Digital Forensic Analysis on Runtime Instruction Flow[⋆]

Juanru Li, Dawu Gu, Chaoguo Deng, and Yuhao Luo

Shanghai Jiao Tong University, Shanghai 200240, China
jarod@sjtu.edu.cn

Abstract. Computer system's runtime information is an essential part of the digital evidence. Current digital forensic approaches mainly focus on memory and I/O data, while the runtime instructions from processes are often ignored. We present a novel approach on runtime instruction forensic analysis and have developed a forensic system which collects instruction flow and extracts digital evidence. The system is based on whole-system emulation technique and analysts are allowed to define analysis strategy to improve analysis efficiency and reduce overhead. This forensic approach and system are applicable to binary code analysis, information retrieval and malware forensics.

Keywords: Digital forensics, Dynamic analysis, Instruction flow, Virtual machine, Emulation.

1 Introduction

Dynamic runtime information such as instructions, memory data and I/O data is a valuable source of the digital evidence, and is suitable for reconstructing system events due to its dynamic characteristic. Traditional digital forensic techniques are sufficient to extract information from memory and I/O data, but to observe runtime instruction flow, a low-level description of a program's behavior, more studies are needed. Network intrusion and malicious behavior are often carried out by a set of program instruction, leaving few evidence on hard disk, reducing the effectiveness of media forensics and increasing the importance of instruction analysis in digital investigations.

Two challenges in extracting evidence from instruction flow are the difficulties of data tracing and evidence distinguishing. Compared to other types of dynamic information, instruction flow is hard to be captured. Instructions are executed on the CPU instantaneously and are more volatile than memory data. Meanwhile, the CPU will produce a huge amount of instructions because of the high execution speed. Known techniques on capturing instruction flow are in two different ways. The First and the most well researched is the debugging technique. A debugger could control a process or even an operation system, and could trace the runtime information. But it is hard to record instruction flow

[⋆] Supported by SafeNet Northeast Asia grant awards.

completely. Moreover, debugging will affect debuggee's behavior. So apparently, debugging is not suitable for evidence collecting on instruction flow. The second technique is virtual machine monitoring. Virtualization is widely used in security analysis, it could observe and capture the privileged operations. But to collect the whole set of instructions using virtualization is not so convenient. Even if the instruction flow could be traced, the huge number of instructions are in form of opcode. It is impossible to manually analyze the flow. Automatic analysis technique must be used to extract useful information. Current technique of binary analysis couldn't be operated directly on instruction flow. Advances in tools and techniques to perform instruction flow forensics are needed.

To solute the problems above, we have developed a series of techniques and tools. The main contributions of this paper are:

– *Evidence from the instruction flow.* Forensic analysis requires the acquisition of many different types of evidence. We have proposed a novel view on capturing and analyzing instruction flow, which extends the range of digital evidence.

– *Emulator with generic analysis capability.* We have implemented a whole-system emulator based on bochs[2] to achieve instruction capturing. Windows and Linux application could be analyzed on this emulator. And our forensic analysis is compatible to various application.

– *Conditional instruction record and automatic data recovering.* We've provided an extensible interface to let analyst define which instruction should be captured, so as to to reduce the amount of record data. Conditions include time, memory Address, operands value and types of instructions. We've provided a series of tools and scripts to deal with the captured instruction flow. The functions of these tools include string searching, simple structure recognition and related data searching. We have also proposed some universal patterns related to certain encrypt algorithm like DES, which helps analyzing such algorithm more effectively based on instruction flow.

– *Efficiency and accuracy.* We have evaluated the capabilities, efficiency and accuracy of our forensic system. The result shows that the running speed of the system with instruction record is acceptable and the pattern based analysis could locate the accurate event or algorithm automatically.

The remainder of the paper is organized as follows. Section 2 introduces the characteristic of the instruction flow and how to use instruction flow as digital evidence. Section 3 describes our forensic analysis technique in detail. Section 4 gives the implementation of our forensic system. Experimental evaluation is described in Section 5 and Section 6 offers conclusion.

2 Background

The instruction flow is an abstract concept that describes a stream of instructions from the process of program execution. When programs are executed, static instructions are loaded into memory and fetched by the CPU. After each clock cycle of the CPU, the executed instruction with its operands is determined. Thus the sequence of the executed instructions composes a flow. Instruction flow contains not only data, but also how data is operated, thus is helpful on reconstructing system events. Additionally, recent researches on virtual machine security shows that instruction level analysis is an important aspect of computer security[10][13]. This section describes the characteristic of the instruction flow and how to extract digital evidence from the instruction flow.

2.1 Characteristics of the Instruction Flow

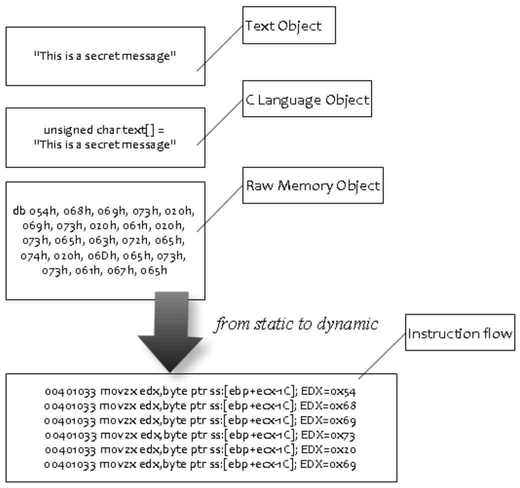
The instruction flow is different from the information flow or the data flow. It is a flow that contains information about low-level operation yet provides more details about the system's status. Like packet in a network dataflow, the basic unit in an instruction flow is the single instruction. Properties of the instruction are important for analysis. First, instructions in a flow are ordered by time, and the same instruction could be executed repeatedly and appears in different positions of the flow. Notice that in the instruction flow, operands are bind to instructions, as illustrated in Figure 1. So even two instructions in the different places of a flow are the same, analyst could learn more from the position and operands. What's more, an instruction could be loaded into different memory addresses of different processes. Same instruction performs distinctly at different virtual memory addresses. Another property is that branch instruction is useless during analysis because the execution path is determined when the flow is generated. Finally, the form of the instruction flow remains the same despite of the changing of upper level operation system. So the same forensic analysis technique could be used ignoring platform differences.

2.2 Instruction Flow as Digital Evidence

Individual disk drives, RAID sets, network packets, memory images, and extracted files are the main source of traditional digital evidence. But taking instruction flow as a source of digital evidence is practical. A typical scenario for the application of instruction flow analysis is the malware analysis[6], which allows analyst to use a controllable, isolated system to test the program and determines whether the behavior is malicious. Consider the event that a Trojan horse program acquires the password, encrypts it with a fix public key and sends it to a remote server. The public encrypt algorithm, public key and remote server's address are all useful evidences. Obviously these evidences are included in the instruction flow, but how to effectively recognize them in a huge quantity of instructions is a problem. Our work gives an approach on how to analyze instruction flow and search digital evidence.

3 Forensic Analysis on Runtime Instruction Flow

Two main steps are essential to perform forensic analysis on runtime instruction flow. First, the instructions are recorded and the instruction flow is generated.



```
"This is a secret message"
```

```
unsigned char text[] =
"This is a secret message"
```

```
db 054h, 068h, 069h, 073h, 020h,
069h, 073h, 020h, 061h, 020h,
073h, 065h, 063h, 072h, 065h,
074h, 020h, 06Dh, 065h, 073h,
073h, 061h, 067h, 065h
```

```
00401033 movzx edx,byte ptr ss:[ebp+ecx-1C]; EDX=0x54
00401033 movzx edx,byte ptr ss:[ebp+ecx-1C]; EDX=0x68
00401033 movzx edx,byte ptr ss:[ebp+ecx-1C]; EDX=0x69
00401033 movzx edx,byte ptr ss:[ebp+ecx-1C]; EDX=0x73
00401033 movzx edx,byte ptr ss:[ebp+ecx-1C]; EDX=0x20
00401033 movzx edx,byte ptr ss:[ebp+ecx-1C]; EDX=0x69
```

Fig. 1. From string to instruction flow

Second, after the instruction flow data is acquired, automatic analysis should be introduced to efficiently process the data and find out useful information. The following two subsections discuss these two steps and then a standard form of evidence from instruction flow is proposed.

3.1 Instruction Flow Generating

To capture instructions directly from the execution process, the CPU must be interrupted on every instruction. A trap flag based approach is introduced in [4]. We choose emulation to fulfil the capture function because it is simple and clear, the implementation detail of the system is described in Section 4. Another important problem is to decide the form of recorded instruction flow. We choose a data-instruction mixed form to record the flow, that is to say, each instruction's opcode, operands and memory address are recorded as a single unit and these units are ordered by time to compose a flow. Two modes are supported in instruction flow generating process:

Complete Record. In this mode, the instruction flow contains every instruction executed by the CPU. The data amount is huge and the running speed of the emulated system will be affected. This mode could bring the most precise record, yet sacrifice efficiency and storage space. Although the amount of instruction flow is large, to collect it is practical. In our experiment, the execution will produce about 1G Byte raw data per minute. That is almost the same volume a raw video stream produced by a DV camera, thus acceptable to store. When analyzing, it is suggested that the conditional record mode introduced below be used first to get some clues, and use these clues to guide the complete record.

Conditional Record. In the execution process, many instructions are useless for analyzing. To reduce the redundant data, various conditions could be used to filter the instruction flow. We've designed an open interface which allows analyst to define their own filtering conditions and the combination. Conditions supported by our system are listed below:

– *Time.* If the analyst knows the start and the end of a specific behavior, the record process could be set to start and stop at certain time point. One situation is to start recording after the boot of operation system.

– *Memory Address.* The CPU executes instruction by fetching it from memory, and the virtual memory address of the instruction is a special feature. For system calls, their entry points are already known and could be used as a condition to determine program behavior. More flexibly, analysts are allowed to capture or filter a range of memory address. A very effective strategy to monitor application on Windows is to filter off instructions with memory address higher than 0x70000000, which belongs to Kernel and system service processes. The same strategy is applicable on analyzing Linux(See Figure 2)

– *Instruction type.* Different analysts may concern different types of instructions. Analysts could determine which types of instructions should be captured, thus constructing specific instruction flow. For instance, if the forensic analysis focuses on encrypt algorithms, arithmetic instruction such as XOR is important while others could be filtered off.

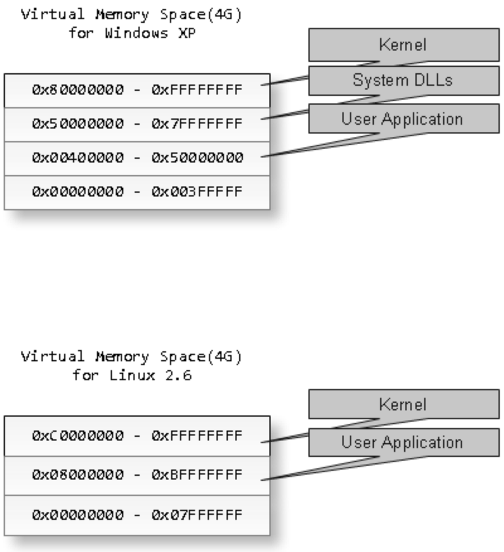


Fig. 2. Memory Allocation in Windows and Linux

– *Operands.* The value of Operands illustrates the content of an operation. To search a string in an instruction flow, analyst could first focus on the instructions with certain value of operands. And operands are a good feature that seldom change if the algorithm and input data are fixed. So code protection is invalid to hide information when using operands value as feature.

Using such conditions and their combination to filter the instruction flow, the data amount could be reduced to a considerably small size.

3.2 Analysis of the Instruction Flow

After collecting of instruction flow, analysis is ready to start. The aim of traditional binary analysis is to reconstruct high-level abstraction of the code. But in the instruction flow analysis process, the core part is data abstraction. The main purpose of the analysis is to express data in a clear form, and to find evidence through data. Two modes are supported in our analysis environment: offline analysis and online analysis. When the analysis runs in the offline analysis mode, saved instruction flow is analyzed, while in online analysis, our system directly analyzes instruction flow in memory.

Offline Analysis. In offline analysis mode, instruction flow is saved first and then scanned multiple times. We developed a series of tools and scripts to deal with the collected instruction flow. The first step is to analyze the data recorded in conditional mode. The provided automatic tools check the data bind to each instruction and maintain a sequence of data related. In low-level language most of the strings and arrays are operated with the same instruction for many times, so a large part of the data information can be recognized after this operation. The second step is to find useful information. Readable strings are automatically listed and are related to instructions. The related instructions are selected as clues of digital evidence. The final step is to run a complete record to gather a full set of instructions that operates the information, and use the selected instructions to slice the program and extract useful fragments.

Online Analysis. Although to analyze realtime instruction flow loses lots of context information, the profit is apparent. Less storage space is needed and running speed of emulation is expected to be faster. Online analysis is a debug-like analysis, which allows analyst to use some strong pattern(e.g. specific memory address, certain opcode) to quickly locate the suspicious instructions. In this mode the forensic system also plays the role of a debugger and supports all traditional debugging technologies.

3.3 Evidence from the Instruction Flow

One question about the instruction flow forensic analysis is how to give a convincing evidence. We propose a format of evidence from the instruction flow which the extracted evidence should follow:

1. **Data information from the instructions**
   Data information from the instruction is the core part of the digital evidence. It can be string information, IP address, URL or any other readable information. These kinds of data illustrate the analyzed events' properties.
2. **Related instruction set**
   The instructions that operate the data information should be provided as supporting evidence to illustrate the generation and transformation of the data.
3. **External supporting data**
   External supporting data such as Memory dump, Network flow, I/O data is collected via black box analysis. These kinds of data could be analyzed by traditional forensic analysis to support the evidence from the instructions.
4. **Testing environment**
   Testing environment should also be provided so that other analysts could replay the analysis.

4 Implementation

In this section we describe the implementation detail. To monitor the program's behavior and capture its instruction flow, a virtual environment is necessary. We choose bochs[2], which is an open source IA-32 (x86) PC emulator written in C++, to build this environment. In bochs we can run most operating systems inside the emulation, including Linux, DOS and Windows. Moreover, bochs is a typical CPU emulator that has a well designed structure for adding monitoring function with little performance overhead[7]. By using CPU emulation, analysts could collect instruction flow and trace software's activity, while the risk of evidence tampering is reduced.

Figure 3 shows the architecture of our forensic system. We have designed an engine on the bochs emulator to deal with the instruction flow. The engine will read parameters from a configuration file first, and analysts are able to set conditional filter parameters in this file. Then, when the emulation starts, the engine filters each instruction according to the configuration and fulfil a conditional record. A buffer in memory is maintained to record the instruction flow, and the data isn't written back to hard disk unless it reaches the buffer's capacity. Real-time data compression mechanism is optional for the buffered data to reduce the storage. We've also provided scripts in perl and python to automatically analyze instruction flow.

5 Evaluation

For digital forensics, accuracy is the most important factor. The using of emulation imports less interference to the analyzed object, yet sacrifices the efficiency. So one essential target of forensic emulation is to decrease emulation overhead. Several measures have been adapted. First, we use Windows PE and SliTaz GNU/Linux as testing operation system platform because these two systems are

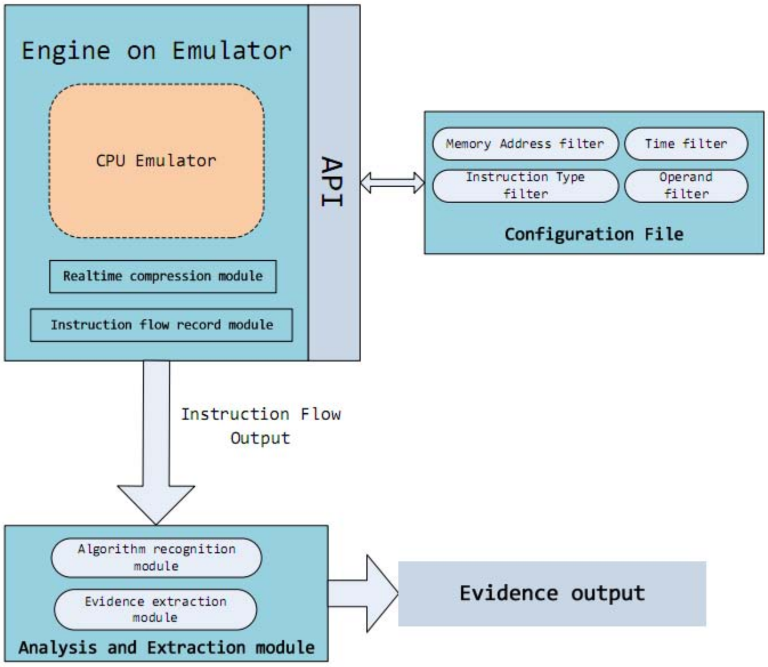


Fig. 3. The architecture of the forensic system

the lightweight version of the currently most widely used OS, and provide complete environment with GUI. Second, the running speed is 10-100 times slower in complete record mode than the original emulation due to the delay of hard disk writing. In order to improve the speed, an SSD driver is used to collect instruction flow and conditional record mode is suggested to be used. A typical configuration for Windows program analysis is shown in Table 1:

Table 1. An typical configuration on analyzing Windows program

Parameter	Configuration
Platform	Windows PE 1.5 (with kernel same as Windows XP SP2)
Range of Memory address	instruction with address $\leq$ 0x70000000
Instruction type	arithmetic, logical and bit operation
Record Time	-
Range of Operands	-

In real world, a program may use crypto algorithm to hide information. The private key and the algorithm are the most important evidences[5]. We give a forensic analysis on a Linux program that hides string information through DES encryption to show how our system works.

Table 2. Search result of the instruction flow

Seq No.	address	opcode	value of operands
143001	0x80486C9	*MOV EAX,[offset]*	[offset]==57
143025	0x80486C9	*MOV EAX,[offset]*	[offset]==49
143049	0x80486C9	*MOV EAX,[offset]*	[offset]==41
143073	0x80486C9	*MOV EAX,[offset]*	[offset]==33
143097	0x80486C9	*MOV EAX,[offset]*	[offset]==25
143112	0x80486C9	*MOV EAX,[offset]*	[offset]==17

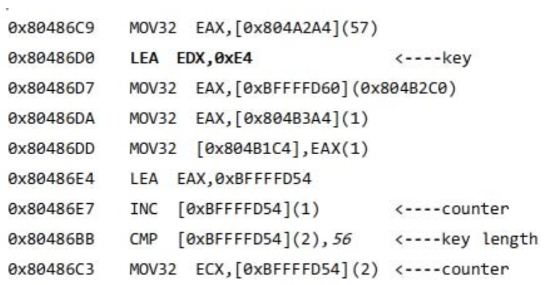

```
0x80486C9    MOV32  EAX,[0x804A2A4](57)
0x80486D0    LEA  EDX,0xE4              <----key
0x80486D7    MOV32  EAX,[0xBFFFFD60](0x804B2C0)
0x80486DA    MOV32  EAX,[0x804B3A4](1)
0x80486DD    MOV32  [0x804B1C4],EAX(1)
0x80486E4    LEA  EAX,0xBFFFFD54
0x80486E7    INC  [0xBFFFFD54](1)       <----counter
0x80486BB    CMP  [0xBFFFFD54](2),56    <----key length
0x80486C3    MOV32  ECX,[0xBFFFFD54](2) <----counter
```

Fig. 4. A DES encryption loop

The tested program is a Linux ELF file. Before checking up the private key, we should first determine whether this program uses the DES algorithm. We configure the forensic system for Linux environment, restricting the range of memory address from 0x08000000 to 0x10000000 and the value of operands: only the instructions with operands less than 0x100 are to be record. Then the system records the running process of the program on Slitaz Linux 3.0. We collect an instruction flow and use scripts to search for the *Permuted choice 1* of DES[3]: $\{57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36, 63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 28, 20, 12, 4\}$ The search gives a solitary result shown in Table 2. The result shows a strong feature of DES encryption. After the search we run the system again in complete record mode and locate the address 0x80486C9. According to the specification of DES, Permuted choice 1 is directly linked to main key. A simple program slicing on 0x80486C9 will give a loop of 56 times. Check the loop(see Figure 4), the private key is easily extracted.

6 Related Work

The topic of forensic analysis on low-level, dynamic information has attracted many researchers. Tools for volatile memory analysis and for program behavioral analysis have been developed. FATKit[8] provides the capability to extract higher level objects from low-level memory images. But memory image can't describe the behavior of program in detail. Capture[9] is a behavioral analysis tool based on kernel monitoring, which could analyze binary behavior. One shortage of Capture is that it focuses on system call rather than program's instruction. Although this would bring abstraction and convenience for analysis, a more fine-grained analysis on binary code is required.

Our work is to introduce low-level instruction analysis to forensic system. Prior to our work, some tools have provided analysis functions focusing on certain aspects. Rotalumé[10] and TEMU[13] are emulation systems based on the QEMU emulator[1]. The target of these systems is to provide syntax and semantics of the binary code, in other words, they try to transfer binary code to a high-level abstraction concept rather than collect detail evidence. Our system targets at collecting data from the instruction flow, providing not only an emulator but also a series of tools and methods to do forensic analysis on dynamic instructions.

7 Conclusion

In this paper we have presented a novel approach for forensic analysis and digital evidence collection on the instruction flow. We have presented details of a forensic system based on emulation. This forensic system deals with dynamic instructions. Functions of the system include: (1) generation of instruction flow, (2) automatical analysis of the instruction flow, (3) extraction of digital evidence. The system also provides a flexible interface which enables analysts to define their own strategy and augment analysis.

References

1. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, p. 41 (2010)
2. bochs: The Open Source IA-32 Emulation Project, `http://bochs.sourceforge.net`
3. FIPS 46-2 - (DES), Data Encryption Standard, `http://www.itl.nist.gov/fipspubs/fip46-2.htm`
4. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 51–62 (2008)
5. Maartmann-Moe, C., Thorkildsen, S., Årnes, A.: The persistence of memory Forensic identification and extraction of cryptographic keys. Digital Investigation 6 (supplement 1), 132–140 (2009)
6. Malin, C., Casey, E., Aquilina, J.: Malware forensics: investigating and analyzing malicious code. Syngress (2008)
7. Martignoni, A., Paleari, R., Roglia, G., Bruschi, D.: Testing CPU emulators. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, pp. 261–272 (2009)
8. Petroni, N., Walters, A., Fraser, T., Arbaugh, W.: FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory. Digital Investigation 3(4), 197–210 (2006)

9. Seiferta, C., Steensona, R., Welcha, I., Komisarczuka, P., Popovskyb, B.: Capture - A behavioral analysis tool for applications and documents. Digital Investigation 4 (supplement 1), 23–30 (2007)
10. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic Reverse Engineering of Malware Emulators. In: 30th IEEE Symposium on Security and Privacy, pp. 94–109 (2009)
11. SliTaz GNU/Linux (en), http://www.slitaz.org/en/
12. What Is Windows PE?, http://technet.microsoft.com/en-us/library/dd799308WS.10.aspx
13. Yin, H., Song, D.: TEMU: Binary Code Analysis via WholeSystem Layered Annotative Execution. Submitted to: VEE 2010, Pittsburgh, PA, USA (2010)