

A Survey on Tools for Binary Code Analysis

Shengying Li
lshengyi@cs.sunysb.edu
Stony Brook University
August 24, 2004

Abstract

Different strategies for binary analysis are widely used in systems dealing with software maintenance and system security. Binary code is self-contained; though it is easy to execute, it is not easy to read and understand.

Binary analysis tools are useful in software maintenance because the binary of software has all the information necessary to recover the source code. It is also incredibly important and sensitive in the domain of security. Malicious binary code can infect other applications, hide in their binary code, contaminate the whole system or travel through Internet and attack other systems. This makes it imperative for security personnel to scan and analyze binary codes with the aid of the binary code analysis tools. On the other hand, crackers can reverse engineer the binary code to assembly code in order to break the secrets embedded in the binary code, such as registration number, password or secret algorithms. This motivates researches to prevent malicious monitoring by binary code analysis tools. Evidently, binary analysis tools play an important double-sided role in security.

This paper surveys binary code analysis from the most fundamental perspective views: the binary code formats, several of the most basic analysis tools, such as disassembler, debugger and the instrumentation tools based on them. The previous research on binary analysis are investigated and summarized and a new approach of analysis, disassembler-based binary interpreter, is proposed and discussed.

1. Introduction on Binary and Binary Analysis Tools

Software industry is one of the most promising areas in the current world. Each year, software companies produce thousands and millions of software products. All software products are ultimately translated to binary before execution, independent of the high-level languages used in the source code. In other words, binary code is one of the lowest representations of software.

The state-of-the-art program analysis tools work best at the source code level, because it can use much more high-level information than that present at binary code level. Why is binary code still interesting besides source code? That is because binaries have their own charms in research. The most important point is that all secrets of a software exist in its binaries. With necessary skill and patience, it is possible to reveal all the secrets of the software from binaries. Since most of the commercial software products, especially on Windows platform and malicious software, such as virus, Trojans, spyware, are distributed in the form of binary code, it becomes extremely important to explore the methods for analyzing binary codes.

Compared to source code, binary code has other obvious advantages; for example, it is convenient to execute but difficult to understand. Once generated in one machine, the binary code is self-contained and all static library routines are bound into the binary file. It has good portability; it can be fed into any other machines with the same instruction set on hardware, and can be executed simply and immediately. On the other hand, binary is much harder to understand than source code written in high-level programming languages, since everything inside binary code is represented with 0 and 1. This feature is very helpful for the protection of privacy of software. These advantages are so useful and practical that software companies prefer to distribute their products in the form of binary.

Source code of some old software may be lost and only binary code is left. It is hard, if not impossible, to recover the original source code from the representation of binary code. A lot of research has been done in the last few years to reverse engineer binary code back to high-level languages. Cifuentes discusses several possible solutions of the problem in her thesis and papers [CIG96] including how to construct inter-procedural data structure, how to build control graph, and so on. Furthermore, binary analysis is very important to protect a system from attacks. Since most of applications are in binary forms, such as Windows applications, security protection with the aid of binary analysis does not require source code and avoid a lot of trouble on legitimacy.

On the flip side, the advantage of binary analysis can be used for malicious purposes. Malicious users can cause serious problems threatening the privacy of software products with binary analysis. This is one of the critical issues of software safety. Since binaries contain all secrets of the software, malicious users may apply tools to crack the binaries and reveal the underlying secrets hidden in them by reverse engineering. Here, reverse engineering means to take the binary code or the low level instructions executed in the processor and extract information of the high level language. Most of the publicly available information about reverse engineering is available at sites dedicated to software piracy and cracking software protection schemes. For years the most famous resource for crackers was [Fravia's website](#)[FRAVIA]. While the collection of tutorials and well documented techniques for cracking makes it invaluable resource for aspiring “reverse

engineer”, cracking software causes thousands or even millions of dollars’ loss for software companies every year.

The development of cracking techniques also invokes a prevalent research direction, software protection that prevents the software from being reverse engineered. Cracking and protection seem like an endless game. Both of them need to use binary analysis to understand the binaries, while protection requires much more than cracking. The reason is that cracking only needs to understand the logic of the code, find the security sensitive parts and disable or modify them; the protection system needs to understand binaries, build up the defense system, insert them into security sensitive parts and furthermore, prevent both original binaries and the defense system from being understood by reverse engineering.

Secondly, malicious software may threaten the safety of user’s system or machine. Malicious software, such as virus and Trojans, are distributed in binary code, and hide their own binary code within the victim binaries. When the original binary code executes, the malicious code will get executed as well and infect more binaries. It may take control of the system with the highest privilege, and disrupt entire system. To make things worse, with increasing popularity of Internet, network based attacks, like network-aware worms, DDoS agents, IRC Controlled bots, spyware, and so on has grown. The infection vectors have changed and malicious agents now use techniques such as email harvesting, browser exploits, operating system vulnerabilities, and P2P networks to spread. Basically, network system transfers data in the form of packets, but it does not inspect what data payload the packets carry. No matter what the packets are, the network system will assemble or disassemble the packets and transfer them faithfully as long as the headers of the packets comply with the network protocol. However, it can be very dangerous in the real world. Virus or worms can travel by network in the form of binary code to all around the world, become activated in the contaminated system and cause serious damage.

This paper focuses on the most fundamental aspects of binary area: binary code’s formats and its most basic analysis tools, such as disassemblers, debuggers and so on. These are the basis of all other advanced binary tools, but unfortunately, until now, there is no paper with enough details on it. Computer simulator and emulator are also investigated in this paper, since they are convenient to be used for the purpose of binary analysis. For each of them, I will give an overview, fundamental skills and related problems. Specific challenges in implementation will be discussed, their disadvantages and advantages will be compared, and the existing state-of-the-art tools will be introduced. I will concentrate on the tools to reverse engineering from binary to assembly code, the technology to analyze information in binary code and its corresponding assembly code, and related applications in the security domains.

Relevant tools for analysis and instrument binaries are also discussed, which have a broad range of applications. For instance, the anti-virus software can use disassemblers and debuggers to scan the data of packets and monitor traffic in a network system. Once the tool finds some suspicious code, it can stop it and prevent the damage in advance. Additionally, with the help of binary analysis tools, even normal users and administrator are able to determine if the binaries are harmful by examining them manually [REM04]. Furthermore, based on analysis information on binaries, security

researchers are able to embed security instrumentation codes into the original programs to protect their binaries [SL02].

The organization of this paper includes six sections. Section 2 presents the overview of different binary forms. Section 3 discusses the static analysis tool of disassembler, followed by section 4 introducing the dynamic tool as debugger and the more complicated emulator, which is, however, hard to be implemented. Section 5 describes state-of-the-art reverse engineering and tamper resistant software techniques. Section 6 proposed a new approach to implement a binary analysis tool for security, combining the advantages of disassemblers and debuggers, and achieving better performance both in time and accuracy. Finally, conclusion and future work are presented at the end of the paper.

2. Binary Object File Formats and Contents

Let's look at how an executable object code is generated. An object file is generated from source code in a high-level language by compilers or from low-level assembly code by assemblers. Then, linkers combine multiple related object files into one executable file to fulfill multiple functionalities in one final object file. At run time, loader loads object files into memory and starts execution. This section introduces several popular object files' formats, structures and contents, and how these object files start to run on their corresponding operating system.

Basically, an object file may contain five fundamental type of information, as shown in figure 1.1 : (1) Header information, which is overall information about a file, such as the size of code, creation dates and so on; (2) Relocation information, which is a list of addresses in the code that need to be fixed when the object file is loaded into an address different from the expected loading address; (3) Symbols: which are global symbols and mainly used by linkers, such as the symbols imported from other modules or exported to other modules; (4) debugging information: which is used by debuggers, such as source file and line number information, local symbols, description of data structure (e.g. structure definition in C language); (5) code and data, which are binary instructions and data generated from source file and are stored in the sections. John R. Levine [JL00] describes detail information of the object format and its operations under the control of linkers and loaders.

File Header
Relocation Table
Symbol Table
Debugging Table
Section 1
Section 2
...
...
Section n

Figure 1.1 Binary File Format General Abstraction

According to the usage of an object file, it falls into several different categories separately or with some combination of them:

- Linkable object file: used as input by a linker or linking loader. It contains a lot of symbols and relocation information. Its object code is usually divided into many small logical segments that will be treated separately and differently each time by the linkers.
- Executable object file: is loaded into memory and runs as a program. It contains object code, usually with page alignment to allow the whole file to be mapped into address space. Usually it doesn't need symbols or relocation information.
- Loadable object file: is able to be loaded into memory as a library along with other programs. It may consist of pure object code or may contain complete symbol and relocation information to permit runtime symbolic linking according to different systems' runtime environments.

With different systems, object files have quite a number of different formats. The most popular ones include MS-DOS.com files, Unix a.out files, ELF files, and Windows PE format and so on.

.COM file and .EXE file for MS-DOS

The simplest MS-DOS.COM object file is a null object file, i.e., it only contains pure executable binary code, and no other information. In Windows DOS, the address from 0 to FF is named as Program Segment Prefix (PSP), which contains arguments and parameters. At run time, an object file is loaded into a free chunk of memory address starting from the fixed address, 0x100. All segment registers are set to point to the PSP, and SP (stack pointer) points to the end of the segment. When the size of the object file is larger than one segment, it is programmer's responsibility to fix the addresses using explicit segment numbers to address the program and data.

MS-DOS.EXE file is an object file with relocation information besides data and code. It has relocation entries that indicate the places in a program where addresses need to be modified when the program is loaded. The reason is that 32-bit Windows gives each program its own address space and each program can require a desired loading address, but it doesn't always load the program at the required address. Figure 1.2 explains header format of a .EXE file, indicating the size of code by lastsize and nblocks, the related relocation information with relocs, nreloc.

```
Char signature[2] = "MZ"; // magic number
Short lastsize; // # bytes used in last block
Short nblocks; // the number of 512-byte blocks in the code
Short nreloc; // number of relocation entries
Short hdrsize; // size of file header in 16 byte paragraphs
Short minalloc; // minimum extra memory to allocate
Short maxalloc; // maximum extra memory to allocate
Void far *sp; // initial stack pointer;
Short checksum; // ones complement of file sum
Void far *ip; // initial instruction pointer;
Short noverlay; // Overlay number, 0 for program
Char extra[]; // extra material for overlays, etc.
Void far *relocs[]; // relocation entries, starts at relocpos
```

Figure 1.2 Format of .EXE file header

a.out file and ELF file for UNIX

The a.out is effective for relatively simple systems with paging mechanism. It contains separate code section and data section. The original motivation for doing this is that PDP-11 only supports 64K address space, which is insufficient to contain both code and data. So it allocates two-address space for code and data separately, each with 64K. 286 or 386 in 16-bit protected mode also have individual addresses for code and data. The more important advantage is that different data sections allow the same code section to be shared at run time when a program runs multiple times. Modern object file keeps this feature.

The more complicated a.out supports relocation. Its header is shown in the figure 1.3. Unix system uses a single object format for both run-able and linkable files. Relocation entries serve two objectives: (1). the addresses, which are needed to be fixed up when the section of code is relocated to a different base address; (2). the referenced addresses to symbols, which are undefined in current object file and will be patched by linkers when the symbols are finally defined. But it is out of popular for two reasons: it is hard for an a.out to support dynamic linking; it doesn't support C++ very well, which has special requirement for initialization and finalization code.

```
text section  
data section  
text relocation  
data relocation  
symbol table  
string table
```

Figure 1.3 a.out header

In order to support cross-compilation, dynamic linking and other modern system features, Unix System V began to use a newer format other than the traditional a.out format. Initially, System V used COFF, Common Object File Format [COFF]. COFF was for cross-compiled embedded systems and didn't work well for a time-sharing system, since it couldn't support C++ or dynamic linking without extensions. Then ELF [ELFM][LINK], Executable and Linking Format, superseded COFF and overcame the shortcomings of COFF. ELF is also a popular file format in the freeware Linux and BSD variants of Unix.

Interesting, ELF files have an unusual dual nature, as what is shown in Figure 1.4.

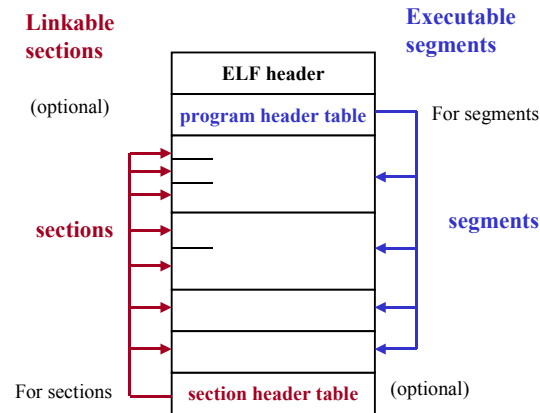


Figure 1.4 Two Views of an ELF file.

Besides an ELF header, ELF has two optional header tables: a program header table or a section header table. Usually, a linkable file has a section table, an executable file has a program header table and a shared object has both tables. Using the section header table, compilers, assemblers and linkers take the file as a set of logical sections, such as code section, data section, symbol section and so on. By the program header table, the loader of system takes the file as a set of segments, such as read-only segment for the code and read-only data, read-write one for read/write data. A single segment usually consists of several sections. For example, a “loadable read-only” segment contains sections of executable code, read-only data and symbols for dynamic linking. By compacting sections into segments, the system can map the file with one or two operations.

```

int sh_name;           // name, index into the string table;
int sh_type;          // section type;
int sh_flags;         // flag bits;
int sh_addr;          // base memory address, if loadable, or zero;
int sh_offset;        // file position of beginning of section;
int sh_size;          // size in bytes;
int sh_link;          // section number with related info or zero;
int sh_info;          // more section-specific information;
int sh_align;         // alignment granularity if section is moved;
int sh_entsize;       // size of entries if section is an array;

```

Figure 1.5 Section headers

A linkable or shared object file is considered to be a set of a collection of sections. Figure 1.5 has the description of a section header, which contains all information for the section tables, such as the name, type, flags, addresses, size, alignment and so on. Each section stands for a single type of information, for example, program code, read-only data, relocation entries or symbols. The type of section is meaningful to the linker to do the specific processing.

```

int type;           // loadable code or data, dynamic linking info, etc.
int offset;        // file offset of segment
int virtaddr;      // virtual address to map segment
int physaddr;      // physical address, not used;
int filesize;     // size of segment in file;
int memsize;       // size of segment in memory
int flags;         // Read, Write, Execute bits
int align;         // alignment requirement

```

Figure 1.6 ELF program headers

An executable object file is considered to be a set of a collection of segments, whose table is in the program header in figure 1.6. The main difference between executable and linkable ELF file is that in executable file, data in the file are arranged to be mapped into memory and run. The executable ELF file looks up the program header, which defines the segments to be mapped, such as read only text, or read/write data segment. An executable usually combines several loadable sections into one appropriate segment, so that the system can map the file with a small number of operations. An ELF shared object contains both section table and segment table, since it needs to play double responsibilities to be linked and to be executed.

PE format for 32 bit Windows

Microsoft Windows NT™ operating system brought significant changes to the object file format, and introduced a new Portable Executable (PE) file format, which draws primarily from the COFF(Common Object File Format) common to UNIX operating system, and retains the old MZ header from MS-DOS for compatibility with previous versions of the MS-DOS and Windows operating system.

The PE format [MP94, MP02] is organized as a linear stream of data. It begins with an MS-DOS header, a real-mode stub, and PE file signature. Immediately following is a PE file header and optional header. Beyond that, all the section headers appear, followed by all of the section bodies. At the close of the file, there are a few other regions of miscellaneous information, including relocation information, symbol table information, line number information, and string table data. All of these features are shown in Figure 1.7.

```

MS-DOS MZ header
MS-DOS real mode stub program
PE file signature
PE file header
COFF header
PE file optional header
.text Section header
.bss section header
.rdata section header
...
.debug section header

```

Figure 1.7, PE file format

The real-mode stub program is an actual program run by MS-DOS when the executable is loaded. The program typically prints out a line of text, such as “This program needs Microsoft Windows.” PE signature is used to specify the intended operating system. The information in the PE header is basically high-level information that is used by system or applications to determine how to treat the file. It includes the machine type, number of sections, time stamp, number of symbols, size of optional header, characteristics, and so on. The PE optional header contains most of the meaningful information about the executable image, although its name is “optional header”. It includes important information, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information and so forth.

An application for Windows typically has nine predefined sections named .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug. Some applications do not need all of these sections, while others may define more sections to suit their specific needs. Some of them are special section that only appear in PE format:

- Exports: They are included in .edata section, with a list of the symbols defined in the module and visible to other modules. Typically, it is not contained in EXE files, but in DLLs, which export symbols for the routines and data that they provide. In order to save space, exported symbols can be referenced via small integers called export ordinals as well as by names.
- Imports: The import table in .idata section lists all of the symbols that need to be resolved at load time from DLLs. It includes the import directory and import address name table and based on the definition, user can retrieve the name of modules and all functions in each module that are imported by an executable file.
- Resources: The resource table is organized as a tree in .rdata section, and contains resource information for a module, such as cursors, icons, bitmaps, menus and fonts that are shared between the program and the GNU. The tree is typically three levels, resource type, name and language. Each resource can have either a name or numbers. A typical example can be type DIALOG (Dialog box), name ABOUT (the About This Program box), language English.
- Thread Local Storages (TLS): This contains private information for each thread of execution in one process. This section points to a chunk of the image for the initialization of TLS, when a thread starts. It also contains pointers to initialization routines to call when each thread starts.
- Fixups: The fixup table, contains an array of fixup blocks, each containing the fixup addresses for one 4K page of the mapped executable. Each fixup block contains the base relative virtual address of the page, the number of fixups, and an array of 16 bit fixup entries. When the executable is moved, it is moved as a unit so all fixups need to be patched with the same value, the difference between the actual load address and the target address.

When a PE executable file is running, the process is as follows: first, read in the first page of the file with the DOS header, PE header, section headers. After determining whether the target area of the address space is available, allocate another one if the

address space is not available. Then map all the sections to the appropriate place into the memory using the information in the section headers. If the address space is not the desired address, apply the fixups. Looking into the import table, load any DLLs that are not already loaded. Resolve all imported symbols in the imports section, create the initial stack and heap using values in PE header and create the initial thread and start the process.

3. Static Binary Analysis Tools

Disassembler, the reverse tool of assembler, is used to statically decode the binary code to its corresponding assembly representation or high-level language such as C or Pascal. In the case that the destination language is higher-level language than assembly code, it is also called as decompilation [CI95]. I discuss both the cases of disassembling from binary to assembly and to high-level language, because the difficult aspects of disassembling binary code to assembly code are usually closely related to data structures or procedures in high-level language; on the other hand, those difficult aspects are shared across disassembling to higher-level language in the initial steps.

As a static tool, a disassembler has several advantages, compared to a dynamic tool. First, a disassembler can analyze intermediate binary code that is unable to run, since it doesn't need to really execute the binary at all. Second, the time of disassembling is positively propagated with the size of code, while the time of dynamic tool is related to execution flow, which becomes much slower especially in the case of a loop with thousands and millions of iterations. Third, it has a good global view sight of the whole binary code and can figure out the entire program logic before running it, while debugger or other dynamic analysis tools can only have an idea of a "slice" of the program trace that is currently running at a given time.

For any tool of analysis or modification of the binary code, the basic requirement is to disassemble the code correctly. However, it is extremely hard for a disassembler to get the assembly code completely accurate.

3.1 Why is Disassembly Difficult?

The most difficult problem for a disassembler is to distinguish between code and data when some data is spread inside code section. One reason for data and code mixing is alignment data, which are inserted into code section for better performance with the consideration of hardware architecture requirement. They usually will not be executed at run time, and with the combination of useless instructions. For example, "INT 3", "nop" or "lea 0x0(%edi), %edi"(edi can be replaced by any other general registers) are favorite data for alignment, and which one to be used depends on compilers. They are not real code, but they have correct code formats, mix with other codes, and are distributed throughout the code section, that makes it hard for a disassembler to tell them from the real codes. The other reason for data and code mixing is the data inserted right after a branch instruction, such as data inserted manually or a jump table for switch-case instructions in C language generated by compilers. Jump table is combined with destination addresses of different branches for a switch instruction. Although in Linux, generated by GCC, jump table is produced in global data area, in windows it is generated inside the function of code section.

The second difficult point of disassembling is that a disassembler cannot get the dynamic information since it only works statically. Although one solution for separating data and code, such as data after branch instruction, is available by following the control flow of instructions, which can get “chunks” of code and skip data area, the static characteristic of disassembler prevents it from analyzing control flow accurately. Disassembler cannot extract correct value whenever dynamic information is involved, such as an indirect branch instruction, because usually the target of an indirect call or jump instruction cannot be computed until the instruction is executed at run time. Some freeware or commercial disassemblers, such as w32dism [W32D], IDApro [IDAP], are able to ignore the data after unconditional jump instruction; however, they cannot figure out the data following indirect call or complex conditional jump instructions.

The third difficult problem for disassembler is raised at variable instruction size. For example, in the X86 system, instructions have variable lengths, which make it more difficult for a disassembler to decide the end of an instruction.

3.2 Disassembly Principles at Assembly Level

Due to the above mentioned factors, i.e. data and code mixing, lack of dynamic information, and with variable instruction size, in the process of disassembly, the following principles are applied implicitly.

- **It cannot disassemble backward.**

Disassembling backward is thought impossible on many machines. The first reason is instruction's length is variable. From the end point of an instruction, it cannot know where is the starting point of the instruction, and therefore, it cannot get the opcode to analyze the instruction's correct length. The second reason is related to data distributed inside codes, since compiler or assembler may insert data between instructions. Overall, given an address that is the end address of a known instruction, there may be a large variety of legal sequences having different start addresses but the same ending addresses.

- **It can disassemble in a completely correct way until the first branch instruction, given that it starts from the beginning address of a legal instruction.**

Given a legal start address for an instruction, it is believed to be able to disassemble this instruction correctly. The instruction formats are designed such that there is one unique interpretation for every instruction. If a set of instructions is arranged close to each other, i.e., without data mixing among them, it must be able to sequentially disassemble all instructions correctly, since the end of the address of a previous legal instruction is followed by the beginning address of the next instruction. Let's consider the case where data is inserted into code section. Data is never in the middle of execution; otherwise it causes errors at run time. It can only be inserted into the addresses unreachable by execution flow. Therefore, data will only appear at the addresses after branch instructions, such as unconditional jmp, or unreachable branch of a conditional jmp instruction, call instruction or ret instruction. So, given an address that is known to be the start of a legal instruction sequence, it can disassemble correctly until the first branch instruction, since it is guaranteed that there is no data mixing till that point.

- **It is wrong if addresses for two or more instructions are overlapped.**

Instruction overlapping means that one instruction appears into other instructions partially or fully. Although at run time, dynamic technology, such as code patching

[SL02, FB92], may cause instructions' addresses overlapped by generating new instructions overlapping old instructions, in static time, instructions are definitely separated from each other. Obfuscation technology [SL02, CL03], gives some interesting scenarios on this issue. It can generate jmp instruction to jump into addresses that appear to be in the middle of "instruction". As a matter of fact, as shown in figure 2.1, this "instruction" is fake one in order to fool disassembler. In the figure 2.1, the fake instruction "mov eax, 10000559" is combined with the data "0xb8" after a call instruction with the real instructions "pop ecx" and "add eax, 401000h". Legal instructions must not be overlapped with each other, which is an important principle to maintain whether disassembly proceeds correctly or not.

```

call    tmptr      // tmptr contains address of lab1;
_emit  0xb8        // opcode for mov instruction.
lab1:
pop     ecx
add    eax, 401000h

by disassembling ==>

call    tmptr      // tmptr's value cannot be know by disassembler

mov    eax, 10000559 // wrong instruction
inc    eax          // wrong instruction

```

Figure 2.1 Obfuscated fake instruction: jump into part of other instruction.

3.3 Two Famous Algorithms

Numerous researches have been done in the area of disassembly. Two of the well-known algorithms are linear sweeping and recursive traversal.

Linear sweeping

It reads in the binary bytes sequentially and tries to match the bytes into instructions in turn, as shown in figure 2.2.

```

Procedure LinearDisasm(addr)
{
  while (startAddr <= addr <= endAddr) {
    I = decode instruction at address addr;
    Addr += length(); // Go to next sequential instruction
  }
}

```

Figure 2.2. Algorithm of linear sweeping

The advantage of this algorithm is that it is able to identify each instruction in a large chance, since it scans through the whole code section. But the algorithm is not able to distinguish between code and data. Data embedding in code will also be taken as instruction bytes, since the algorithm fetches bytes sequentially to translate them into instructions. A disassembler will fail to translate silently and may continually makes

quite a number of wrong instructions until it encounters bytes that doesn't match any valid opcode. Furthermore, it is impossible for the disassembler to know from where it has made errors continuously. GNU utility objdump [OBJD] and many link-time optimization tools are applying this algorithm.

Recursive traversal

The main weakness of the linear sweeping algorithm is that it does not take into account the control flow information of the binary. Therefore, it cannot avoid misinterpreting data embedded in code as instruction while disassembling and producing wrong instructions, which is not only incorrect translation of the data embedded in code, but also wrong for the code immediately following the data. In order to avoid misinterpreting data as instruction, recursive traversal algorithm takes the approach to follow the control flow:

```
Procedure RecursiveDisam(int addr)
{
  while (startAddr <= addr <= endAddr)
  {
    if ( addr has been visited already ) return;
    I = decode instruction at address addr;
    Mark addr as visited;

    If (I is a branch instruction)
    {
      For each target of I do          //Go to target instruction
      RecursiveDisam(addr); //recursively if it is a branch instruction.
    }
    else addr += length();           // Go to next instruction sequentially
                                     // if it is not a branch instruction.
  }
}
```

Figure 2.3, algorithm of recursive traversal

Whenever the disassembler meets a branch instruction, it tries to know the target addresses of both of the two potential branches and continues to disassemble all possible future instructions along both branches. Ideally, if the disassembler knows accurate destination address of each branch, following the control flow, the disassembler can travel the entire code that will execute at run time. The code section will be divided into multiple “chunks” of code, with embedded data out of them. But the ambiguity of target addresses for indirect branch instructions causes trouble, whose target addresses may change dynamically at run time and a disassembler has no way of knowing it statically. Therefore, a disassembler could miss real instructions or cause errors by guessing target address wrongly.

A number of binary translation and optimization systems use this algorithm, such as the UQBT translation system, and researches on Control flow graph extraction.

3.4 State-of-the Art Solutions on Critical Issues

Although the above two algorithms have weakness separately, they have complementary strengths. The linear sweep algorithm does not need to identify the target of indirect jumps for correctness, but it may take embedded data as code and leads to disassembly error. The recursive traversal algorithm can intelligently deal with control flow and thereby disassemble around data embedded in code, but it cannot know the target of the indirect jumps and miss part of code. Based on the two fundamental algorithms, there are a lot of researches in the area of disassembler.

From the viewpoint of high-level language, major sources of the indirect transfer instructions are considered in research. There are three possible cases for indirect “*jmp*” in source code:

1. Switch-statements which is implemented by a look-up table (jump table) and an indirect jump instruction;
2. Exception handling, whose control is transferred indirectly on raising an exception to an exception handler;
3. Goto-statements to pointer-assigned labels;

Two cases for indirect “*call*” in source level are listed in the following:

1. Implicit or explicit function calls
2. Dynamic method invocation in object oriented programming languages.

Jump tables

Jump table is a special case of indirect jump as well as data in code; so many researches focused on it. Basically, jump table is a structure used by compilers to implement C-style *switch* statement [CASE].

<u>source code</u>	<u>code for compiler</u>
Switch (<u>I</u>) {	(1) <u>r := I;</u>
Case 0: ...	(2) if (<u>r >= N</u>) <u>goto default</u>
Case 1: ...	(3) <u>r *= 4</u>
...	(4) <u>r += BaseAddr</u>
Case n-1: ...	(5) <u>jmp *r;</u>
Default: ...	
}	

Figure 2.4 Switch-case in source code and intermediate code in compilers.

A jump table is contiguous N addresses pointing to N corresponding cases in a switch statement. After reading the index number, confirming it is inbound of the jump table and plus the index offset with the base address of jump table, the code will jump to the destination address by an indirect jump. To distinguish jump table in code is thought as a classic problem in disassembler, most of the solution proposed until now is to scan the source code and search the pattern for it.

Cifuentes [CJT01] proposed a solution to make distinguish jump table inside the code. The approach is to use disassembler to search the base address and the bounds of entries. The basic idea is to scan reverse from the indirect jump instruction to find the instruction that adds the scaled index to the base address (instruction 4 in the example),

the checking instruction for jump table sizes (instruction 2), so that the base address and jump table size can be known. After this, the disassembler can continue at each target addresses in the N table entries.

This seems to be a straightforward solution to the problem of jump table. But it depends on one assumption that most of the time the jump table lookup is implemented with a recognizable sequence of instructions: index normalizations, index bound test, loading the target address by base address of table and index and the indirect jump instructions. However, different compilers generate jump tables in different ways, and it is hard to know all of the models that the instructions for the switch statement. Furthermore, the scheduler may mix other instructions in the sequence. The normalization instructions differ with the data type of the operand in the switch-statement. Sometimes, for the scheduling reason, in the sequence of instruction, there is even a duplicate of the index.

B.De Sutter [SICT00] uses a similar approach as Cifuentes to solve the problem of jump table. It applies a pattern match to the program slice that produces the target address of jump to a set of known slices. From the upper-bound test value and normalization of the index to get the number of elements in the address table.

Benjamin Schwarz and others [BS02] proposed an extended linear sweep algorithm to deal with jump tables embedded in the text segment. They observe that the jump table entries' memory locations are relocatable, and the entries' value is pointing to the text segment. Besides, there is an upper bound (K_{max}) in the instruction sets of typical modern architectures for the number of instructions that have above properties and appear adjacent. For example, in the Intel X86 architecture, the number is 2. So for each sequence of N contiguous relocatable text segment addresses, mark the last ($N-K_{max}$) address as data. Then for each sequence of unmarked addresses in the text segment, disassemble by linear sweep algorithm and stop when it meets marked location. Check from the last instruction being disassembled before marked location, delete all incompletely disassembled instructions and discard them. Examine the last correctly disassembled instruction, assuming that there are m ($0 \leq m \leq K_{max}$) address satisfying the above properties, then there must be ($K_{max}-m$) unmarked relocatable text segment address that should be marked as data.

Although this approach is unique, it raises problems when the first K_{max} addresses of a jump table coincidentally good to be disassembled into complete instructions. This algorithm cannot make a difference in this case.

Exception

The control of a program will go to a handler by a computed jump on raising exceptions. Sutter [SICT00] discussed this case. In SPEC95 benchmarks compiled on the Alpha architecture, there is only one target handler at the jump program point. It is easy to find its address, which is in a read-only section of the binary and the indirect jump to exception handler will be solved. But it is not always true for other architecture. For example, in IBM compatible system, there are usually multiple exception handlers for every program. It is hard to find out all possible target addresses of exception handlers. So far, there is no safe algorithm to solve this problem.

Indirect jumps / calls

The general indirect jumps cannot be processed like a jump table, which has comparatively fixed form requirements. The UQBT binary translation system [UQBT]

uses an algorithm as “speculative disassembly”. The idea is to process undisassembled portions of the text segment that appear to code, with the assumption that they might be the targets of indirect function calls. When the speculative disassembly of a particular region of memory encounters error, the disassembly result for the whole region of memory is abandoned.

Benjamin Schwarz [DECR] proposed a hybrid approach to combine the advantages of both linear sweep and recursive traversal algorithm. The idea is to disassemble the program using the linear sweep algorithm, then verify the results of this disassembly one function at a time using the recursive traversal algorithm. At the stage of verification for each instruction i obtained at address A_i , check if the original disassembly using linear sweep has also obtained the same instruction i at address A_i . If not, report *failure*. If no failure is encountered while processing the instructions in the function, report *success*.

De Sutter [SICT00] has a paper on the special topic of static analysis of indirect calls in binaries. Indirect *call* includes two-type, memory indirect calls and register indirect calls. They find some interesting phenomenon for memory indirect calls. For indirect procedure calls in binary compiled for the Alpha, they found that a large number of the indirect calls are actually direct calls. Many function calls are implemented to call a memory location, where the real destination target address is stored.

In the IBM-compatible architecture, it is also true. Especially, when program wants to call library functions, they usually call them in GOT (global offset table). This table includes offsets that need to be relocated at runtime for the library functions and thus allows them to work. This kind of indirect function calls can be solved simply by constant propagation. The reason is that the destination address of the target function is stored in a read-only section of the binary. To resolve the destination address, it only needs to extract it and do the constant propagation back to the code section.

Another interesting things is for the register indirect calls, register live analysis can help a lot to resolve the value of target of the indirect calls. In IBM-compatible architecture, especially for Windows application, it is very common that an indirect call simply moves the destination address to a register and calls the function using the register. It is very frequent that one register value is assigned once and used throughout the function for several times. By using the register liveness analysis, most of the indirect register call can be resolved, but not all of them.

3.5 Disassembler for Security Purpose

3.5.1 Obfuscating Code and Anti-Obfuscation

In security area, in order to protect the privacy of software and prevent binary reverse engineering, lots of researches have been done to make the work of disassembling more difficult, such as obfuscation [CC02, CC97, CL03, TO03], polymorphism and metamorphism [SL04]. All these researches are fundamentally doing the same thing: modify the original code (source code or binary code) such that it is really hard to disassemble and understand, while it still keeps the original functionalities. It is important to protect against software piracy and prevent the intellectual property from theft. It is also helpful to prevent security breaches and prohibit attackers to discover vulnerabilities in an application. But on the other hand, it is convenient for viruses to use the similar technologies and hide themselves inside binary codes.

In the assembly code level, Cullen Linn [CL03] discussed approaches of disrupting, such as inserting junk code, making the control out of order and so on. The technique of polymorphism and metamorphism [SP01] is also applied mostly in the assembly code level, which is also heavily used by the virus and prevent them from being discovered by anti-virus tools.

Since malicious software can use the techniques to protect itself from being detected, security researchers are also interested in how to defeat the obfuscation effectively. Chris Eagle [CE04] uses an emulator embedded in the disassembler named as IDA Pro to work around obfuscation of encryption. He mainly focuses on statically analyzing binary code and use emulator to decrypt the encrypted code. Christopher Kruegel [CK04] investigated on how to disassemble the obfuscated binaries in much more general way. The obfuscation he tried to defeat includes inserting junk instructions, adding half-instruction data, manipulating control flow graph and so on. The main idea of anti-obfuscation is to discover the obfuscated binaries by combining the control flow graph and the improved traditional disassembly strategies. For disassembly strategy, he uses recursive traversal algorithm with several assumptions for the disassembly: conditional branches can be taken or not taken; junk instructions can be after branch instructions, and so on. Intra-function control flow graph is constructed statically. He tried to avoid disassembly incorrectly by carefully confirming intra-function control flow graph nodes and selecting correct ones from all conflicting nodes.

3.5.2 Static Analysis to Find Malicious Software

In the malicious software, attackers put quite a lot of effort to make the software resilient and flexible. The analysis of the malicious software is considered to be very complicated, which combines several stages, creating a controlled environment, collecting the information, code analysis, and behavior analysis. The dynamic analysis will be discussed in the application of debugger in the next section. In the static code analysis, resources that are embedded in the binary are extracted. A program like *binary text scan* [BTS] is usually used to extract the human-readable strings from the binary, which reveal a lot of information about the function of the binary. Complicated resources, like GUI elements, scripts, HTML, graphics, icons and more, can be extracted and recorded by the *resource hacker* [RH].

In academia research, M. Christodorescu [MC03] introduced a general architecture to use static techniques to detect the malicious patterns in executables. It simulated the whole processing: generating virus, and the victim executables, executable loading, program annotating which handles the control flow graph and the malicious code detection, which uses an algorithm based on language containment and unification. It uses external predicates to summarize result of various static analyses. It allows uninterpreted symbols in patterns, which make the method resistant to a common obfuscation technique as renaming.

J. Bergeron [JB99] proposed a static analysis approach to isolate the malicious behaviors in commercial-off-the-shelf (COTS) component. With the popularity of networks, organizations are becoming interested in moving to the COTS component. Inside the COTS, malicious code can become part of the code and affect the integrity of the data and control flow. It decompiles the binary code into high level language first, then applies the slicing techniques to extract the code fragments, and finally, verify

against behavioral specifications of these fragments in order to statically search the malicious code.

3.5.3 Windows Disassembler to Find the Bugs in Source Code from Binaries

Currently the most popular operating system, Windows, is most vulnerable to attacks. Furthermore, Windows itself has a lot of flaws, which makes it very attractive for to attackers. Recently, Microsoft turned to some static technology as “static source code checkers” to help it catch its flaws. A product known as PREFIX made by a company named as Intrinsa can analyze the code created by developers and flags the potential errors. It is said that Microsoft has saved \$60 million in 1999 by using this spot-checking security technique [RL04].

4. Dynamic Binary Analysis Tools

Compared to the static analysis tool such as disassembler, the dynamic analysis tools, such as debugger and simulator have several advantages. First of all, it can be used to probe the context information at run time since most values of register or memory can only be produced and watched on the fly. Second, debugger’s single stepping and breakpoint functionality have made debugging work much simpler. Breakpoint can stop the execution at desired points and single step can continuously go through the execution step by step, which are widely used by programmers and help them to find the error point.

4.1 Debugger

Generally, a debugger must obey several basic principles: intrude the debuggee to the minimum degree, guarantee correct information and provide the context information. First of all, Heisenberg [Grmlich 1983] has proposed that the debugger must intrude the debuggee as least as possible. This principle can be easily violated. For example, when a new process of debugger is introduced, the execution time of the original debuggee process must be changed due to the operating system’s scheduling. It is possible to move the elusive bug to somewhere else or mask it. Secondly, Polle Zellweger [Zellweger 1984] states that the debugger should provide truthful information during debugging. Debugger for optimized compiler is very easy to violate this principle. For example, compiler would like to keep a memory variable’s modifications in register until the last time of modification for the purpose of performance. But debugger may want to probe the variable’s value from the memory address, which unfortunately will return a stale value, while the latest value is kept in temporary register.

4.1.1. Basic Functionalities of Debugger

The basic functionality of a debugger is to set the breakpoint, single step it, get the context snapshot and variable value at any point. Let’s discuss the basic functions of debugger one by one.

The general way for setting a breakpoint is to insert a special instruction into the point, where it is expected to occur an interrupt, and save the original instruction somewhere else. Then at run time, when the execution hits this point, the special instruction causes an interrupt, which will be caught by debugger since in Windows or Linux, debugger has the highest priority to handle the exceptions. Then debugger will get control from this point. The special instruction can be INT3 or any other special instructions for interruption. Breakpoint can have attributes such as temporary or internal.

Temporary breakpoint means that once this breakpoint is hit, it will be removed, while internal breakpoint will be kept until the user wants to remove it.

Single step may set an internal breakpoint, where the single step will start from, and run to the key internal breakpoint with full speed. The speed of the single step running could be 1000 times slower than full-speed running. "Step over" is to keep the debugger in the context of current function, and run all of the descending functions at full speed. A debugger usually puts an internal breakpoint on the return address of the function to allow the function and its descendent functions to run at full speed. "Step into" is to go into the descendent functions if found. It can be supported by the hardware single stepping mechanism or simulated by the debugger itself. Modern computer may eliminate the single step support in hardware for performance. In this case, it is the debugger to decode the instruction, at the same time decide the address of the next instruction, and put an internal breakpoint into the next instruction. For branch instruction, some debugger will decode the destination of the branch and put the breakpoint at the destination instruction while other debuggers will put breakpoints in both branches to guarantee the stop in the next step.

Context can provide the answer to the question of what is the current status and how the program comes here. The first functionality of the context information is to back-trace the stack frame, especially to back-trace the procedure calls. The debugger uses stack-unwinding algorithm to find the frames on the stack. Once the debugger finds out the boundaries of a frame, it can get the return address in the frame based on the procedure call conventions, and map it to the name of the procedure that contains the address. By searching all the names of the procedures that are currently in the stack, the debugger will give a list of the procedure activation records on the call stack, a stack trace. Although the stack unwinding algorithm varies much on the calling conventions for the machine and compiler, basically, each time the debuggee stops, the debugger needs to unwind the stack by following the chain of stack frames back to the initial program entry point. There are several problems in special cases at the time of unwinding. For example, when the debuggee stops at the first instruction at the prologue, it is different from the case that the debuggee stops after the local variables has got space allocated in the stack frame.

The debugger implements an interpreter for the expression queried by the user, evaluates the expression and returns the value to the user. The biggest difference between normal interpreter and debugger's interpreter is the normal interpreter will allocate storage for variables, while the interpreter for debuggers will not allocate storage, but instead access the debugger's child process address space according to the addresses in the symbol table generated by compilers and use the real value which is used by the running program.

As for the symbol table, it is much different from the symbol table used by the compiler itself. Compilers always evaluate an expression in the current scope, while debugger may permit user to point to other stack frame and evaluate its expression. Therefore, the compiler only need to keep one simple symbol table for the current scope, however, the debugger need to keep all scopes all along the time. The symbol table for debugger and compiler need to be kept separately.

Scope resolution in the debugger is critical since the debugger keeps symbol tables for all scopes all the time. In order to let user specify variables that are not in the

current scope, the debugger's evaluator subsystem must extend the variable syntax to attach the information of the scope. Debugger needs to be very careful in the scope resolution in the watch point, the data breakpoint. When a user sets a data breakpoint in a local variable, which is allocated in the stack, the debugger must make sure that the data breakpoint will not be fired when some other scope's variable is using the same stack address. One solution is to disable the breakpoint on local variables when the scope changes. The other solution is to remember the data breakpoint's scope value and confirm the scope information before fire a data breakpoint. Another problem caused by data breakpoint is that the variable may be kept in the register instead of memory for some time, and the data breakpoint may miss the variable's modification. Debuggers can keep a table of lists of memory addresses for the usage addresses of variables, and use the table to track the variable. But the compiler may move the value of variables into register for performance. Without any support of hardware or operating system to raise an exception for the modification of the value in a register, the debugger will fail to fire a watch point in this case.

During the evaluation of an expression, it may need to invoke a function implicitly or explicitly. Although the debugger can interpret most of expressions, it is still not a good idea to parse a whole function, process it semantically and use the interpreter based on the resulting trees. Most debuggers play a trick in the case of invoking functions while evaluating expression. In the child process, the debugger push arguments into stack, sets a breakpoint at the return address of the function and let the child process run from the beginning of the function. When the child process meets the breakpoint, the debugger will get the result value and restore the context of the child process. The return address of the function must be selected very carefully in order to make it work in both recursive and non-recursive calls. The debugger must also be able to handle the following special cases when the function may not return in the return address: there are maybe some user-set breakpoints in the function, or the function may fall into faults, or terminate the child process, or may go out of the function by a non-local goto or "longjmp" that will bypasses the normal return addresses. The solution can be to disable the user-set breakpoints temporarily and take a fault as an error and report to the user and restore the state of the child process. The non-local goto may cause serious error for the debugger since it may not be able to restore the state at all.

4.1.2 Hardware Support for Debuggers

Modern computers provide all kinds of mechanism to support debuggers in the hardware. The set of mechanisms basically include the following, although they are only supported partially in a specific computer:

- Breakpoint – a special trap instruction to halt execution;
- Single-step – a special mode in the processor to execute instructions one by one;
- Data-watch – page protection or special registers in the processors
- Fault-detect – exception / fault detection mechanism

Some computers provide a special trap instruction for breakpoints. When it is executed, the debuggee will be stopped and the operating system will be informed, which further inform the debugger processor that a breakpoint is met. In the computer with variable size of instructions, the trap instruction should have the minimized size of a legal instruction, so that the breakpoint will be in the boundary of any instructions. Intel x86

[PIS] has a one-byte instruction INT3 to be used as the trap instruction for breakpoints. MIPS [GK91] has a special code as BREAK and Alpha has a special trap instruction as BPT. PowerPC [PPCM] doesn't have a special breakpoint instruction, but it has two special registers: HID1 to indicate the state of debugging and HID2 to indicate the breakpoint address. Using registers to record the breakpoint address has advantage as no intrusion to the debuggee's address space. But since the number of the register is so limited, PowerPC needs to use illegal instructions to trap breakpoints when the expected number of breakpoints is larger than one.

For single-step support, computer has a single-step bit to indicate the processor to execute only one instruction and cause a trap to the OS, which in turn informs the debugger that the single step is finished. This single-step bit can be only set by the operating system for debugger. But for the purpose of performance, modern computer may not support this feature, and the debugger will simulate the single step by setting a breakpoint on the next instruction. Intel x86 has a Trap flag bit in the processor to be a single step mode switch. PowerPC has a bit in the Machine State Register (MSR) to set for single step. There is no instruction-level single-step support in the MIPS and Alpha.

Data breakpoint is used to inform the debugger that specific addresses in the debuggee are modified. Some computers provide special data breakpoint registers to indicate the base address and the length of the range to detect. Within this range, if any address is modified, the processor will notify the debugger. Another mechanism is to mark the whole data page as read-only, so that once a write is executed, a page access violation will occur and the debugger will check whether it falls into the checking data range for data breakpoints. Intel x86 has four registers to record the addresses of data watch and support the data breakpoint mechanism. PowerPC has Data Address Breakpoint Registers from HID3 to HID5. MIPS and Alpha has no such special data breakpoint register support. In the case of no or no enough data breakpoint register, in order to implement the data watch, the page fault mechanism is used.

Fault detection support is very important to debuggers, because the responsibility of debugger is to handle all kinds of faults most of time. Generally, some fault is processed by the processor, such as the *divide by zero* and *memory access violation*, and some fault is handled by OS, such as the I/O failure. In the case of the debugging, the debugger always has the highest priority to know the information of faults before the debuggee is allowed to control again. The debugger needs to check whether it is a trap occurring for debugging purpose, or whether it is an ignorable failure, such as the time-out alert, which may be part of the correct running of the debuggee, or whether it is a real fault caused by the debuggee, for which it will inform the debuggee the context state to help it figure out the reason.

4.1.3 Difficult Points of the Debugger

Most of the bugs can be detected simply by setting breakpoints, single-stepping the execution, and extracting the context information. Unfortunately, not all bugs can be easily found due to the following several reasons.

First, some bugs are silent, which will not cause any exception at the time of running or debugging. For example, when assigning value to an array out of the bound, it will cover some other variable's name whose address is immediately after the array. Later, when the variable is used, it will cause the computing result wrong silently without any noticeable exception. Some other examples are like the value without initialization.

For this type of bugs, it depends on the programmer to set the breakpoint or single step at the correct place and debugger cannot provide other special help.

Secondly, some bugs will cause exception, but only a long time after the error is made or far way from the instruction where the error really occurs. One example is when a pointer is misused, some other memory address is modified unexpectedly. However, the modified address causes noticeable exception, such as invalid memory access, only when it is really being used. That could be long time later and far distance away.

Good supports from the debugger for this type of bugs are data breakpoint, which is to monitor the access and modification on specific memory addresses, and conditional breakpoints, which means that the breakpoint fires and only fires with some special condition. For the data breakpoint, although several hardware architectures provide special supports, such as the data breakpoint registers, or the page fault for the whole page of addresses, the number of the data breakpoints supported by registers are very limited. Paul E. Roberts [PER96] has an excellent thesis describing several software approaches, such as trap patching, virtual memory and kernel-supported checking to improve the performance.

Thirdly, some bugs are not able to be reproduced or will not happen when the debugger begins working together with debuggee. Real time related bugs are belonging to this category. The error will not reappear when the debugger is used to monitor the execution, because the debugger changes the execution time and the speed of execution of the debuggee.

4.1.4. Debugger for Security Purpose

In security domain, the strong power of the debugger to view and modify the running process' context state is easily to be used by malicious attackers. It is true that the attackers can only trace the assembly language, but they understand what is happening in the application code very well, especially the critical points, such as the entry point of a program, imported functions from other DLLs and so on. The famous popular tools in attackers includes NuMega's SoftIce [Sfl], Data Rescue's IDAPro [IDAP], Windows advanced debugger [WD] and GNU's gdb [GDB].

Most of the binary codes for software strip off the debugging symbols and other related information before they are distributed to customers. It is mainly because the size of the executable file can decrease dramatically by stripping the debugging symbols while it doesn't affect the real functionality of the software, since the debugging symbols are only used for the purpose of the debugging. It is also critical for security consideration, since it makes debugger unable to get the higher level information support [SMI00], such as addresses of functions, global variables and so on.

However, it doesn't stop attackers cracking software. Even without the aid of the debugging symbols, attackers still can utilize the debugger, to probe and extract the embedded secret of software. Compression and Encoding is common approaches to protect the software from being cracked. One advantage of compressing and encoding is that the program must be decompressing or decoding before it can be modified. This increases the difficulty of using debugger, since debugger has the limited viewpoint of the instruction trace currently running, which is only part of the whole program. However, if the attacker has enough patience, it still can trace all instructions one by one to the point where the code is decrypted or decompressed. Therefore, anti-debugger becomes a technique that must be applied in the software protection tools. It is used to

stop the monitor of debuggers by detecting whether the program is running under the control of a debugger and giving up executing once it is true. The famous software protection tools include ASProtect [ASPT], tElock, BurnEye, UPX [UPX], InstallShield[INSH] and so on.

Although debugging may be threaten to the privacy of the software and it could be used to search the flaw or backdoor in the software with malicious intention, debugger is also a powerful tool to protect system for the purpose of security. Good security guys can use debuggers to search the flaws in the existing software products and apply corresponding solutions for those flawed software. Debuggers can also be used to detect the virus, trace or monitor the execution flow of software, and prevent virus or Torjans to continuously infect others or make damage to the system.

The key idea of using debugger to monitor the execution of applications is to insert breakpoint into security critical points statically or dynamically, stop the execution of the applications at run time, turn to execute the monitor program and afterwards resume the execution of the original application if it is passed the examination of security monitoring. Breakpoint is an essential issue in this case, because how to implement breakpoints directly affects the performance and effectiveness of the security system.

Different Process Debugger

The most natural approach for a security guy to use a debugger to monitor the execution is to start another process of debugger, and collect information of the original debuggee. The breakpoint is usually implemented with a trap to the operating system, which will schedule the debugger to run in turn. Most of the modern interactive debuggers are like so, such as dbx [SUN], gdb [GDB], and windows advanced debuggers [WIND].

There are several advantages for the approach of different processes. Since the debugger and the debuggee are in different processes, that is, in the separate address spaces, the debugger will not messed up the debuggee's memory, or compete for the debuggee's resources. On the other side, a bug of the debuggee or the virus in the debuggee to screw up a random memory will not affect or disable the debugger's monitoring. Furthermore, a debugger in another process can support the kernel debugging, and teledbugging. In the case that security guys only concerns with the power of finding flaws or vulnerabilities in the software, this approach helps a lot to dig into the debuggee in depth.

Cristina further proposes to debug the application in the high level language [CC01], which means to use the decompilation technology at the run time in the debugger, reconstruct high-level language such as C to help the user understand the binary code better. It obviously increases the performance overhead of the debugging further, but it provides a more friendly user interface for debugging.

Same-Process Debuggers

When users want an application running with normal speed although being under security monitoring, the approach of separate process debugger is not a good choice, since it decreases execution speed of the application quite a lot. With the consideration of performance, the debugger in the same process as the debuggee is much better than that in different process.

First, there is no time cost for context switching among the debugger and debuggee. Instead, the state exchange between debugger and debuggee is performed

simply by several instructions. It dramatically improves performance. Secondly, debugger can access the debuggee's memory and registers directly without any operating system's support, since they are in the same address space. It makes much easier of the work of debugger to set breakpoints, examine and modify the debuggee's state and erase the breakpoints when they are useless.

Existing famous same-process debuggers include the cdb debugger [MF83] for the DEC Rainbow 100, MD-86 [MG87] for the Intel 8086 and VAX Debug [BB83] with supports of multilingual language, source-level debugging and watch points. In the same process debugger, the breakpoint can be implemented with either a trap instruction or a branch instruction. VAX Debug is implemented with a set of exception handlers. At the time of the exception occurring, the operating system walks through the call stack to look for the appropriate exception handler. Once it is found, the execution control will be given to the exception handler. If no appropriate exception handler is found, the debugger registers additional exception handlers.

Detours [GH99], a binary interceptor made by Microsoft, implements and discusses different same-process breakpoint techniques. It intercepts Win32 functions by re-writing target function images and patching the instructions inside the function, edits the import table of the binary files and attaches arbitrary data segments to binary files. It compares the performance for different implementation approaches for the break points, such as the direct jump instruction, call replacement, DLL redirection, and breakpoint trap instructions.

Peter B.Kessler's paper [PB90] has a description on the implementation of breakpoints in the same process. The basic technique to plant same-process breakpoints is writing the code used as breakpoint code, and then patching the instruction at the breakpoint addresses with a branch instruction to jump to the new breakpoint code. But he used the SPARC architecture, which has a fixed length of the instruction. When the same-process debugger is implemented on the instruction with variable length, such as 80x86, it will be a much different story.

4.2 Simulator, emulator and instrumental tools

Simulator and Emulator

To investigate the binary code, besides to disassemble the binary code statically and to monitor the execution flow by debugger dynamically, there exists another related but quite different approach: to simulate, or emulate the execution of application. A code simulation tool is a software program that simulates the hardware execution of the test program by fetching, decoding, and emulating the operation of each instruction. An emulator tool also attempts to copy the hardware of a system, but it mainly aims to run on a specific platform and allow users to run software written for a different platform.

A complete simulator and emulator both need to simulate all actions of hardware in a computer, including the processor, registers, memories, IO, buses, micro controllers, and interrupts. The work to build a simulator or emulator is very complex and time consuming.

SimOS [SimOS98, SimOS97] is a MIPS-based multiprocessors machine simulator and models the hardware components of the target machine. It contains software simulation of all the hardware components of modern computer systems:

processors, memory management units, caches, memory systems, and I/O devices such as SCSI disks, Ethernets, hardware clocks and consoles.

SPIM [JH93] is a simulator introduced in a famous textbook written by J.Hennessy and D.Patterson. Shade [BC94] is an instruction-set simulator and custom trace generator. Application programs are executed and traced under the control of a user-supplied trace analyzer, which is running in the same address space as Shade. Other famous multiprocessor architecture simulators include Mint [JE93], Tango [HD91] and Proteus [EB91].

Instrumental Approach

Instrumental tools for system, is another unique and effective approach, which usually collect a trace of the behavior that can be processed to extract useful information of applications. An instrumental tool is capable of modifying a program under study so that essential dynamic information of interest is recorded while the program executes.

An instrumental tool is working by rewriting the program that is the target of the study so that the desired run-time information is collected during its execution. The program is still executed by the native hardware of the original application, the logical of the target program is the same as it was without instrumentation, but data collection routines or other routines with special concerning on the executing states are invoked at the points in the target program's execution to record interesting information.

Instrumental tools can be used in the executable binary code, linking objects, or compiling source code. Most of the instrumental tools involve executable code only, and do not have the requirement of source code. In general, code instrumentation includes the following key steps: First, extract code, by disassembling or analyzing the structure; Second, choose the inserting points in the program, insert instrumentation code; Third, update the original code to reflect new code addition, such as update the relocation information or control instruction target translation; and finally, construct the new executable if necessary. Many security tools for security applications of disassembler and debugger are belonging to the category of instrumental tools.

Historically, there are many strong and famous instrumental tools. ATOM [AS94] is a tool on Digital's Alpha workstation to customize instrumentation and analysis by allowing user to writing small instrumental routines and decides the inserting points. For example, the ATOM provides library routines to allow users to access to each procedure in an application, each basic block in that procedure, and each instruction in that basic block. IDtrace [JP94] is an instrumental tool for Intel architecture Unix platforms, but it is compiler dependent in order to recognize jump table code for disassembly purposes. Pixie [FC86] is the first widely used binary instrumentation tool. It runs on MIPS R2000, R3000, and R4000, and is included in the performance/debugging software package of most systems on MIPS architecture. QPT [JL94] instruments binary code and generate compact traces for later simulation. It performs control flow analysis, with the information of symbol table and code structure knowledge. The address of each function is found in symbol table and a control flow graph is constructed with a basic block at each node. It decides the likeliest execution path by heuristics, locates the optimal code insertion points on CFG edges and add the addition instruction to original code. There exist a lot of other instrumental tools, such as Goblin [CS91] for IBM RS/6000 applications, SpixTools [BC93] for SPARC application binaries, Spike [MG91] based on

a compiler (GNU CC) and so on. For the Win32/X86 binaries, Etch [TR97] is a general-purpose instrumental tool for binary rewriting. It provides a framework for x86 binaries to modify executable files for both modification and optimization. PIN [VI04] is a binary instrumentation tool running on the Itanium system for the purpose of computer architecture education.

Simulator vs. Instrumental

Simulator has many advantages, such as complete event coverage and non-intrusiveness. In contrast, instrumentation tools may miss some important events, such as bus information, which can only be collected by the hardware monitors. Further, intrusiveness is a problem for instrumental tools, since it requires patching or rewriting the existing application. The application manipulated by an instrumental tool will have a longer running trace than original form. Therefore, the events generated may not match what really occurs in the execution of original application. Hardware simulation is less intrusive and do not need to insert new instructions to original code, but it is slow in running and hard to implement.

Compared to simulator, the advantage of instrumental tool is with wide applicability, being much more fast and easy to be implementation. They are popular because they are applicable to many architectures and programs. They have relatively low overhead, because the instrumental code are added into the code directly and the code is running on the native hardware. They are much more easily to be implemented, since it doesn't need to know about the underling hardware structure, or operating system. The instrumental tools on binary only need to take care of the binary format and appropriate modification due to the insertion of new code.

5. Reverse Engineering and Tamper Resistant Software

Software attack and defense is an endless game against each other. With the aid of reverse engine tools, such as disassembler, debugger, and memory dumper, attackers might trace programs and reveal the internal secret or even the whole source code from the binary code, which will lead great military panic for armies and enormous economic lose for commercial companies. Given enough time, effort, perseverance, it is believed that attackers might trace and reveal any program. However, it is also believed that a little more effective protection strategies would make the work of cracking much more daunting. The objective of tamper resistant software is to increase the complexity of attacking to such an extent that the prohibitively high cost makes the attacking worthless.

5.1. Reverse Engineering

Reverse engineering is to take the binary code or the low level instructions executed in the processor and backs out information of the high level language. It is widely used by attackers to crack the software.

Tools for reverse engineering include disassembler, debugger, memory dumper and so on. For disassembler, win32dism is a widely used one, which can provide assembly code and the binary header's information, such as import functions, export functions, symbol table and so on. For debugger, Ollydbg, SoftIce, and IDAPro are popular power ones. SoftIce is a kernel-mode debugger, with insight into the Windows environment. With SoftIce, attackers can track calls and events down to the lowest level

of the operating system. Traditional debugging tools run at the Windows application level, and can't observe and report actions and events occurring at the kernel level. SoftICE is specifically designed to run between the operating system and the hardware, and can see interactions between drivers, calling routines and kernel services. It is now a commercialized software product and support kernel and driver debugging. IDAPro, Interactive Disassembler Professional, is recently the most popular one in the area of reverse engineering. One distinguishing characteristic of IDAPro is that it supports plug-in. Chris Eagle[CE04] develops an ida-x86 emulator plug-in, which can emulate execution of x86 instruction set without really running the application. With the emulator, it defeats the static reverse engineering technique, such as encryption, by allowing the program "run" through the decryption routine within IDAPro.

Reverse engineering [CB03] starts with collecting the information of the application. It needs to investigate several points first by trying disassembler or debugger. Is the application protected by anti-disassembler or anti-debugger mechanism? Is the file packed or encrypted? Does it need any other system related file to work? In order to crack the registration number, the normal standard steps are like following. Put a breakpoint in hmemcpy, run the program and enter any number. Now, the cracker breaks in the execution of application. Then, the cracker looks around the assembly code, and eventually gets to the place where the check on his number is performed. Next single step inside the code and watch what happens. Finally, the cracker analyzes the code, understands what it does and chooses either to disable the registration number checking or to discover the legal registration number.

5.2. Tamper Resistant Software

In order to protect software from being cracked, research on tamper resistant software becomes prevalent.

One passive approach is to insert copyright marking into the binary, including fingerprinting and watermarking[CC00,CC99]. But it is in limited success. One reason is that the watermarking can be discovered, removed or modified from the binary. The more important reason is that it is too passive, and cannot stop the attacker cracking the software at all.

More applicable approach is to insert automated unpacking/decrypting of "protected" binaries, such as UPX, burneye, shiva, tElock, ASPack, Sphinx and so on. The aims of these tools are to increase the complexity of attacking to such an extensive degree that the prohibitively high cost makes attacking meaningless. The most fundamental skill is to encrypt or pack the code and embed an automated decryption or unpacking routine inside the code. The encrypted or packed code makes disassembling meaningless and improves the difficulty of investigation of debugger. Most of them apply anti-disassembler and anti-debugger tricks to try to discover crackers. At run time, the "protection" code embedded in the code first gets executed. It actively checks whether the execution environment is safe or not. If there is debugger monitoring the execution, the "protection" code will give up execution silently. Only in the case that the program is running alone without monitoring of debugger, the original program will be decrypted and start to execute. The idea is very cool but until now there is no good way to guarantee detect the debugger in a new system environment. Another problem is how to hide the key of encryption inside the binary. Since the key is needed to decrypt the encrypted

code, it is all of the protection tools such as UPX, ASPack, shiva are once being cracked and they distribute newer version continuously based on old failure experience.

Obviously, the defense of cracking is much more difficult than cracking, since cracking only needs to understand the code while the defense needs to understand code and insert instrumentation code to prevent from being understood. On the other hand, the defense of cracking in tamper resistant software is very similar with the techniques inside virus, which tries to escape the detection of anti-virus tools. But tampering human attacking is much more difficult than virus to defend the detection of virus-scanner. The prior one is to defend human's intentionally analysis while the latter one is anti-automated scanner.

Another popular technique in the tamper resistant software is obfuscation. Here, obfuscation means to change the appearance of the code or data, but keep the functionality the same as original application. Cloakware corporation [JR97] applies obfuscation on control flow encoding and data flow encoding. They deploy the applications to another forms, while they still perform the functions they were designed to perform and conceal proprietary algorithm and secrets in the software. But it requires source code to do the modification before compilation. Cullen [CLS03, CL03] performs the obfuscation in the level of assembly code. He uses a perfect hashing function to patching all destination addresses of direct jump or call to indirect ones. By converting direct branch instruction to indirect branch instruction, it increases the difficulty of static reverse engineering. But using debuggers to trace the execution can defeat it. Even in static time, emulating can defeat the obfuscation of branches' target addresses. Metamorphism and polymorphism are used in tamper resistant software to make obfuscation in assembly code level. They do register exchanging, instruction substitution, procedure replacement, code shrinking and expanding on either decryption routine or the original code itself. They effectively prevent the cracker to distribute automated cracking tools through Internet. Since with metamorphism and polymorphism each version of code is different and need to apply different cracking tools, there is no use to broadcast the cracking tools. Problem of them are how to evaluate their efficiency of obscuring of human's mind. It can modify the code's instructions dramatically, but the underline logic may still exist.

6. Disassembler-based Binary Interpreter

The disassembler-based binary interpreter is based on a combination of the disassemblers and the debuggers. The target of the binary interpreter is to fully use the disassembler's advantages, and combine with the dynamic advantages of the debugger to compliment the inaccuracy of disassemblers. In contrast to the normal debugger, that only uses the disassembler to provide the run time instruction information, our disassembler-based binary interpreter has the advantage of using the disassembler both statically and dynamically. The disassembler is used to provide the maximum accurate code address information, collect the unclear address area information, leave the minimum ambiguous address area for debugger to make clear at run time.

Different from all of the other current binary tools, the tool aims to provide a simple and fast solution with good portability across different hardware platforms. Instead of a complex emulator, we take the approach of using debugger to interpret

instructions dynamically base on the disassembly result. Since we do not rely on an emulator or simulator, the tool can be easily ported to different hardware platforms, which makes it extremely convenient to use.

Speed is a critical issue for the performance of a binary interpreter, which is usually slow for interpreting all instructions one by one. The goal is to make our tool fast by static disassembling first and inspecting each instruction once and only once at run time. This improves performance dramatically since most of the time the application can run in native speed. Only when the program encounters indirect branch instructions or enters unknown area, it invokes debugger. Furthermore the debugger obeys the rule that it only interprets each instruction once, which improves performance especially on iterations of a loop.

The tool also has very good transparency. Since it is a binary tool, we do not require recompilation of the source code. The tool will do special processing on the binary of an application directly, which avoids many problems. For example, when applications are modified in source code, we do not need to recompile by our special tool. The only thing we need is the executable file for the new version. Besides, we do not need special hardware or OS support, which makes it easier to port.

The application area of the binary interpreter is broad. It can be a runtime code analyzer or a runtime code modifier, because our interpreter can observe any instruction before it is executed at run time. Besides, with the aid of implementation of application program interface, the binary tool can modify original code freely. The ultimate target for application is to build a security level in a system. Based on the analysis information of the fast binary interpreter, we are able to embed security instrumentation codes into the original programs to protect it. Combining the binary interpreter with the technology of security, it is easy to build a special security level in a system. All the executable files running in the computer will be observed and protected by our security instrumentation level before it really accesses the underlying hardware level. Therefore, the security level can stop the malicious code before it really gets executed. It is a very simple and promising approach to protect the whole system.

Disassembler Part of the Binary Interpreter

The disassembler starts to disassemble from the entry point of a program, which is definitely a correct code address. As a common recursive traversal algorithm, the disassembler continues to disassemble the instruction in sequence until it meets a branch instruction. If it is an indirect branch, the disassembler does not try to guess the target address of indirect branch instruction. Instead, the disassembler marks it for the debugger to take care at run time. If it is a direct branch instruction, the disassembler will recursively start new instruction sequences by following all possible branch instruction traces, which can be obtained accurately at static time. The instruction traces ultimately divide the code section into multiple blocks. We call the blocks of sequential instruction trace that have been disassembled into code by recursive traversal algorithm as “known_block”s and the leftover chunks as “unknown_block”s.

Here, there is a simple principle. When a disassembler begins from an address, with completely certainty that it is a legal instruction start address, it can disassemble the following instructions completely correctly till it meets the first branch, which can possibly be followed by data. That is, for a known block of instruction, if we are sure two

things: 1. the beginning address is an instruction; 2, there is no branch instruction in the middle of the block, we are sure that the whole known block contains only continuous code, without data. Using the above recursive traversal algorithm, the disassembler will only take direct branch instruction's possible targets as the beginning of the next known block. Obviously, whether the control flow really goes through direct branch instruction's possible target addresses is the key for whether our disassembler has made a correct choice of the beginning address of the next known block. It raises three further questions:

Question 1: What should be the beginning address of the next known block, assuming that the start address of a known block has been correctly identified?

The target is to make sure that the beginning address of a known block is really starting address of a valid instruction. Obviously, unconditional jump instruction's target address and call instruction's target address must be an instruction in the control flow. Conditional jump instruction's target address has some risk of dangling, in the case of false/true-forever.

Question 2: What should be taken as the end of a known_block?

This question is equal to which branch instruction's next instruction is possibly data, so that disassembler should terminate the known block. There are several different kinds of branch instructions: "ret", unconditional jump, conditional jump and call. Obviously, the instructions after "ret" and unconditional "jmp" are not belonging to the control flow. It is also probable that alignment data are filled after them. Therefore, it is easy to tell that "ret" and "jmp" will be the end of a known block. As for a conditional jump, there are two possible branches, the target and the next instruction. But in the case of True-forever or False-forever, either of the branches can never be taken. As for call instruction, although usually return address of the callee function is the next instruction of the caller, some special cases, such as exception handler or a system call, will cause the control flow to divert from the original return address. Experiments in the next section will show that a strict requirement of making all of the branch instructions to be the end of the known block will lead the number of covered code instruction eventually very small. So we make several tests and try to find a tradeoff between accuracy of control flow information and instruction number covered by known blocks.

Question 3: With the possibility of entering not-taken branch in the middle of a block, how to increase the liability of the known block?

We introduce a confidence system. First, using the aggressive recursive traversal algorithm, we scan through the whole code section. Here, the aggressive recursive traversal means that it takes all possible target branches as the beginning of the next known instruction block, including the addresses after "ret" and "jmp" instruction. The reason for using aggressive recursive algorithm is to cover instructions as much as possible and cut the code section into different instruction blocks with the requirement of the end of known blocks. The block includes all the instructions in continuous addresses that are expected to stay in the same control flow from the start point of block to the end point of the block. At the same time, disassembler records all direct branch instruction addresses. In the second stage, we check the confidence degree of each address, using these branch instruction addresses.

The main idea of confidence degree checking is to get the instruction addresses information through reference and propagation. Confidence degree of an address is a metric to show the confidence level that this address is instruction code, not data. The implementation is first to set all addresses' confidence degrees as zero, subsequently increase the confidence degree of the entry point of the program, since it is definitely an instruction address. Then apply the following three rules to all other addresses.

1. If a branch instruction (call or jump) has a high confidence degree to be code, the confidence degree of its target reference address should be increased. We call it a confidence forward propagation from call sites to reference sites.
2. On the contrary, if a target reference address of a branch instruction has high confidence degree, the confidence degree of the branch instruction should be increased. This is the case of a confidence back propagation from reference sites to call sites.
3. If one instruction's confidence degree is increased, all subsequent instructions until the end of the known block's confidence degree should be increased. This is called a confidence down propagation from one instruction to its following instructions.

The current implementation is to increase 1 each time when an instruction gets confidence from other instructions. Figure 6.1 shows an example, which contains the three types of propagations. Both of instruction 0x40108e and 0x4010b6 has a reference to instruction 0x40b034. So by forward propagation, 0x40b034 increase confidence by 2 while by backward propagation, the two callers increase confidence by 1 separately. All of the instructions in the same block also increase corresponding number of confidence.

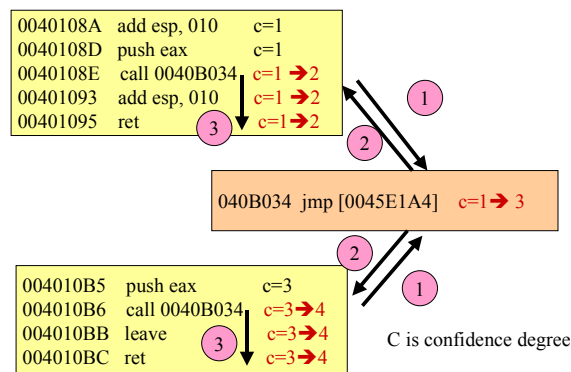


Figure 6.1. An example of confidence system.

Dynamic Part of the Disassembler-based Binary Interpreter

Currently, we use a debugger to be the dynamic part of the binary interpreter. It provides compliment of dynamic information for the uncertain unknown part of the static disassembly result. Using the disassembler, we already know statically, the part of the binary code that has instructions, the part that is unknown and may be either instruction or data, and the instructions in known area that are indirect branch instructions that may lead the execution flow to transfer from known area to unknown area. Therefore, with the dynamic debugger, we set traps to all the indirect branch instructions in known area,

which are the only points to let the execution flow escape to the unknown area. Once the trap is raised when it is executed at run time, the debugger gets the control and single steps or uses the run-time disassembler to collect the instruction information of known area and unknown area.

Besides using the trap instructions, patching the indirect instructions with a jump instruction to our newly inserted checking routine code is also implemented in our binary interpreter. This provides seamless transferring from the original code to the analysis code. But since patching need to rewrite binary, we allocate a new block of memory in the binary to hold new patching routine, the memory for code section must be writeable. Besides, patching indirect instructions with a jump instruction leads much more problems, which we must resolve carefully:

- There must be enough space to plant the branch instruction in the breakpoint address.

The branch instruction might be larger than the original instruction at the breakpoint addresses. In this case, it needs to patch some more instructions following the breakpoint addresses. The original instructions need to be saved in some other places. It must be very careful. First, the moved instruction may need to patch for relocation. For example, if the instruction is a relative jump instruction, it needs to patch the distance between the new address and the original address to the destination offset. Second, there must not be any other instructions that will jump to the middle of these patched instructions; otherwise the execution flow will not find the correct instructions.

- The distance between the breakpoint address and the destination breakpoint code must be within the range for a legal branch instruction.

There are limitations on the branch distances for branch instructions in each instruction set, such as Pentium [PIS]. For example, in the Pentium instruction set, the relative jump instruction can only transfer to the addresses within around 128 bytes forward or backward. The distance between breakpoint address and the destination breakpoint code must fall into the range.

- The new breakpoint code must save and restore state, such as registers.

The debugger shouldn't violate the execution of the original application, therefore, the debugger needs to save the state before executing its breakpoint code and restore the state before it returns back to the original application.

- The original instruction replaced by branch instruction must execute after the breakpoint.

This is for the integrity of the execution of the original application. They can be executed in the original place or be executed in some other places, such as heap or stack.

- It must handle modern architecture optimization characteristics, such as delayed branch.

Some modern architecture supports the optimization from hardware, such as delayed branch, which means to execute the instruction after the branch instruction before the branch instruction. The simplest solution is to disable this characteristic.

- It must clear the breakpoint code when it is useless.

The breakpoint code need to be clear after it is useless for the purpose of saving memory. In a single processing environment, it is easy. In the multiple processing environment, it must make sure that all of the process are not using the breakpoint code any more.

Sometimes, the breakpoint address doesn't satisfy the above mentioned requirements. For example, a 2 byte indirect register jump instruction "jecxz" needs to find 4 more bytes following it, so that a branch instruction can be patched to it and transfer the execution flow to the checking routine. But the 4 more bytes must belong to instructions that are not destination of other jump instructions. In this case the requirement cannot be satisfied, we still need to use the shortest trap instruction, such as "INT3" to insert a break point there. In order to decrease the time of context switching, we move the dynamic monitor from separate process debugger to same process debugger, such as exception handler. When the trap instruction is executed, the same process exception handler will take control and get more information of known address of instructions from unknown area. This design achieves extremely fast "context-switching" between the analysis tools and the original application.

Performance evaluation

We tested the confidence system with different known block end mechanism in windows XP, Intel Pentium (M), 1.5GHz. Around 10 applications are tested in 6 different known block end mechanisms. All of the applications are console applications for the convenience of testing performance impact of the debugger on the applications at run time.

All of the 6 different end block mechanisms use the above confidence system, checking confidence degree through branch reference recursively and set the threshold of confidence degree as 2.

One group, named as "Require_end_next_true", is using aggressive recursive traversal algorithm, which starts the new block with all possible next instruction addresses and even the next instruction address after ret and jmp. The other group, named as "Require_end", is using pure recursive traversal. "Require" has 3 types of ending requirement of blocks. "rj_cj_call_end" takes ret, jmp, conditional jmp, and call as the end of a block. "rj_cj_end" takes ret, jmp, conditional jmp as the end of a block, while "rj_end" takes ret and jmp only as the end.

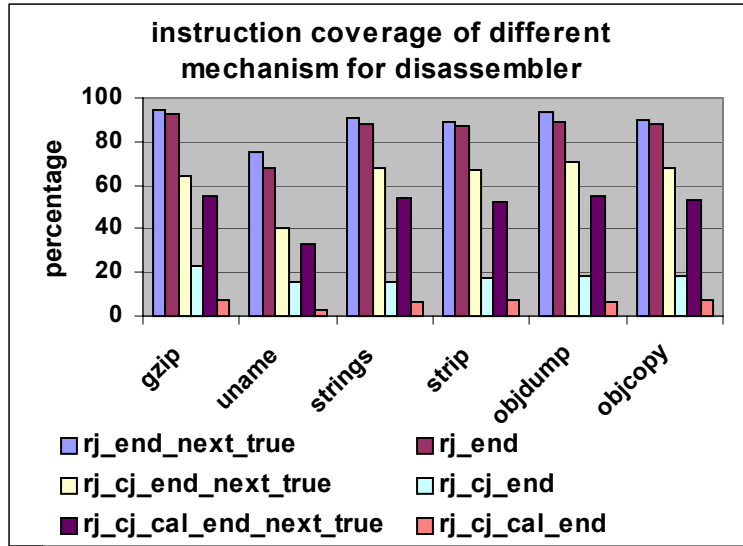


Figure 2. Instruction coverage of different mechanisms for end of block in disassembly.

The applications include gzip, uname, strings, strip, objdump, and objcopy. The most aggressive one is around 90% coverage of code, while most conservative one is in very small number, around 5%. Experiment shows that with less restrict rules for the end of blocks, there is no much difference between aggressive recursive traversal and pure recursive traversal. Both of them have around 90% addresses that are confirmed to be instruction. But for the restrict end of block rule, the aggressive recursive traversal obviously covers around 30 times of pure recursive traversal algorithm. The reason is when the more types of branch instruction become end of block, the smaller of each instruction block is. Therefore, it has much less chance to have call or jmps inside the block to continue the control flow. One purpose of our disassembler is to provide as much as possible code address information, while another purpose is to provide accurate code address information. The disassembler chooses rj_end mode of ending known block mechanism with a confidence checking.

	gzip	strings	objcopy	objdump	Uname	strip
Indirect_patch_single_step_unknown	2.30%	1.20%	196%	10%	200%	60%
Static_initial_runtime_disasm	1.20%	0.20%	68%	2.20%	35%	20%

Table 6.1. Performance evaluation for the binary interpreter on several applications.

There are two versions of performance evaluation shown in table 6.1. One is to patch the indirect branch instructions with INT3 and then single step through the first encountered unknown area instructions using the debugger. The other is optimized one, with the unknown area and indirect branch instruction information initialized and embedded into the binary statically. Furthermore, at run time, when the execution goes into unknown area, instead of single-step all instructions, the debugger invokes a run-time disassembler to investigate the instructions. With the optimization, the performance becomes 6 times faster than original version on average. This is because static initialization saves the time spent on dynamic initialization, and runtime disassembler

further decreases the number of context switches between the debugger and debuggee. With the optimized version of binary interpreter, the overhead of interpreter is around 20%. Only in objcopy, the overhead is more than 50%. The reason is that the objcopy has over 1000 indirect branch instructions in it, which is a large number and costs a lot of time to do the checking. Application uname has 35% overhead because the uname is a small program with only thousands of instruction overall and within 0.01 second by native execution. Therefore, the overhead of interpreter monitoring becomes obvious comparing to the native execution time. Experimental results indicate that the binary interpreter is quite effective, because the overhead caused by the interpreter is usually small.

7. Conclusion and Possible Future Direction

Binary analysis becomes very important and useful due to the increasingly demanding requirement of security system to detect malicious code or bugs in codes in the existing applications or network traffic. This paper provides a survey from the most fundamental aspect of binary analysis, starting from the introduction of typical binary code formats, followed by the discussion of several basic analysis tools, such as disassembler, debugger, simulator and emulator. For each of them, the advantages and disadvantages are compared, the applications in the security area are illustrated and the state-of-the-art techniques are introduced. Furthermore, we proposed a new disassembler-based binary interpreter, which is a combination of the advantage of disassembler and dynamic analysis tools.

Reference:

- [AS94] A. Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools," Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1994.
- [ASPK] Aspack Software, <http://www.aspack.com/asprotect.html>.
- [BB01] B.Barak, O.Goldreich, R.Impagliazzo, S.Rudich, A.Sahai, S.Vadhan, and K.Yang, "On the (Im)possibility of Software Obfuscation", In Crypto, 2001.
- [BB83] Bert Beander, "", in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, pages 173-179, August, 1983.
- [BC93] B. Cmelik, "SpixTools Introduction and User's Manual", Technical Report SMLI TR-93-6, Sun Microsystems Laboratory, Mountain View, CA, Feb. 1993.
- [BC94] B. Cmelik and D.Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", Proceedings of 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 128-137, May 1994.
- [BTS] Binary Text Scan, <http://netninja.com/files/bintxtscan.zip>
- [CASE] R.L.Bernstein, Producing Good Code for the Case Statement, Software—Practice and Experience, 15(10):1021-1024, October 1985.

- [CB03] The Codebreakers Magazine, issue #1, 2003, <http://codebreakers.anticrack.de>
- [CC99] C. Collberg, C.Thomborson, “Software watermarking: Models and dynamic embeddings”, in Proc. 26th ACM Symposium on Principles of Programming Languages(POPL 1999), pages 311-324, January 1999.
- [CC02] C.Collerg, C. Thomborson, “Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection”, IEEE Transactions on Software Engineering, 28(8):735-746, August 2002.
- [CC97] C.Collberg, C.Thomborson, and D.Low, “”, Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
- [CE04] Chris Eagle, “Attacking Obfuscated Code with IDA Pro”, Black Hat, USA 2004, Las Vegas, July 2004.
- [CI01] C Cifuentes, T Waddington and M Van Emmerik, “Computer Security Analysis through Decompilation and High-Level Debugging”, Proceedings of the Working Conference on Reverse Engineering, Workshop on Decompilation Techniques, Stuttgart, Germany, October 3, 2001, IEEE Press, pp 375-380.
- [CI93] C Cifuentes, KJ Gough, “A Methodology for Decompilation”, Proceedings of the XIX Conferencia Latinoamericana de Informatica, pp 257-266, Buenos Aires, Aug 1993.
- [CI94] C Cifuentes, Reverse Compilation Techniques, PhD thesis, Faculty of Information Technology, Queensland University of Technology, July 1994.
- [CI95] C Cifuentes and KJ Gough, “Decompilation of Binary Programs”, Software – Practice & Experience, Vol 25(7), July 1995, 811-829.
- [CID96] C Cifuentes, “Interprocedural Data Flow Decompilation”, Journal of Programming Languages, Vol 4, 1996, 77-99.
- [CIE95] C Cifuentes, “An Environment for the Reverse Engineering of Executable Programs”, Proceedings of the Asia-Pacific Software Engineering Conference(APSEC). IEEE Computer Society Press, Brisbane, Australia, Decemeber 1995, pp 410-419.
- [CIG96] C Cifuentes, “Structuring Decompiled Graphs”, Proceedings of the International Conference on Compiler Contrsruction(CC’96), Lecture notes in Computer Science 1060, Linkoping, Sweden, 22-26, April 1996, pp 91-105.
- [CJT01] C. Cifuentes and M.Van Emmerik, Recovery of jump table case statements from binary code, Science of Computer Programming, 40(2-3): 171-188, July 2001.
- [CL03] Cullen Linn, Saumya Debray, “Obfuscation of Executable Code to Improve Resistance to Static Disassembly”, 10th ACM Conference on Computer and Communications Security(CCS), pages 290-299, October, 2003.
- [CLS03] Cullen Linn, Saumya Debray, John Kececioglu, “Enhancing Software Tamper-Resistance via Stealthy Address”, ACSAC2003.
- [CS91] C. Stephens, B. Cogswell, J. Heinlein, G. Palmer, and J. Shen, “Instruction Level Profiling and Evaluation of the IBM RS/6000”,

Proceedings of 18th Annual International Symposium on Computer Architecture, Canada, May 1991, pp. 180-189.

- [BS02] Benjamin Schwarz, Saumya Debray, Gregory Andrews, “Disassembly of Executable Code Revisited”, in Proc. IEEE 2002.
- [EB91] E.A. Brewer, C.N.Dellarocas, A.Colbrook, and W.E. Weihl, “Proteus: A High Performance Parallel-Architecture Simulator”, Technical Report MIT/LCS/TR-516, MIT, Sept. 1991.
- [ECFG] H.Theiling, “Extracting Safe and Precise Control Flow from Binaries”, Proceedings of the 7th Conference on Real-Time Computing Systems and Applications, Dec. 2000.
- [ELFM] ELF man pages, SunOS 5.4, July 1990.
- [FB92] F.B. Cohen, “Operating System Protection Through Program Evolution”, <http://all.net/books/IP/evolve.html>, 1992.
- [FC86] Fred Chow, A. M. Himmelstein, Earl Killian and L. Weber, “Engineering a RISC Compiler System,” IEEE COMPCON, March 1986.
- [FRAVIA] <http://www.woodmann.net/fravia/index.htm>.
- [GDB] GDB: The GNU Project Debugger, <http://www.gnu.org/software/gdb/gdb.html>.
- [GH99] Galen Hunt and Doug Brubacher, “Detours: Binary Interception of Win32 Functions”, Proceedings of the 3rd USENIX Windows NT Symposium, pp. 135-143, Seattle, WA, July 1999.
- [GK91] Gerry Kane, Joe Heinrich, “MIPS Risc Architecture(2nd Edition)”, Prentice Hall PTR, September 1991.
- [HD91] Helen Davis, Stephen R. Goldschmidt, and John Hennessy, “Multiprocessor Simulation and Tracing Using Tango”, in Proceedings of the 1991 Conference on Parallel Processing, pages II-99-II-107, August, 1991.
- [IDAP] <http://www.datarescue.com/>.
- [INSH] install shield, <http://www.installshield.com/>.
- [JB99] J.Bergeron, M.Debbabi, M.M. Erhioui, and B.Ktari, “Static Analysis of Binary Code to Isolate Malicious Behaviors”, in 8th Workshop on Enabling Technologies: Infrastructures for Collaborative Enterprises, 1999.
- [JE93] Jack E. Veenstra and Robert J. Fowler, “Mint tutorial and user manual”, Technical Report 452, The University of Rochester, Computer Science Department, Rochester, New York, November, 1993
- [JH93] J. Hennessy and D.Patterson, “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufman Publishers: san Mateo, CA, 1993.
- [JL00] John R. Levine, “Linkers and Loaders”, Morgan Kaufmann Publishers, 2000.
- [JL94] J. Larus and T.Ball, “Rewriting Executable Files to Measure Program Behavior”, Software Practice and Experience, Volume24, Number 2, Feb. 1994, pp. 197-218.
- [JP94] J. Pierce and T. Mudge, “Idtrace: A Tracing Tool for i486 Simulation”, Technique Report CSE-TR-203-94, Dept. of Electrical Engineering and Computer Science, University of Michigan, Jan. 1994.

- [JR97] J.R. Nickerson, S.T.Chow, H.J.Johnson, “Tamper Resistant Software: Extending Trust into a Hostile Environment”,
- [LINK] Linker and Libraries Manual, Nov. 1993.
- [MC03] M.Christodorescu and Somesh Jha, “Static Analysis of Executables to Detect Malicious Patterns”, in 12th USENIX Security Symposium, 2003.
- [MF83] Michael Farley, Trevor Thompson, “A C Source Language Debugger”, in Proceedings of the 1983 Usenix Summer Conference, Toronto, Ontario, Canada, July, 1983.
- [MG87] Marek Gondzio, “Microprocessor Debugging Techniques and Their Application in Debugger Design”, Software-Practice and Experience, 17(3):215-226, March 1987.
- [MG91] M. Golden, “Issues in Trace Collection Through Program Instrumentation”, MS Thesis, Department of Electrical and Computer Engineering, The University of Illinois, Urbana-Champaign, 1991.
- [MP02] Matt Pietrek, “An In-Depth Look Into the Win32 Portable Executable File Format”, MSDN Magazine, February 2002.
- [MP94] Matt Pietrek, "Peering Inside PE: A Tour of the Win32 Portable Executable Format", Microsoft Systems Journal, Vol. 9, No. 3, pg 15-34, March 1994.
- [OBJD] Objdump, GNU Manuals Online, GNU Project – Free Software Foundation, http://www.gnu.org/manual/binutils-2.10.1/html_chapter/binutils_4.html.
- [PB90] Peter B.Kessler, “Fast Breakpoints: Design and Implementation”, the ACM/SIGPLAN Conference on Programming Languages Design and Implementation, 1990.
- [PER96] Paul E. Roberts, “Implementation and Evaluation of Data Breakpoint Schemes in an Interactive Debugger”, thesis, Department of Computer Science, the University of Utah, December, 1996.
- [PIS] Pentium Instruction Set Reference Manual, <http://developer.intel.com/design/pentium/manuals>.
- [PPCM] PowerPC Microprocessor Family: The Programming Environments for 32-bit Microprocessors, IBM Technical Library.
- [RH] Resource Hacker, <http://www.users.on.net/johnson/resourcehacker/>
- [RL04] Robert Lemos, “Will Code Check Tools Yield Worm-proof Software?”, CNET News.com, May 2004.
- [RSM04] Lenny Zeltser, “Reverse Engineering Malware”, www.zeltser.com/sans/gcjh-practical/revmalw.html.
- [SICT00] B.De Sutter, B.De Bus, De Bosschere, P. Keyngnaert, and B.Demoen, “On the Static Analysis of Indirect Control Transfers in Binaries”, Proc. International Conference on Parallel and Distributed processing Techniques and Applications (PDPTA), 2000.
- [SimOS97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Steve Herrod, “Using the SimOS Machine Simulator to Study Complex Computer Systems”, in ACM TOMACS Special Issue on Computer Simulation, 1997.

- [SimOS98] Stephen A. Herrod, "Using Complete Machine Simulation to Understand Computer System Behavior", Ph.D. Thesis, Stanford University, February 1998.
- [SL02] Shengying Li, Lapchung Lam, Tzicker Chiueh, "Tamper Resistance Tool: Warrior of Software Protection", Technical Report, Rether Network Inc., August, 2002.
- [SL04] Shengying. Li, Lapchung Lam, Tzicker Chiueh, "Sphinx: a Tamper Resistant Software- Through Polymorphism and Randomization", Technical Report, Rether Network Inc., February, 2004.
- [SLB04] Shengying Li, Tzicker Chiueh, "Fast Binary Interpreter with the Aid of Debugger", CSE684 Project Final Report, December, 2003.
- [SMI00] Shaun Miller, "Generating and Deploying Debug Symbols with Microsoft Visual C++ 6.0", <http://msdn.microsoft.com/library/en-us/dnvc60/html/gendepdebug.asp>.
- [SP01] Szor, P., Ferrie, P., "Hunting for Metamorphic", Virus Bulletin Conference, September 2001.
- [SUN] dbx, Sun Microsystems, Inc., "Debugging Tools", Part No. 800-1775-10, May 1988.
- [TO03] T.Ogiso, Y.Sakabe, M.Soshi, and A. Miyaji, "Software Obfuscation on a Theoretical Basis and its Implementation", IEICE Transactions on Fundamentals, E86-A(1), 2003.
- [TR97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad, Instrumentation and Optimization of Win32/Intel Executables Using Etch, In USENIX Windows NT Workshop, August 11-13, 1997.
- [UPX] the Ultimate Packer for executables, <http://upx.sourceforge.net/>.
- [UQBT] C.Cifuentes and m.Van Emmerik, UQBT: Adaptable Binary Translation at Low Cost, IEEE Computer, 33(3):60-66, March 2000.
- [VI04] Vijay Janapa, Reddi, Alex Settle, and Daniel A. Connors, University of Colorado; Robert S. Cohn, Intel Corporation (USA), "Pin: A Binary Instrumentation Tool for Computer Architecture Research and Education," Workshop on Computer Architecture Education Held in conjunction with the 31st International Symposium on Computer Architecture, Munich, Germany, June 19, 2004
- [VN90] Vern Paxson, "A Survey of Support For Implementing Debuggers", available from lbl.gov/papers/debuggersupport.ps.Z, October, 1990.
- [WIND] "Debug and Release Configurations", MSDN library, Visual Studio.