Impact of Environment on Branch Transfer of Software

Jianming Fu[1,2(✉)], Yan Lin[1,2], and Xu Zhang[1,2]

[1] State Key Laboratory of Aerospace Information Security and Trusted Computing of the Ministry of Education, Wuhan 430072, China
jmfu@whu.edu.cn
[2] Computer School, Wuhan University, Wuhan 430072, China

Abstract. Current intrusion detection approaches based on control flow integrity (CFI) can detect the majority of control flow hijacking attacks, but few of them take into account the impact of environment on CFI, so there may exist false alarms. In this paper, we have investigated systematically the impact of environment on branch transfer from time, space and mechanisms of Linux operating system. Moreover, we have presented finite state automata (FSA) to describe difference patterns caused by these environmental factors, and have exploited FSA-Stack model to detect these impacts. Finally, for some common applications (gzip, grep, tesseract, bzip2 etc.), we have leveraged a dynamic binary instrumentation tool Pin to record direct and indirect branch transfers produced by them and the shared libraries they depend on. The experimental results demonstrate that impact of environment on branch transfer exists universally and normally among usual applications, and the difference patterns of impacts can be beneficial to understand and mitigate the false alarms of CFI.

Keywords: Intrusion detection · Control flow integrity · Environmental factors · Finite state automata · Dynamic binary instrumentation

1 Introduction

Software (Program) behavior is a sequence of states or state transitions, which can be described by the low-level machine code or the high-level program statements, functions and system calls. Generally, software behavior is used in intrusion detection [1], but, it is dependent on the running environment and may be prone to be bypassed by attackers. Many existing intrusion detection techniques rely on monitoring the sequence of system calls a program invoked [1,2], or the arguments of system calls [3], or both [4] to detect malicious behaviors. However, if the attacker does not use the monitored system calls to achieve his goal, this kind of detection can be bypassed easily. For example, code-reuse attack [5,6] leverages existing code to form malicious gadgets, instead of using existing functions or system calls to achieve the attack.

R. Deng et al. (Eds.): SecureComm 2016, LNICST 198, pp. 575–593, 2017.
DOI: 10.1007/978-3-319-59608-2_32

In order to counter this kind of low-level control flow hijacking attack, control flow integrity (CFI) [7,8] is proposed, which marks the valid targets of indirect control flow transfers with unique identifiers(IDs), and then inserts ID-checks before each indirect branch transfers. CFIMon [9], CFIGuard [10] and ROPecker [11] collect all possible indirect transfers or potential gadgets, then leverage Branch Trace Store (BTS) or Last Branch Record (LBR) [12] to capture control flow transfers when the program is running. Not only the shellcode but also some environmental factors will affect the control flow. For instance, in Linux, the Global Offset Table and LD_PRELOAD environment variable will have an impact on branch transfer.

According to Zhong [13], software behavior not only depends upon the program itself, but also depends on the running environment, including time, event, space, shared libraries, OS kernel, device driver, Hypervisor, garbage collector, compiler and so on. In this paper, we study the impact of environment on branch transfer of software from time, space and mechanisms of Linux operating system. In the meantime, we have confirmed these impacts based on a dynamic binary instrumentation tool Pin, and analyzed how these factors affect a program's branch transfer. Finally, we have used finite state automata (FSA) to describe difference patterns of branch transfers resulted from these environmental factors. These FSAs can be used to mitigate false alarms in CFIMon and ROPecker, and to distinguish what factors will cause the difference. In general, these FSAs are helpful to detect and counter shellcode.

In summary, this paper makes the following contributions:

- We have investigated systematically impacts of environment on branch transfers from time, space and mechanisms in Linux systems, and have discovered interested observations to understand the false alarms of CFI.
- We have presented a model of Finite State Automata(FSA) to capture these impacts, and have exploited FSA-Stack model to detect these impacts.
- We have designed the convinced experiment to validate impact patterns of different environmental factors, and its results are beneficial to reduce the false alarms of CFI.

The rest of this paper is structured as follows. Section 2 summarizes and discusses related work on intrusion detection based on control flow integrity. Section 3 analyzes environmental factors affecting branch transfer in detail. Section 4 introduces our approach which leverages a dynamic binary instrumentation tool Pin to record all branch transfers. Meanwhile, we give all FSAs that describe the difference patterns of branch transfers caused by environmental factors. Section 5 outlines the experimental results. Section 6 makes some concluding remarks and discusses the limitations of our work.

2 Related Work

Software behavior integrity detection has a long history, from the detection based on audit data and log information [14,15] to the detection based on system

call [1–4,17,18], now it has been extended to control flow integrity detection. The purpose of the extension is to detect attacks that try to divert the program's control flow.

Some of these proposals mark the valid targets of indirect branch transfers with unique identifiers (IDs), and then inserts ID-checks into the program before each indirect branch transfer [7,8,20–23]. For example, Abadi et al. [7,8] introduced the term CFI and they suggested using a single identifier for all indirect branch transfers. Bin-CFI [21] used two IDs for all indirect branch transfers, one for ret and indirect jump instructions, another for indirect call instructions. And all indirect branches are instrumented by means of a jump to a CFI validation routine. But it does not validate the integrity of addresses in the global offset table (GOT), this leaves it be vulnerable to the so-called GOT overwriting attacks. CCFIR [20] implemented a 3-IDs approach, which extended the 2-IDs approach by further separating returns to sensitive and non-sensitive functions. In CCFIR, all targets for indirect branches are collected and randomly allocated on a so-called springboard section, and indirect branches are only allowed to use control flow targets contained in the springboard section. However, memory disclosure can reveal the content of the entire springboard section, which can be leveraged by attackers. DynCFI [23] used a dynamic code optimization tool to enforce CFI detection, which has the same problem with Bin-CFI, as it does not validate the integrity of GOT.

Others often leverages available hardware support for branch recording in commercial processors to collect the sets of control transfers when the program is running, and then compare them with the valid targets collected beforehand [9–11,24]. For instance, CFIMon [9] made use of static analysis and online training to get all valid targets of indirect branch transfers. It leverages BTS mechanism supported by hardware to collect in-flight control transfers and once the BTS buffer is nearly full, a monitor process will start to compare them with the valid targets to decide whether there exists an attack. But the variance of environment variable may cause some indirect jump instructions have different target addresses, which will produce false alarms. Similar CFI polices are also enforced by ROPecker [11]. It collects all potential gadgets beforehand, then leverages LBR mechanism to record all source addresses and target addresses of branches to decide whether there are gadgets. However, it does not take into account the signal mechanism in Linux. Due to the signal handler is in the process address space, ROPecker also will record the branch transfer when it is running, but these records are not in the potential gadgets database gained beforehand.

All these work mentioned above does not take into account environment may have an impact on the control flow integrity. Although Zhao et al. [25] introduced many factors (memory state, operating system kernel, system time etc.) would affect the control flow, but they did not analyze why these factors have impacts on the control flow, and what branch transfers may be affected. In this paper, we introduce some factors that will affect the control flow, and analyze why they have impacts on control flow, and construct branch patterns produced by these factors. There is little work related to environment, we list them in Table 1.

Table 1. Researches related to environment

Approach	Description
Giffin et al. [19]	Take configuration files, command-line parameters, and environment variables into intrusion detection
Mytkowicz et al. [16]	Introduce UNIX environment size and link order have an effect on performance analysis
Zhao et al. [25]	Introduce many factors (memory state, operating system kernel, system time et al.) will affect the program's control flow
This paper	Introduces environmental factors that will impact on the control flow from a fine-grained perspective (branch transfers), and analyzes why they have impacts on the control flow, and gets difference patterns produced by these factors

3 Impact of Environment on Branch Transfer

Ideally, with the same input, the control flow of a program will be the same too. However, the experimental results show that even with a simple program, its control flow may be different in different environments. The difference is the number of branch transfers is nearly the same, but branch transfers are different greatly. In this section, we give definitions related to control flow and factors impacting on branch transfers.

Definition 1. A **basic block** is a sequence of consecutive instructions, without any branches except at end of the sequence. For an arbitrary basic block b, $\mathrm{b} = i_1, i_2, \ldots\ldots, i_k$, if instruction i_j is executed at step n, then instruction i_{j+1} must be executed at step $\mathrm{n}+1 (1 \leq \mathrm{j} < \mathrm{k})$.

Definition 2. Branch transfer can be represented as $I = <From, To>$, where $From$ is the address of the last branch instruction in a basic block. To is the address of the first instruction in consecutive basic block.

Definition 3. Control flow transfer depends on the branch instructions in the program. A program's **control flow** can be represented as: $S_0 \xrightarrow{I_0} S_1 \xrightarrow{I_1} S_2 \ldots \xrightarrow{I_k} S_k \ldots \xrightarrow{I_n} S_n$. Where I_k is a branch instruction, whose address is $From$ in I defined in Definition 2, the targets of the branch instruction is To, S_k is the state of the program. Due to environment will affect the control flow, there is $S_i \times E \rightarrow S_j$, E is the environmental factor.

3.1 Impact of Time

System time will affect the behavior of programs. For example, the dynamic linker will invoke rdtsc (Read Time Stamp Counter) to decide whether to jump. The experimental result shows that the number of a branch transfer

will change. This branch transfer is located in function `HP_TIMING_DIFF_INIT` in `_dl_start_final()`, and it is a direct branch transfer. This branch transfer is the 13th line in `HP_TIMING_DIFF_INIT` shown in Listing 1. When the time difference between __t2 and __t1 is less than the threshold `dl_hp_timing_overhead`, this branch transfer will take happen.

Listing 1. HP_TIMING_DIFF_INIT

```
1   /* Use two 'rdtsc' instructions in a row to find out how long it takes. */
2   #define HP_TIMING_DIFF_INIT() \
3   do { \
4         if (GLRO(dl_hp_timing_overhead) == 0) \
5         { \
6           int __cnt = 5; \
7           GLRO(dl_hp_timing_overhead) = ~0ull; \
8           do \
9           { \
10             hp_timing_t __t1, __t2; \
11             HP_TIMING_NOW (__t1); \  // Gets the current time
12             HP_TIMING_NOW (__t2); \  // Gets the current time
13             if (__t2 - __t1 < GLRO(dl_hp_timing_overhead)) \
14               GLRO(dl_hp_timing_overhead) = __t2 - __t1; \
15          } \
16          while (--__cnt > 0); \
17        } \
18      } while (0)
```

3.2 Impact of Space

Branch transfer is closely related to the program's memory layout. In Linux, a program's memory layout is illustrated in Fig. 1, the stack area includes the command line, environment variables and the context of function calls, dynamically allocated memory area is located in the heap. For example, when the program allocates memory using `malloc` or `new`, the reserved area is protected and not allowed to access. The stack area has a great impact on the branch transfer, which we will introduce in detail later.

Address Space Layout Randomization (ASLR). In Linux, addresses of stack, heap, and shared libraries are randomized using ASLR [26]. In the 32-bit operating system, the 4–23 bit of the stack base address is randomized, the 12–27 bit of the heap and shared libraries base addresses are randomized. As we know, local variables are located in stack, so the address of the local variables may be not the same every time as the program is loaded. However, In some applications, many branches are conducted in accordance with the last 8-bit or 12-bit addresses of local variables, which will cause the targets of the branch transfers become different.

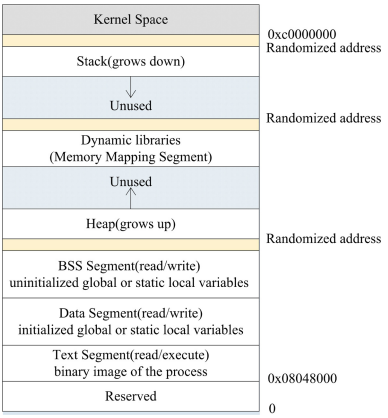

Fig. 1. A process's virtual memory layout in Linux

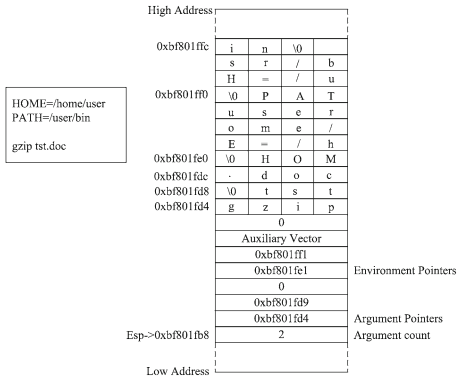

Fig. 2. Initial stack of a process

Environment Variables and Command Lines. The operating system must know something about the running environment before the program starts to run. Such as environment variables and arguments of the process, the operating system will save related information into the stack.

Assume the environment variables and the command-line are as the left side of Fig. 2, and the value of register `Esp` is 0xbf801ffc, then after the program is loaded, the content of the stack is shown as the right side in Fig. 2. Now `Esp` points to the count of the arguments, and the next are pointers linking to arguments and environment variables. `Auxiliary Vector` saves some auxiliary information needed by the dynamic linker, for instance, `AT_ENTRY` and `AT_BASE` respectively represents the entry address of the executable and the loaded address of the dynamic linker.

When the CPU processes memory-related operations (mainly located in shared libraries), it will check whether the address is memory boundary alignment. For example, In the 32-bit operating system, the processor will check whether the address is 4 byte boundary alignment, there will be different operations for different alignment. As shown in Listing 2, in function `strrchr()`, it will check whether the address of the string is 4 byte boundary alignment (testl $3,%esi), if it is, then executes it in sequence, otherwise jumps to L(19).

Listing 2. strrchr function

```
1  ENTRY (strrchr)
2        ......
3        testl 0x3, %esi                /* correctly aligned ? */
4        jz L(19)                      /* yes => begin loop */
5        movb (%esi), %dl
6        ......
7  L(19):movl (%esi), %edx /* get word (= 4 bytes) */
8        movl 0xfefefeff, %edi
9        addl %edx, %edi
```

It shows that the memory address boundary alignment has a significant impact on branch transfer. Usually, the memory address boundary alignment are impacted by the length of environment variables and command-line.

3.3 Mechanisms in Linux

There are many mechanisms in Linux, such as signals, the management of shared libraries, Global Offset Table (GOT), and all of them will have impacts on branch transfers. In this section, we will introduce these impacts in detail.

Searching for Shared Libraries. Linux uses a called SO-NAME (retaining only the shared library's major version number) naming mechanism to record shared library dependencies, meanwhile a symbolic link is created in each shared library's directory which has the same name with its "SO-NAME". And the directories of shared libraries a program depends on are saved in the section of dynamic.

In Linux, there is a procedure called ldconfig, which is responsible for creating, deleting, and updating the symbolic linking, and then collecting these symbolic linking into a file called /etc/ld.so.cache, which has a special structure. And the dynamic linker will directly search for shared libraries from this file. When a new application installs shared libraries into the system, ldconfig is invoked automatically to update the content of /etc/ld.so.cache, thus, the branch transfers that the dynamic linker searches for the shared libraries will change too.

Signal in Linux. Signal is an asynchronous communication mechanism in Linux, which notifies the process what event occurs. When a process P2 sends a signal to a process P1, the kernel will receive this signal, and put it into the signal queue of P1. When the process P1 traps into the kernel, the kernel will check its signal queue and invoke the signal handler according to the corresponding signal number.

Global Offset Table (GOT). In Linux, cross-module access is achieved according to Procedure Linkage Table (PLT) and GOT. The former contains a series of jump entries, and the latter contains the absolute addresses of library functions. For dynamic linking programs, there are many function calls between modules, ELF makes use of an approach named Lazy Binding to accelerate the speed of the dynamic linking.

As shown in Fig. 3(a), when the program first invokes `printf()`, it will jump to `printf@plt` to execute instruction jmp *0x804a000, which links to the address of the instruction push 0x0, 0 is the reference index in the relocation table `.rel.plt` for symbol `printf`, then it will invoke function `_dl_runtime_resolve` to achieve symbol resolution and relocation, and then patch the address of `printf` into GOT. When the program calls `printf` function again, its procedure

is shown as Fig. 3(b), also it will jump to `printf@plt` to execute instruction jmp *0x804a000, but now the pointer *0x804a000 points to the address of `printf`, it will call `printf` directly. In this procedure, the number of branch transfers this time is two times less than the first time. Also this mechanism leaves it be vulnerable to so-called GOT-overwriting attacks.

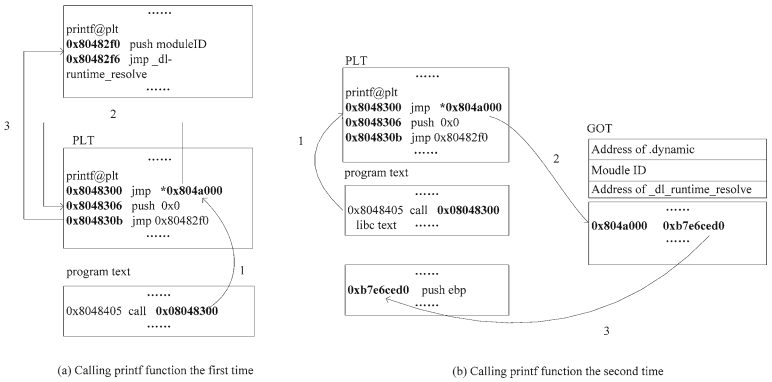

Fig. 3. Process of calling shared library function in Linux

Configuration of Environment Variables. In Linux, many environment variables can be used by attackers, such as attackers can leverage `LD_PRELOAD` to load shared libraries that they defined. The file specified in `LD_PRELOAD` will be loaded before the dynamic linker searches for shared libraries in accordance with fixed rules. And as a result of the existence of global symbol mechanism, global symbols specified in the shared libraries through `LD_PRELOAD` will cover the same global symbols specified in the normal shared libraries, which makes it easy to modify the functions in standard libraries. Meanwhile attackers also can modify the configuration file /etc/ld.so.preload to load the target files.

4 Software Behavior Analysis

We have recorded all direct and indirect branch transfers in the program and shared libraries it depends on based on the dynamic binary instrumentation tool Pin [27] on x86 32-bit version of Ubuntu 12.04, and compared the difference between the branches to get the behavior patterns caused by factors mentioned above, these patterns are described using Finite State Automata.

4.1 Environment and Branch Transfer

In Sect. 3, we have introduced many environmental factors that will impact on branch transfers. In this section, we will introduce how these factors affect the direct and indirect branch transfers of the program and the shared libraries, the results are shown in Table 2.

Table 2. Impact of environmental factors on branch behavior

Factor	Item	Shared library		Program	
		Direct branch	Indirect branch	Direct branch	Indirect branch
Time	Time	Y			
Space	ASLR	Y	Y		
	Environment variable length	Y	Y		
	Command line length	Y	Y	Y	
Mechanims of Linux	GOT	Y	Y	Y	Y
	Searching for shared libraries	Y			
	Linux singal	Y	Y	Y	Y
	LD_PRELOAD	Y	Y		

Same Input for Same Program. All factors mentioned in Sect. 3 will affect the direct branch transfers in shared libraries even with the same input. These direct branches mainly belong to jump instructions and call instructions in the same function. There is an observation in these differences of branches: just the last bits of the source address (relative address) are different, and the target address is the same. Moreover, the indirect instructions affected mainly are `ret` instructions and `jmp` instructions in the same function, but the difference is the source address is the same, the target addresses are different. The branch transfer of the program itself is all the same when the input is the same. When the input is the same but the length of the directory that the input file is located in are different, the branch transfer of the program itself will be different on direct branches, and there is no impact on indirect branches in program itself.

Different Inputs for Same Program. When the inputs are different, there will be a great differences on branch transfers, especially for direct branches in shared libraries. We do not take into account the impact on direct branch transfers when the inputs are different, also because the most majority of attacks just leverage indirect branches to achieve their goals.

The difference on indirect branches are mainly indirect call instructions and fast system call instructions, that is to say different inputs will lead to different function calls and system calls.

4.2 Representation of Differences

We use the dynamic binary instrumentation tool Pin to record all branch transfers, and then compare them using the tool `diff` in Linux. For example, we run the program graphicsmagic to convert the format of a picture two times, and get the branch transfers in libc.so, the difference is shown in Fig. 4. These two hexadecimal number are the source and target addresses respectively, the number in the last column is the version number of the shared library, the differences

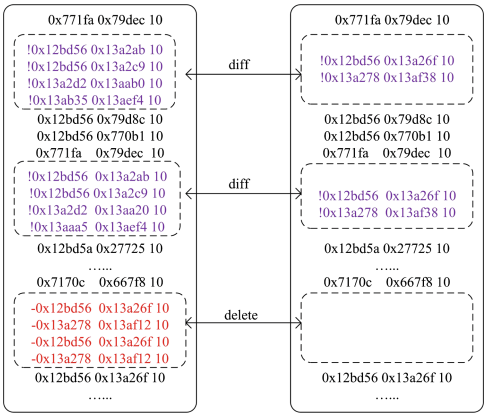

Fig. 4. Comparison of branch record

are shown in the dotted box, where '!' denotes they are different, '-' denotes these branch transfers should be deleted.

Definition 4 (Concepts of FSA). Difference pattern can be described as a sequence of addresses $<addr_0, (*), addr_1, ..., (*), addr_n>$, where $addr_k$ is the source address, $0 < k < n$, $addr_0$ and $addr_n$ can be the source or the target address, $(*)$ represents there may be other source addresses that do not equal to $addr_{i+1}$ between $addr_i$ and $addr_{i+1}$. For instance, the sequence of $<0x12bd56$, $0x13a278>$ represents that the difference pattern is the source address of a branch is 0x12bd56, and the next branch's source address is 0x13a278.

Definition 5. Finite State Automata (FSA) includes five parts (Σ, S, S_0, T, F), and the meaning of each part is as follows:

1. Σ is the input symbol set. In this paper, it is the set of source and target addresses.
2. S is the state set. In this paper, it is a set of number from 0 to n.
3. S_0 is the initial state.
4. T is the state transition function, which gives the subsequent state according to the state in S and the symbol in Σ.
5. F is the accepted state.

Representation of Difference Patterns Using FSA. We leverage FSA to describe the impact of environment on branch transfers. The test programs are shown in Table 3, which were run on an Intel Core i7 processor with 32-bit Ubuntu 12.04 system.

Input is the same. When the input is the same, the differences are mainly located in the shared library libc.so. In this section, we mainly discuss differences produced by libc.so for programs in Table 3.

Table 3. Test programs

Application	Size	Experiment
gzip	806 KB	Compress multiple files
grep	153.7 KB	Search regular expression in multiple files
bzip2	30.2 KB	Compress multiple files
bubblesort	7.3 KB	Sort different inputs
cat	46.8 KB	Connect two files
diff	112.3 KB	Compare two files
tesseract	236.9 KB	Recognize multiple license plates
graphicsmagic	5.4 MB	Connect two pictures, convert pictures into different forms
bunzip2	30.2 KB	Uncompress multiple files
hmmer	617.2 KB	Search sequence databases for homologs of protein sequence

The difference pattern of indirect branch transfers is the sequence started with a source address 0x12bd56 and ended with a source address 0x13a278 or 0x13a288. It can be described as the FSA in Fig. 5(a), where 0 is the started state, state 2 is the accepted state, '||' represents *or* operation.

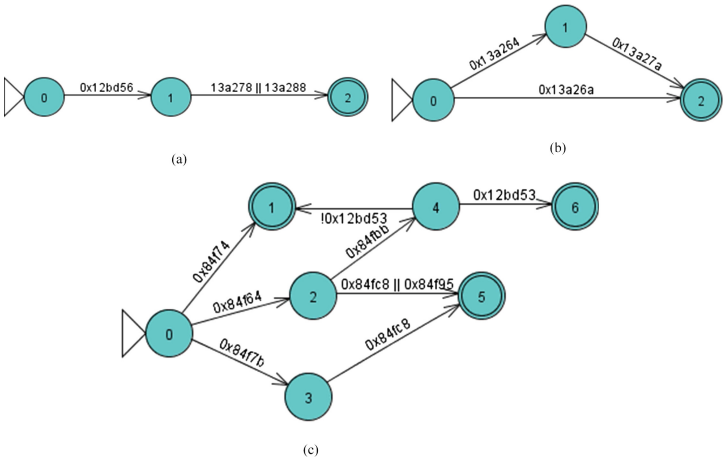

Fig. 5. Difference patterns produced by libc.so

The difference patterns of direct branch transfers in libc.so are shown as Fig. 5(b) and (c). In Fig. 5(b), all addresses are source addresses. In Fig. 5(c) all addresses are source addresses except address 0x12bd53. When the source address of a branch is 0x84f64, and the source address of the next branch is 0x84fbb, there is a need to decide whether the target address of another branch is 0x12bd53, if it is, then the difference patterns is $<0x84f64, 0x84ffb>$, otherwise, the difference pattern is $<0x84f74>$.

Same input but different length of command-line. Although the input is the same, the different length of the input file name will have an impact on branch transfer too. Since the program itself has variability, this section also discusses the difference patterns produced by shared library libc.so. The difference patterns of indirect branch transfers are shown in Fig. 6(a), (b). We can find the differences are mainly the return address of `__i686_get_pc_thunk_bx` and `__i686_get_pc_thunk_dx`. The difference patterns of direct branch transfers are shown in Fig. 6(c), (d), (e) and (f), & means *and* operation.

Linux signals. The difference pattern caused by Linux signals is described as Fig. 7. The procedure of the OS to execute the signal handler will produce branch transfers between targets 0x414 and source address 0x427.

Procedure of searching for shared libraries. When there are other shared libraries installed, the procedure of searching for shared libraries will have an impact on branch transfers, and these difference patterns can be described as Fig. 8 and all addresses are source addresses.

In order to accelerate the speed of pattern matching, for these FSAs, we allocate a 1-bit flag for every state except for the accepted state to distinguish the source address and target address. If flag equals 0, it is the source address, otherwise, it is the target address.

As we can see, different environmental factors will generate different FSAs, and space has the most significant impact compared with other factors. For most difference patterns caused by space, its feature is that they just go through from the initial state to the accepted state straightly, and the difference pattern caused by signal, when it arrives the state before the accepted state, it may goes back to the former state, this is because there may be more than one signal at a time.

4.3 Getting Rid of Environment Impact

In order to get rid of the impact of environment on branch transfer, we allocate a stack for every state except for the initial state. Each stack records the row number of the branch transfers before meeting the accepted state. As shown in Fig. 9, the left is the FSA, the right is the difference pattern.

Assume the row numbers of branch transfers are shown in Fig. 9. The contents of stack 1, 2, 3 are (12513, 12514, 12515, 12516, 12517), (12518, 12519) and (12520) respectively. When it meets the accepted state, outputs these contents to a file until all branch transfers have been recorded, then we can delete these branch transfers according to the row number.

The time overhead of getting rid of environment impact depends on the number of states and state transitions a FSA have. The space overhead will be $O(M * N)$, where M is the number of states in the FSA and N is the average number of transitions for every state. If we use `Bloom filter`, the time overhead can be a constant.

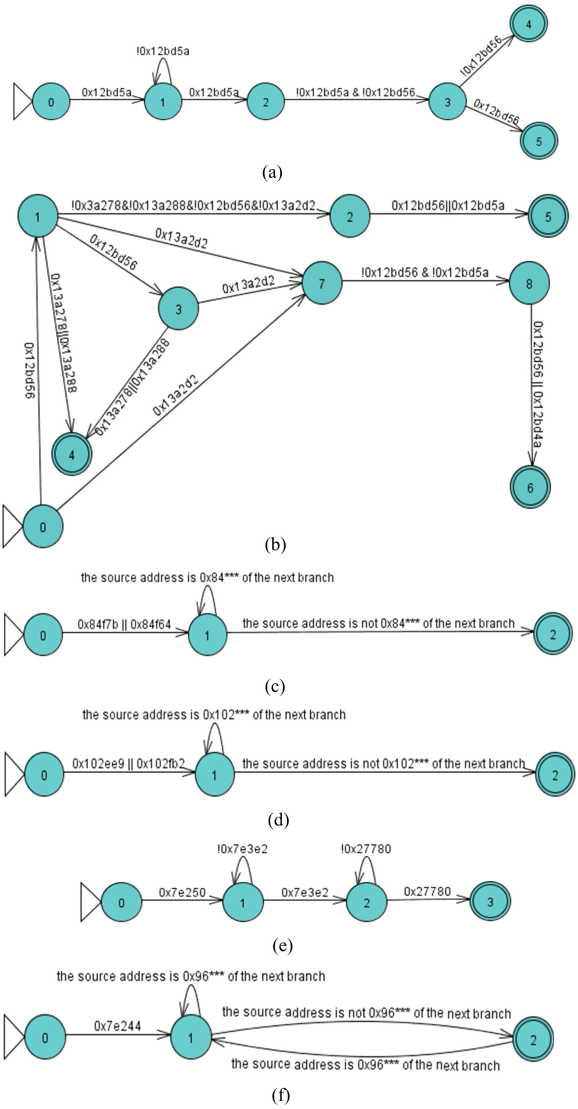

Fig. 6. Difference patterns caused by the different length of command-line

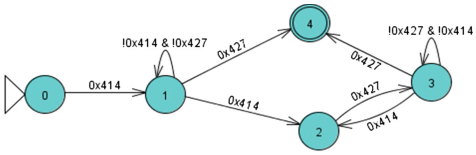

Fig. 7. Difference pattern caused by Linux signals

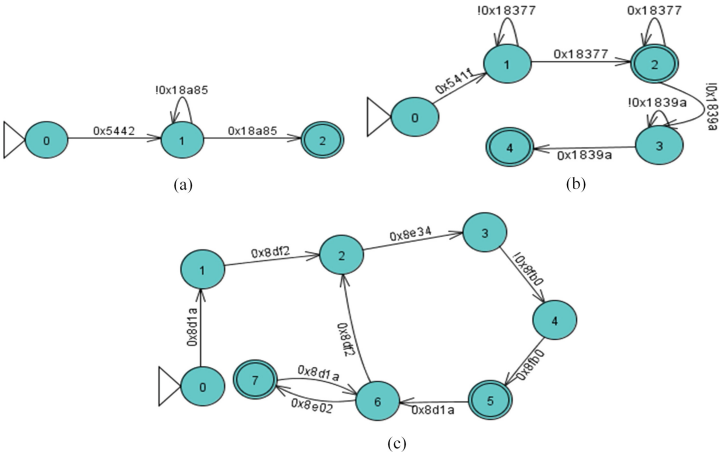

Fig. 8. Difference patterns caused by the procedure of searching shared libraries

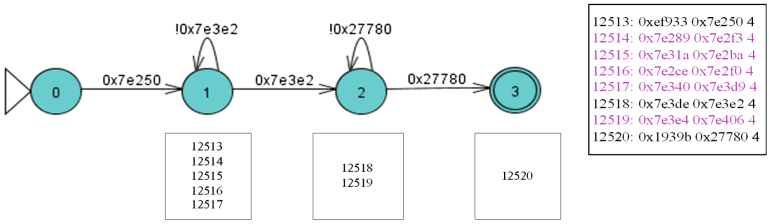

Fig. 9. FSA-Stack model

5 Experimental Result

In this section, we will introduce how environment impacts on branch transfers through our experiments.

5.1 Comparing the Number of Branches

Same Input. For programs in Table 3, we have run them 50 times to get the difference between branch numbers. We have discovered that except tesseract and graohicsmagic, the difference of branch number in other programs is very small, which is no more than 3. The difference of branch number in tesseract is no more than 300, and in grahicsmagic is less than 80. This is because space has a greater impact on tesseract and grahicsmagic, which will be discussed in Sect. 5.3.

Same Input but Different Length of Input File Name. For programs in Table 3, we change the length of the input file's name (close ASLR) and then observe the difference of the branch number. We can find that for most programs,

the difference of branch number is mainly located in direct branches produced by shared libraries. The number of direct branches of program itself will increase as the length of the input file's name increases in gzip and graphicsmagic. And we can get that the number of indirect branches for program itself is always the same.

Different Inputs. For programs gzip, bzip2, grep, and tesseract, we change their input (types of input file include doc, pdf, txt, ppt, and tif) to observe their branch number. We get that the difference of branch number is great when inputs are different. These differences are mainly located in direct branch transfers of the program itself.

5.2 Impact of Time

We have tested the impact of time on branch transfer for programs in Table 3. As described in Sect. 3, the impact of time on branch transfer is focused on branch transfer $<0x00004e80, 0x00004e86>$. Table 4 records the number this branch occurs, we can find this branch occurs no more than four times. There are 4 occurrences of this branch for 38 out of 50 runs in gzip. And it is related to the CPU, the faster the CPU runs, the more this branch transfer number is.

Table 4. The number of branch transfer $<0x00004e80, 0x00004e86>$

Application	Number				Total
	4	3	2	1	
gzip	38	8	3	1	50
grep	39	10	1	0	50
bzip2	37	9	4	0	50
bubblesort	38	10	2	0	50
cat	38	8	2	2	50
diff	38	6	5	1	50
tesseract	33	13	4	0	50
graphicsmagic	41	8	1	0	50
bunzip2	38	7	5	0	50
hmmer	38	10	2	0	50

5.3 Impact of Space

User Input. For programs in Table 3, we find that as the input is the same, only the control flow of gzip, tesseract, and graphicsmagic are different(excluding the impact of time), and these differences are located in shared library. Functions that these differences are located in are shown in Table 5.

Table 5. Impact of input on software behavior

Application	Function name	Library
gzip	`strlen_sse2_bsf`	libc.so
tesseract	`strrchr_ia32`	
	`strlen_sse2_bsf`	
	`memcpy_sse3_rep`	
graphicsmagic	`strlen_sse2_bsf`	
	`strrchr_sse2_bsf`	
	`memcpy_sse3_rep`	

As shown, when the input is the same, the differences of branch transfers are located in functions related to string operation in shared library libc.so. For string handling functions, in order to accelerate the processing speed, the address of the string will be checked to determine whether it is 4-byte boundary alignment or 16-byte boundary alignment (Streaming SIMD Extension (SSE) instruction). These differences cover direct call branches, direct jump branches, ret branches, and indirect jump branches.

From experimental result, we find that these differences are caused by the ALSR for stack, which leads to the temporary variables' 16-byte boundary alignment change. In order to determine the impact of ASLR on software behavior, we close ASLR mechanism in Linux, and run programs in Table 5, get that there is no difference on branch transfer for all programs.

Meanwhile, we change the directory of the input file to observe the change of the control flow. The experimental results show that when the length of the input file's directory is not changed, the control flow still is not different. But if the length of the input file has been changed, the control flow will produce great differences. This is because in Linux many operations are related to the input, such as getting the length of the command-line, copying parameters of the command line, and getting the name of the program.

Environment Variables. We change the length of the environment variables (close ASLR) to observe its impact on branch transfer. We find that these functions almost are the same with Table 5, and the reason is the same as user input.

5.4 Impact of Signal

In order to determine the impact of signals on software behavior, we add a signal SIGINT into the program bubblesort. When it receives `^C`, the signal will be triggered. We have found that when there is a signal, the number of branch transfers will be more than without signal. The pattern is $(0x414, ..., 0x427)$, the control flow will be transferred to the signal handler, and then return to 0x427 to continue the normal control flow. Due to the signal handler is located in user space, so we can record its branch transfers using Pin.

5.5 Impact of Searching for Shared Libraries

When we install r-base on the testing system, it will install other shared libraries into the system, we compare the contents of /etc/ld.so.cache, and find its contents are modified, the difference of the content is shown in Fig. 10.

```
101a102
> libpcrecpp.so.0 -> libpcrecpp.so.0.0.0
123a125
> libgfortran.so.3 -> libgfortran.so.3.0.0
195a198
> libodbc.so.1 -> libodbc.so.1.0.0
226a230
> libodbcinst.so.1 -> libodbcinst.so.1.0.0
......
```

Fig. 10. Difference of /etc/ld.so.cache

In Fig. 10, many other shared libraries are written into /etc/ld.so.cache. For example, 101a102 represents that now there is a new symbolic linking libpcrecpp.so.0 → libpcrecpp.so.0.0.0 in the file, and 'a' means *add*.

6 Discussion and Conclusion

Software behavior is affected by environmental factors, especially for branch transfers. In this paper, we study the impact of environment on branch transfer of software from time, space (memory boundary alignment) and mechanisms of the Linux operating system (the procedure of searching for shared libraries, signals, GOT/PLT, and the configuration of the environment variable `LD_PRELOAD`). At the same time, we leverage Finite State Automata (FSA) to describe the difference patterns of branch transfers caused by environmental factors. These difference patterns can be used to control flow integrity detection that the testing and validation code is independent on the program to mitigate the false alarms. Meanwhile they can be used to CIMB [28] to reduce the impact of environmental factors on computation integrity measurement.

In future work, we will focus on attacks caused by these environmental factors as these factors are ignored by CFI.

Also, there are some limitations in our work. For instance, we just investigate environmental factors from time, space and mechanisms in Linux, there may be some other factors, such as compiler optimization and the upgrade of operating systems. And we do not investigate the impact of environment on the branch transfer of kernel code.

Acknowledgements. Supported by the National Natural Science Foundation of China (61373168), and Doctoral Fund of Ministry of Education of China (20120141110002).

References

1. Forrest, S., Hofmeyr, S., Somayaji, A.: The evolution of system-call monitoring. In: Proceeding of the 24th Annual Computer Security Applications Conference, California, USA, pp. 418–430 (2008)
2. Wee, K., Moon, B.: Automatic generation of finite state automata for detecting intrusions using system call sequences. In: Gorodetsky, V., Popyack, L., Skormin, V. (eds.) MMM-ACNS 2003. LNCS, vol. 2776, pp. 206–216. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45215-7_17
3. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: Snekkenes, E., Gollmann, D. (eds.) ESORICS 2003. LNCS, vol. 2808, pp. 326–343. Springer, Heidelberg (2003). doi:10.1007/978-3-540-39650-5_19
4. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Trans. Dependable Secure Comput. **7**(4), 381–395 (2010)
5. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceeding of the 14th ACM Conference on Computer and Communications Security, Alexandria, USA, pp. 552–561. ACM (2007)
6. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceeding of the 6th ACM Symposium on Information, Computer and Communications Security, Hong Kong, China, pp. 30–40. ACM (2011)
7. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceeding of the 12th ACM Conference on Computer and Communications Security, Alexandria, USA, pp. 340–353. ACM (2005)
8. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. J. ACM Trans. Inf. Syst. Secur. **13**(1), 1–41 (2009)
9. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: detecting violation of control flow integrity using performance counters. In: IEEE/IFIP International Conference on Dependable Systems and Networks, Boston, USA, pp. 1–12. IEEE/IFTP (2012)
10. Yuan, P., Zeng, Q., Ding, X.: Hardware-assisted fine-grained code-reuse attack detection. In: Bos, H., Monrose, F., Blanc, G. (eds.) RAID 2015. LNCS, vol. 9404, pp. 66–85. Springer, Cham (2015). doi:10.1007/978-3-319-26362-5_4
11. Cheng, Y., Zhou, Z., Yu, M., Ding, X., Deng, R.H.: ROPecker: a generic and practical approach for defending against ROP attacks. In: Proceeding of Symposium on Network and Distributed System Security, San Diego, USA. ISOC (2014)
12. Intel Manual. Intel 64 and IA-32 architecture software developers manual, vol. 3
13. Zhong, S.: Certified software. Commun. ACM **53**(12), 56–66 (2010)
14. Murali A, Rao M. A survey on intrusion detection approaches. In: Proceeding of the First International Conference on Information and Communication Technologies, Karachi, Pakistan, pp. 233–240. IEEE (2005)
15. Garcia-Teodoro, P., Diaz-Verdejo, J., Maci-Fernndez, G., Vazque, Z.: Anomaly-based network intrusion detection: techniques, systems and challenges. Comput. Secur. **28**(1), 18–28 (2009). Elsevier
16. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweenry, P.F.: Producing wrong data without doing anything obviously wrong! ACM Sigplan Not. **44**(3), 265–276 (2009)
17. Yeung, D.Y., Ding, Y.: Host-based intrusion detection using dynamic and static behavioral models. Pattern Recogn. **36**(1), 229–243 (2003). Elsevier

18. Li, P., Park, H., Gao, D., Fu, J.: Bridging the gap between data-flow and control-flow analysis for anomaly detection. In: Proceeding of the 24th Annual Computer Security Application Conference, California, USA. IEEE (2008)
19. Giffin, J.T., Dagon, D., Jha, S., Lee, W., Miller, B.P.: Environment-sensitive intrusion detection. In: Proceeding of Recent Advances in Intrusion Detection, Hamburg, Germany, pp. 185–206. Springer (2006)
20. Zhang, C., Wei, T., Chen, Z,, Duan, L.: Practical control flow integrity and randomization for binary executables. In: Proceeding of IEEE Symposium on Security and Privacy, San Francisco, USA, pp. 559–573. IEEE (2013)
21. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: Proceeding of the 22nd USENIX Security Symposium, pp. 337–352. IEEE, Washington, D.C. (2013)
22. Wang, M., Yin, H., Bhaskar, A.V., Continent, B.C., et al.: Finer-grained control flow integrity for stripped binaries. In: Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, USA, pp. 331–340. IEEE (2015)
23. Lin, Y., Tang, X., Gao, D., Fu, J.: Control flow integrity enforcement with dynamic code optimization. In: Bishop, M., Nascimento, A.C.A. (eds.) ISC 2016. LNCS, vol. 9866, pp. 366–385. Springer, Cham (2016). doi:10.1007/978-3-319-45871-7_22
24. Pappas, V.: kBouncer: efficient and transparent ROP mitigation. Technical report, Columbia University (2012)
25. Zhao, T., Tang, Y., Xu, W., Fu, G., Qi, S., Jia, X., et al.: Exactly reproducible program execution and its application in computer architecture simulation. Chin. J. Comput. **34**(11), 2073–2083 (2011)
26. Shacham, H., Page, M., Pfaff, B.: On the effectiveness of address-space randomization. In: Proceeding of the 11th ACM Conference on Computer and Communications Security 2004, pp. 298–307 (2004)
27. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., et al.: Pin: building customized program analysis tools with dynamic instrumentation. ACM Sigplan Not. **40**(6), 190–200 (2005)
28. Fu, J., Lin, Y., Zhang, X., Li, P.: Computation integrity measurement based on branch transfer. In: Proceeding of the 13th International Conference on Trust, Security and Privacy in Computing and Communications, Beijing, China, pp. 590–597. IEEE (2014)