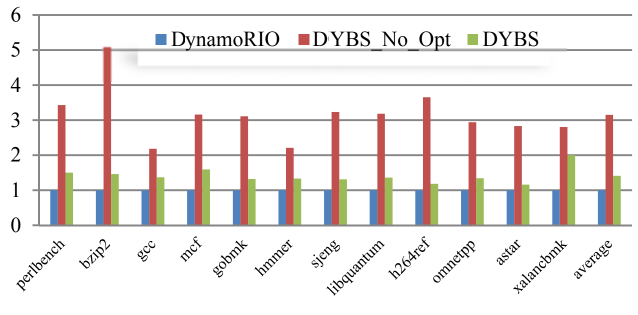

Figure 5. the slicing efficiency evaluation of DYBS.

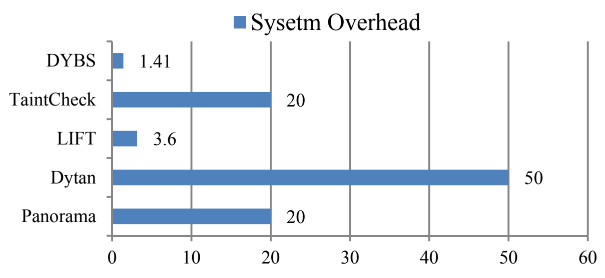

Figure 6. System overhead comparison between DYBS and other attack diagnosis tools.

Panorama (slowed down the target programs by 20x on average), Dytan (50x), LIFT (3.6x) and TaintCheck (20x), DYBS inflicted much lower runtime overhead (1.41x). DYBS achieved this better performance mainly because of the slicing algorithm, the Function Call Filtration optimizing mechanism, and the efficiency of the underlying DynamoRIO system.

B. *Effectiveness Evaluation*

This subsection presents two experiments, function analysis and inter-procedural analysis, to evaluate the effectiveness of DYBS. The tested applications that been selected in these experiments were all widely used in both personal computers and Internet.

A) Inner-procedural Dynamic Binary Slicing

As described in Table 3, the target functions were both mature and small applications. *Ping* is a computer network administration utility used to test the reachability of a host on an Internet Protocal (IP) network and to measure the round-trip time for a message sent from the originating host to a destination computer. *Netstat* (network statistics) is a command-line tool that used to display network connections (both incoming and outgoing), routing tables, and a number of network interface statistics. It is useful for finding problems and determining the amount of traffics in the network. The *Tracert* command is used to trace the route of a network packet and to determine the number of hops required for the packet to get to its destination. *Comp* is a simple command that compares two groups of files to find information that does not match. *Findstr* is a command used in MS-DOS to find a specific string of a plain text.

In the experiment, the entry address of each tested function is set as the slicing criterion, and DYBS is employed to get the interesting parts of the tested target programs. In Table 3, the column *AppNa* represents the name of the target function, *FEAdd* is the entry address of the corresponding target program, *NOIns* represents the number of instructions in the target program, *FSliCri* the forward slicing criterion, *FSINum* the number of sliced instructions with forward slicing, *FSliRate* the forward slicing rate, *BSliCri* the backward slicing criterion, *BSINum* the number of sliced instructions with backward slicing, and *BSliRate* the backward slicing rate. From Table 3, we can see that the number of the sliced instructions of the tested programs was much less than in the original programs with either forward or backward slicing. In the table, the results of the slicing rate were derived from equation (2):

$$\text{Slicing Rate}=\frac{\text{Sliced insturctions number}}{\text{Target program insturctions Number}}\times 100\% \qquad (2)$$

B) Inter-procedural Dynamic Binary Analysis

This experiment is carried out to verify the effectiveness of DYBS across all functions of the target program. The tested programs in the experiment are all commonly used in computers. *Notepad* is a simple text editor in Windows, *Calc* is a calculator in Windows, *Matrix* is a program for calculating matrix multiplication, and *gzip* is an application for compressing and uncompressing files. In the experiment, the input of each program is set as the slicing criterion. Table 4 displays the results. The column *APPNa* has the name of the target application, *OriCallNun* the number of calls in the original target program, *SliedCallNum* the number of function calls in the slicing result, *FunSliedRate* the function slicing rate, *OriBBNum* the number of basic blocks in the target program, *SliedBBNun* the number of basic blocks in the slicing result, and *BBSliedRate* the basic block slicing rate. The results for *FunSliedRate* and *BBSliedRate* are derived from equations (3) and (4) respectively:

TABLE 3.
THE SLICING RESULTS OF A SINGLE FUNCTION

AppNa	**FEAdd**	**NOIns**	**FSliCri**	**FSINum**	**FSliRate**	**BSliCri**	**BSINum**	**BSliRate**
ping	0x1002b22	108	ebp	52	48%	ebp	61	56%
netstat	0x1004fe0	21	eax	12	57%	eax	2	10%
tracert	0x1001591	184	esp	84	46%	esp	108	59%
comp	0x1002ee7	52	esp	23	44%	esp	21	40%
findstr	0x1002ca7	51	esp	14	27%	esp	18	35%

TABLE 4.
THE INTER-PROCEDURAL DYNAMIC BINARY SLING RESULTS OF DYBS

AppNa	**OriCallNun**	**SliedCallNum**	**FunSliedRate**	**OriBBNum**	**SliedBBNun**	**BBSliedRate**
Notepad	59	15	25.4%	1790	175	9.7%
Calc	95	9	9.5%	1581	192	12.1%
gzip	457	13	2.8%	388	43	11.1%
Matirx	169	1	0.6%	109	69	63.3%

$$\text{FunSliedRate} = \frac{\text{Number of sliced function calls}}{\text{Number of function calls in Target program}} \times 100\% \quad (3)$$

$$\text{BBSliedRate} = \frac{\text{Number of sliced basic blocks}}{\text{Number of basic blocks in the target program}} \times 100\% \quad (4)$$

From table 4, we see that the number of function calls and basic blocks are sharply decreased in the slicing results.

C. Accuracy and Practicality Evaluation

In this experiment, four popularly used applications are selected to test the accuracy and practicality of the framework. The experiment tries to use DYBS to discover vulnerabilities regardless they are the known attacks or potential vulnerabilities. Table 5 provides the analyzed results by running *hangul HWP* (a word processing software for Korean), *JustSystems Ichitaro* (a word processing software for Japanese), *IrfanView* (a free graphic viewer for Windows), and *Foxit Reader* (a widely used document processor for Chinese) on DYBS.

In Table 5, the column *Attacks* contains the number of attacks that incorporated in the tested target programs. Actually, all attacks in the tested target programs are defined by the CVE (Common Vulnerabilities & Exposures) vulnerabilities library. The *Attack Source* describes the attack source for the corresponding target program. The *Discovered Attacks* describes the number of attacks of the target programs that discovered by DYBS. From the data that shown in the table, we see that all the predefined (by CVE library) attacks are discovered by DYBS, so the recognition ratio is 100%.

In the experiment, forward program slicing is used to find the potential vulnerabilities of the tested target programs. During this process, the inputs of each program are set as the slicing criterion. So, if there are functions (e.g. strcpy()) that might cause buffer overflow, DYBS would give an alert for a potential vulnerability. In Table 5, the column *Potential Vulnerabilities* gives the number of potential vulnerabilities that are discovered by DYBS.

Since the CVE library does not contain the vulnerability information for the hangul HWP target program, and we cannot define the vulnerabilities ourselves, the corresponding columns of the hangul HWP target program are set null.

V. Related Work

Since it is an effective approach, much research focuses on employing program slicing technology to detect vulnerabilities and improve the security of software. PSE [16] is a static program slicing technique for diagnosing program failures. It is precise because of its consideration of error conditions. It is similar to Das's earlier work, ESP [17], a symbolic dataflow analysis engine. Using program slicing, Monate [8] introduces an automatic source-to-source method to preserve the confidentiality of the target program.

As well as the static program slicing technique, many researches are focusing on its dynamic counterpart. Pan [18] presents a family of heuristics for fault localization using dynamic slicing. Kamkar et al. [19] present a generalized version of an algorithmic debugger, a method of semi-automatic bug localization. By using dynamic program slicing, they can compute which parts of the target program are relevant for the research. Based on the barrier slicing, Cellato [20] proposes a solution to identify the parts of the client code that have to be moved to the server to protect unsafe variables. He also investigates the trade-off between security loss and performance overhead of his method [21]. In the process of dynamic slice computation, different types of information are computed and then discarded after the computation of dynamic slicing. Korel et al. [22] first exploit the features of these kinds of information (e.g. executable dynamic slices, partial dynamic slicing, influencing variables, and contributing nodes), then incorporate them into their dynamic slicing tool to improve the process of program debugging. Similar to Korel's work, Tibro [23] also introduces a forward computation method for relevant slices; it requires less space. By integrating the potential of a delta debugging algorithm with forward and backward dynamic slicing, Neelam [10] narrows down the scope of the search for the faulty code.

Researches on static and dynamic program slicing mentioned above are all based on the availability of source code of the target programs. However, the source code of many programs is not easy to obtain in practice, which makes this type of slicing computation impossible.

There are also many dynamic binary taint analysis tools which, likes DYBS, work with object code. TaintCheck [24] is a runtime taint analysis approach; that automatically detects most types of attacks on binary-level applications. TaintCheck uses the heavy weight binary instrumentation framework Valgrind [25], so its overhead is high. LIFT [26] is a software-only information flow tracking system that uses StarDBT [27] for detecting software attacks on x86 binary applications. During the execution of the target program, LIFT first dynamically instruments the binary code and tracks its information flow, then, if unsafe data is detected, it

TABLE 5.
ATTACKS DIAGNOSING AND POTENTIAL VULNERABILITIES DISCOVERING IN DYBS

Target Program	Attacks	Discovered Attacks	Potential Vulnerabilities	Attack source
hangul HWP	/	/	3	/
JustSystems Ichitaro	3	3	1	CVE-2010-3915,CVE-2010-3916
Irfan View 4.25	32	32	3	CVE-2010-1509
Foxit Reader 3.0 build 1120	22	22	17	CVE-2009-0836,CVE-2009-0837

switches the program control flow to process it. Furthermore, LIFT is a system that emphasizes efficiency, and employs three binary optimization methods, Fast Path (FP), Merged Check (MC) and Fast Switch (FS), to optimize its performance. Dytan [28] is also a dynamic taint analyzing system for x86 binaries. Dytan is designed to be general and flexible, so it allows for implementing different kinds of techniques based on dynamic taint analysis with little effort. However, its general and flexible features seriously degrade the performance of the system. Panorama [29] is a hardware-assisted flow tracking system that is implemented on QEMU [30] for detecting and analyzing malicious software on commodity desktops.

DYBS is also a dynamic binary taint analysis framework implemented using a dynamic instrumentation system; however, the way of achieving the taint analysis goal of DYBS is different from the frameworks mentioned above. In DYBS, the dynamic program slicing mechanism is employed to analyze the target programs. For the same reason, the performance of DYBS is much better than the other systems (as described in Fig.6).

VI. Conclusions and Future Work

DYBS is a dynamic binary program slicing framework, and it is designed to diagnose attacks in binary-level target programs. During the execution, DYBS first gathers the profile information by deploying the analyzing instructions in the target program. However, once an attack is encountered, normal execution terminates, and backward program slicing is employed to slice the target program. Using the attack point as the slicing criterion, DYBS slices out basic blocks related to the slicing criterion and organizes them into an executable subset. DYBS constructs a CFG to organize basic blocks in each function, and builds a CG for the whole target program to implement inter-procedural program slicing. Based on the organized and structured slicing results, users can locate and diagnose attacks easily. Furthermore, the Function Call Filtration optimization mechanism is proposed to optimize the process of dynamic slicing. Results of the experiments on SEPC CINT2006 benchmarks and several popularly used applications show that DYBS is efficient, accurate, and practical.

The results to data have been promising and research is continuing on extending and improving the framework. The extensions and improvements include: (1) contriving better optimizing methods to improve the efficiency of the framework; (2) strengthening the ability to analyze large-scale target programs to improve the accuracy of the framework; (3) expanding the usage scope to improve the practicability of the framework.

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 (863 Program) of China (Grant No. 2012AA010905), the National Basic Research Program (973 Program) of China (Grant No. 2012CB723401).

References

[1] Wen Jiang, Xin Fan, Dejie Duanmu, Yong Deng. A New Security Risk Assessment Method of Website Based on Generalized Fuzzy Numbers. Journal of Computers, 8(1), 136-145, Jan 2013.

[2] Weihui Dai, Qi Zhu, Chunshi Wang, Yujiao Zeng. Risk Management Model of Information Security in IC Manufacturing Industry. Journal of Computers, 7(2), 317-324, Feb 2012.

[3] Xiu-qing YU.Internal P-set and Security Transmission-identification of Information. Journal of Computers, 6(10), 2249-2254, Oct 2011.

[4] Chris McNab.Network Security Assessment: Know Your Network (2nd Edition).O'Reilly Media, 2007.

[5] Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, Haibin Guan. Combining Static and Dynamic Analysis to Discover Software Vulnerabilities. In: Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS '11), June 2011, pp.175-181.

[6] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, Lin Chen. A brief survey of program slicing. ACM SIGSOFT Software Engineering Notes 30(2) (2005) 1-36.

[7] N.Sasirekha, A.Edwin Robert, Dr.M.Hemalatha. Program slicing techniques and its applications. International Journal of Software Engineering & Applications 2(3) (2011) 50-64.

[8] Benjamin Monate, Julien Signoles. Slicing for Security of Code. In: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications (Trust '08), March 2008, pp.133-142.

[9] Dasarath Weeratunge, Xiangyu Zhang, Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In: Proceedings of Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10), March 2010, pp.155-166.

[10] Neelam Gupta, Haifeng He, Xiangyu Zhang, Rajiv Gupta.Locating faulty code using failure-inducing chops. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05), November 2005, pp.263-272.

[11] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, Bogdan Korel. Theoretical foundations of dynamic program slicing. Theoretical Computer Science 360(1) (2006) 23-41.

[12] Rajiv Gupta, Mary Lou Soffa , John Howard. Hybrid slicing: integrating dynamic information with static analysis. ACM Transactions on Software Engineering and Methodology 6(4) (1997) 370-397.

[13] Xiangyu Zhang, Rajiv Gupta, Youtao Zhang. Cost and Precision Tradeoffs of Dynamic Data Slicing Algorithms. ACM Transactions on Programming Languages and Systems 27(4) (2005) 631-661.

[14] Xiangyu Zhang, Rajiv Gupta. Cost effective dynamic program slicing. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04), June 2004, pp.94-106.

[15] Derek Bruening, Qin Zhao, Saman Amarasinghe. Transparent Dynamic Instrumentation. In: Proceeding of

Eighth Annual International Conference on Virtual Execution Environments (VEE '12), March 2012, pp.133-144.

[16] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, Zhe Yang. PSE: explaining program failures via postmortem static analysis. In: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering (SIGSOFT '04), October 2004, pp.63-72.

[17] Manuvir Das, Sorin Lerner, Mark Seigle. ESP: path-sensitive program verification in polynomial time. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI '02), June 2002, pp.57-68.

[18] Hsin Pan, Eugene H. Spafford. Heuristics for Automatic Localization of Software Faults. Technical Report, SERC-TR-116-P, Purdue University, 1992.

[19] Mariam Kamkar, Nahid Shahmehri, Peter Fritzson. Bug Localization by Algorithmic Debugging and Program Slicing. In: Proceedings of the 2nd International Workshop on Programming Language Implementation and Logic Programming (PLILP '90), August 1990, pp.60-74.

[20] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, Paolo Tonella. Barrier Slicing for Remote Software Trusting. In: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '07), September 2007, pp.27-36.

[21] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg. Trading-off Security and Performance in Barrier Slicing for Remote Software Entrusting. Automated Software Engineering 16(2) (2009) 235-261.

[22] Bogdan Korel, Jurgen Rilling. Application of Dynamic Slicing in Program Debugging. In: Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG '97), 1997, pp.59-74.

[23] Tibor Gyimóthy, Árpád Beszédes, Istán Forgács. An Efficient Relevant Slicing Method for Debugging. In: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-7), September 1999, pp.303-321.

[24] 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.

[25] Nicholas Nethercote, Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07), June 2007, pp.89-100.

[26] 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), 2006, pp.135-148.

[27] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz Jr., Zhiwei Ying, Youfeng Wu.StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. Lecture Notes in Computer Science 2007(4697) (2007) 4-15.

[28] 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.

[29] 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.

[30] Daniel Bartholomew. QEMU: a multihost, multitarget emulator. Linux Journal 2006(145) (2006) 41-46.

[31] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00), June 2000, pp.1-12.

Erzhou Zhu is currently a lecturer with the Faculty of Computer Science, 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.

Feng Liu is currently a professor with the Faculty of Computer Science, 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.

Xianyong Fang received Ph. D. degree from Zhejiang University in 2005. He worked as a Postdoc in LIMSI-CNRS from 2007 to 2008. He is a professor of Anhui University and a committee member of the Intelligence CAD and Digital Art Committee of Chinese Association for Artificial Intelligence. His research interests focus on image/ video processing related graphics and vision topics.

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

Yindong Yang received the Ph.D. degreeat Department of Computer Science and Engineering (2012), Shanghai Jiao Tong University, Shanghai, China. His main research interests are in virtual machines, computer architecture, and compiling.

Alei Liang is an assistant professor in the School of Software at the Shanghai Jiao Tong University (SJTU). He received his Ph.D. in Computer Science and Engineering from Shanghai Jiao Tong University in 2005. His research interests are distributed computing, virtualized security, model checking, and program analysis and embedded systems.