Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

A Runtime Code Modification Method for Application Programs

Kazuhiro Yamato
Miracle Linux Corporation
kyamato@miraclelinux.com

Toyo Abe
Miracle Linux Corpration
tabe@miraclelinux.com

Abstract

This paper proposes a runtime code modification method for application programs and an implementation. It enables the bugs and security problems to be fixed at runtime. Such software is notably useful for applications used in telecom, which cannot be stopped because of the need to maintain the required level of system availability. The advantages of the proposed method are short interruption of the target application and easy maintenance using trap instructions and utrace.

This paper also shows evaluation results with three conditions. The interruption times by the proposed method were comparable to, or shorter than those by existing similar software, *livepatch* and *pannus*. In a certain condition, our implementation's interruption time was three orders of magnitude shorter in comparison.

1 Introduction

Although there have been a number of activities to improve software quality, there is no way to completely prevent bugs and security problems. These are generally fixed by rebuilding the program with patches to the source code. This fix naturally requires termination and restarting of the program. The termination of the program not only interrupts the service, but also loses various data such as variables, file descriptors, and network connections. It is impossible to recover these properties unless a recovery mechanism is built in the program itself.

Fixing with a termination is a serious problem especially for application programs used in telecom, because they cannot easily be terminated to keep the required level of system availability.[1] Therefore, the problems should be fixed at runtime with binary patches. In addition, interruption time to apply binary patches should be short because long interruption degrades the quality of voice and video, which are major services of telecom.

In this paper, we call an application program to be fixed by RBP (Runtime Binary Patcher) a *target*. Two major open source RBPs, *livepatch* [1] and *pannus* [2], already exist. However, *livepatch* potentially interrupts the execution of a target for a long time. The maintenance of *pannus* doesn't seem to be easy.

This paper proposes a runtime code modification method for application programs, and, an implemented RBP. It achieves short interruption and easy maintenance using the trap instruction and the utrace APIs [3].

2 Existing Methods

2.1 `ptrace` system call and *gdb*

The `ptrace` system call provides debug functions, such as reading/writing memory in the target's virtual memory space, acquisition/modification of the target's registers, and catching signals delivered to the target. These functions are enough to realize runtime code modification. Actually, we can modify a target's memory with *gdb* [4], which is one of the most popular debuggers in the GNU/Linux environment and also a typical application using `ptrace`.

For example, the *gdb* command `"set variable *((unsigned short*)0x8048387)=0x0dff"` overwrites `0x0dff` (`dec` instruction on i386) at address `0x8048387` by calling `ptrace(PTRACE_POKEDATA, pid, addr, data)`, where `pid`, `addr`, and `data` are the process ID of the target, the address to be overwritten, and the address of data to overwrite, respectively.

However, this approach potentially causes long interruption of target execution when the target has many

[1] The CGL (Carrier Grade Linux) specification requires 99.999% availability.

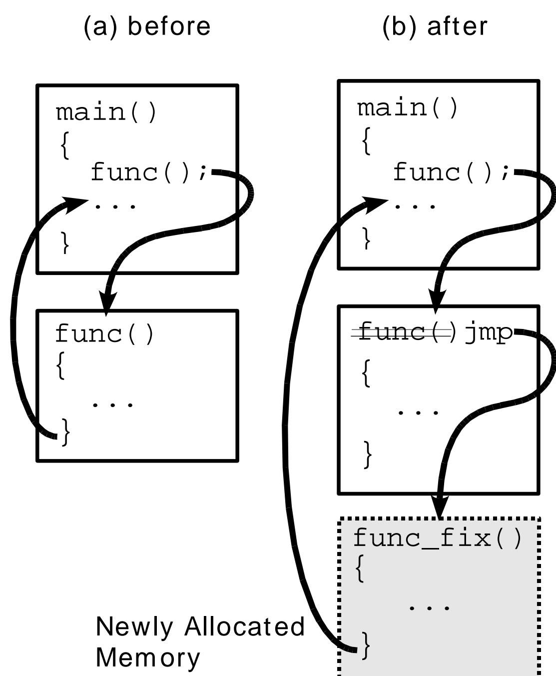

Figure 1: Function call path before and after applying a binary patch

threads or the patch is large. *gdb* frequently stops all threads in a target. As the number of threads increases, interruption time increases, too. The writing size of PTRACE_POKEDATA is a 'word,' which is architecture-dependent—four bytes on i386.

Some practical cases will require additional memory to apply a binary patch whose size is greater than the original code. ptrace doesn't provide a direct function to allocate memory. However, *livepatch* solves this by making a target execute instructions to allocate memory as described in Section 2.2.1.

2.2 Open Source RBPs

livepatch and *pannus* are two major open source RBPs. They both fix problems by adding a binary patch in the target's virtual memory space and overwriting the jmp instruction to the binary patch at the top of the function to be fixed (we call it the target function) as shown in Figure 1. This means problems are fixed by the function unit. Thus a patch is provided as a function (fixed-function) in an ELF shared library (patch file). The basic processes of *livepatch* and *pannus* are similar and roughly divided into the following four stages.

1. Preparation: opens a patch file, obtains the size of the patch, gives addresses to unresolved symbols, and so on.

2. Memory Allocation: allocates memory for the patch in the target.

3. Load: loads the patch into the allocated memory.

4. Activation: overwrites an instruction to jump to the patch at the top of the target function.

```
int prot = PROT_READ|PROT_WRITE;
int flags = MAP_PRIVATE|MAP_ANONYMOUS;
mmap(NULL, size, prot, flags, -1, 0);
asm volatile("int $3");
```

Figure 2: The code written in the target stack

2.2.1 livepatch

livepatch [1] was developed by F. Ukai, and consists only of a utility in user space, which is about 900 lines of code. In the preparation stage, *livepatch* gets the information about the patches in the ELF file with libbfd. In the memory allocation stage, *livepatch* first obtains the stack pointer of the target using PTRACE_GETREGS. Then it writes the machine code corresponding to the source shown in Figure 2 on the stack. The code is executed by setting the program counter to the top address of the stack using PTRACE_SETREGS, followed by PTRACE_CONT. The mmap() in the code allocates memory in the target, because the target itself calls the mmap(). The assembler instruction 'int $3' generates the SIGTRAP to bring back the control to *livepatch*, which is sleeping by wait(NULL) after the PTRACE_CONT.

In the load stage, *livepatch* writes the patch in the allocated memory by repeating PTRACE_POKEDATA. In the activation stage, PTRACE_POKEDATA is also used to overwrite the instruction to jump.

2.2.2 pannus

pannus [2] was developed by a group from NTT Corporation. It consists of a utility in user space and a kernel patch. In the preparation stage, *livepatch* analyzes a

patch file by itself without external libraries and obtains necessary data. The memory allocation is mainly performed by a `mmap3()` kernel API which is provided by the kernel patch. Actually it also plays a role in a portion of the load stage, because the `mmap3()` maps the patch file. The `mmap3()` is an enhanced version of `mmap2()`, which is a standard kernel API. It enables other processes to allocate memory for the specified process, directly accessing the core kernel structures such as `mm_struct`, `vm_area_struct`, and so on. Because members of the structures or access rules are often changed, the maintenance of the patch doesn't seem to be easy.

In the load stage, the `access_process_vm()` kernel API is used via the kernel patch to set relocation information in the allocated memory. The API reads/writes any size of memory in the specified process.

In the activation stage, *pannus* also uses `access_process_vm()` to overwrite the instruction to jump. Note that *pannus* checks whether the status is safe before the overwriting. The safety means that the number of threads whose program counters point the region to be overwritten is zero. The program counter is obtained by `PTRACE_GETREGS`, after the target was stopped by `PTRACE_ATTATCH`. If the status is not 'safety,' *pannus* resumes the target once by `PTRACE_DETATCH`, and tries to check again. If the result of the second check is not also safety, *pannus* aborts the overwriting. The probability that this situation happens increases with the call frequency of the target function.

3 Proposed Method and its Implementation

3.1 Patching process

We propose a new patching method, which is used in our project *kaho*, which means *Kernel Aided Hexadecimal code Operator*. The patching process of *kaho* is divided into four stages similarly to *livepatch* and *pannus*. They are shown in Figure 3 with detailed steps. Its implementation consists of a user-space utility program and a kernel patch. The white boxes in the figure are processing by the utility. The shaded steps are processed in the kernel via IOCTLs. The supported architectures of *kaho* are x86_64 and i386 at this moment. All these steps are described here and details of IOCTLs and 'Safety check' are explained in the following sections.

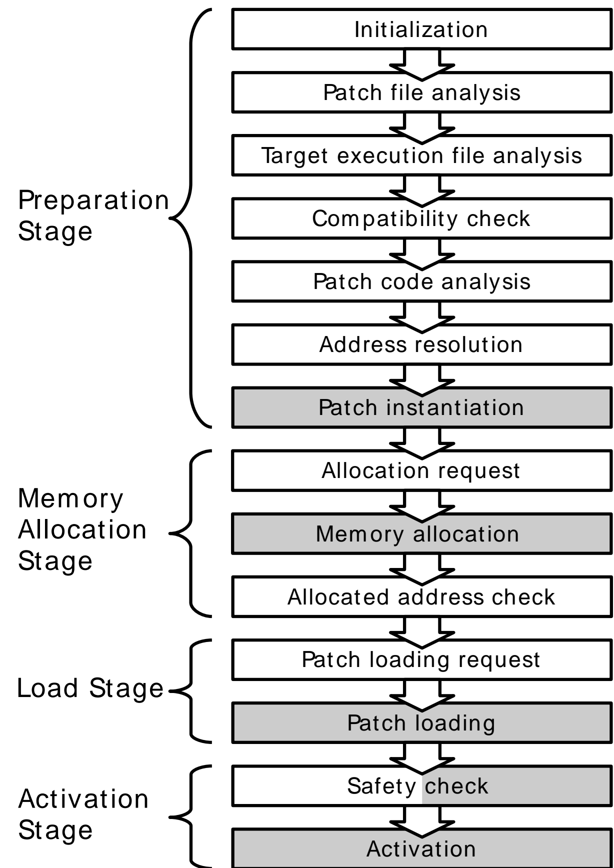

Figure 3: The patching process of *kaho*

The *Preparation Stage* consists of seven steps. *Initialization* interprets the command line options, which include information about the target and the command file. The command file defines the name of a target, a fixed function, a patch file, and a map file. *Patch file analysis* opens the patch file and reads ELF information such as ELF header, section headers, program headers, symbol table, dynamic sections, and versions. *Target execution file analysis* finds the executable from `/proc/<PID>/exe` and acquires ELF information. *Compatibility check* confirms that byte order (endian), file class, OS ABI,[2] and ABI version of the patch file, which are contained in the ELF header, are identical to those of the target execution file. *Patch code analysis* searches for an entry whose `st_name` member is identical to the name of the fixed function from the symbol table, and its file position from the `st_value`. *Address resolution* works out the addresses of unresolved symbols using a target executable, the depending libraries, and `/proc/<PID>/maps`. When symbols

[2]such as UNIX – System V, UNIX – HP-UX, UNIX – NetBSD, GNU/Hurd, UNIX – Solaris, *etc.*

are stripped out in the executable and libraries, the map file must be specified in the command file, because the map file lists function names and the corresponding addresses. *Patch instantiation* allocates the data structure to manage binary patches in the kernel space, and generates a unique handle for every a patch instance.

The *Memory Allocation Stage* consists of three steps. *Allocation request* finds a vacant-address range near the target function using `/proc/<PID>/maps` and calls the *Memory allocation* IOCTL. *Allocated address check* confirms that the address[3] is within a $\pm$2GB range from the target function. Note that *Allocated address check* is needed for the x86_64 architecture only, because the immediate operand of the `jmp` instruction is relative-32-bit despite having a 64-bit accessible memory space.

Load Stage consists of *Patch loading request* and *Patch loading*. *Patch loading request* calls an IOCTL with a patch instance handle, the fixed function's address in *kaho* utility virtual memory, size of the fixed function, and the target function's virtual memory address. *Patch loading* is an IOCTL to load the fixed function in the target.

Activation Stage consists of *Safety check* and *Activation*. *Safety check* confirms that no threads are executing code on the region to be overwritten by *Activation*. If this is not checked, threads may fetch illegal instructions. There are two modes to check safety, standard mode and advanced mode. In the standard mode, the check is performed by the *kaho* utility program. In the advanced mode, *Activation* performs it. *Activation* overwrites the instruction to jump to the fixed function at the top of the target function.

In fact, *kaho* can deactivate and remove the activated patches. In addition, *kaho* can modify the data in the target. In the case, *Loading Stage* stores the data from the utility in the kernel. *Activation Stage* overwrites the data with `access_process_vm()`.

3.2 Patch instantiation

The *Patch instantiation* IOCTL receives the process ID of the target and the number of patches. It first takes an available handle from its own handle pool and creates the data structure to manage patches, which is named *patch instance*. Patch instance's member variables include the number of patches, the pointer to the target's `task_struct`, addresses of the target functions, sizes of the patches, and so on. Then it gets the pointer to target's `task_struct` with `find_task_by_pid()`, adds the patch instance to the dedicated list named patch-instance list, and attaches the target by calling `utrace_attach()` with the callbacks shown in Figure 4. After the target is attached, `utrace_set_flags()` with flag `UTRACE_EVENT(EXEC)|UTRACE_EVENT(REAP)`[4] is called to delete the patch instance from the patch-instance list and release it for the case in which the patch becomes no longer needed. Finally, *Patch instantiation* returns the handle to be used in the other IOCTLs.

```
static const struct
utrace_engine_ops kaho_utrace_ops =
{
  .report_exec = kaho_report_exec,
  .report_quiesce = kaho_report_quiesce,
  .report_reap = kaho_report_reap,
};
```

Figure 4: *kaho*'s `utrace` callbacks

3.3 Memory Allocation

Memory Allocation IOCTL receives a handle, a request address, and a request size. It first finds that the patch instance which has the handle is in the patch-instance list and stores the request address and the request size in the patch instance. Then it enables the callback `kaho_report_quiesce()` by calling `utrace_set_flags()` with flags `UTRACE_ACTION_QUIESCE | UTRACE_EVENT(QUIESCE)`. Shortly after the flags are set, `utrace` executes the callback specified in `.report_quiesce` (that is `kaho_report_quiesce()` in this case) on the target context as shown in Figure 5.

`kaho_report_quiesce()` calls `do_mmap()` with the requested address and size on the target context. As a result, memory is allocated in the target's virtual memory space. The basic idea is similar to

[3] The address in which the fixed-function is loaded in a precise sense.

[4] The flag enables callbacks specified in `.report_exec` and `.report_reap` to be called when the target calls the `exec()` family and when the target terminates, respectively.

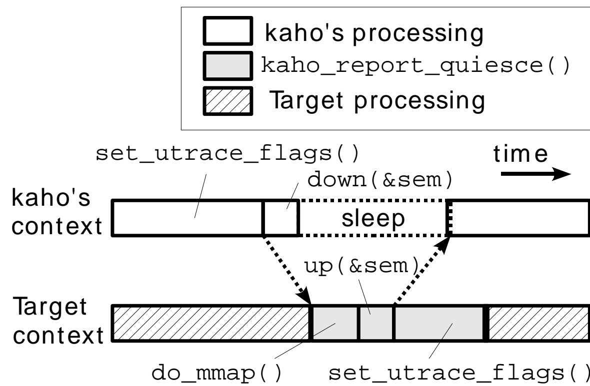

Figure 5: Mechanism of memory allocation

that of *livepatch*. Next `kaho_report_quiesce()` executes `up(&sem)` to wake up the *kaho* utility program which is sleeping by `down(&sem)`. Finally, `kaho_report_quiesce()` stores the address of the allocated memory in the patch instance and calls `utrace_set_flags()` without flags `UTRACE_ACTION_QUIESCE|UTRACE_EVENT(QUIESCE)` to disable this callback and resume the target's processing. After the sleep finishes, the address is returned.

3.4 Patch Loading

Patch Loading IOCTL receives the handle, the address at which the fixed function is in the *kaho*'s memory space, the address to be loaded in the target's memory space, the address of the target function in the target's memory space, the size to be loaded, and the subpatch ID. The sub patch ID is the sequential number to identify patches which are applied at one time. After *Patch Loading* stores them in the patch instance, it loads the fixed functions into the target by calling `access_process_vm()`.

`access_process_vm()` is the standard kernel API which reads or writes the memory in the specified process. It is also used from the `ptrace` system call and some kernel functions to handle `/proc/<PID>/mem`. This means that memory in the target can be written by calling `ptrace` and writing `/proc/<PID>/mem`. However, This works only when the target is in a traced state; namely, the target must be stopped. Although this is necessary to prevent unexpected results in usual cases, we don't access memory loading fixed functions. Therefore, *Patch Loading* calls `access_process_vm()` without stopping the target.

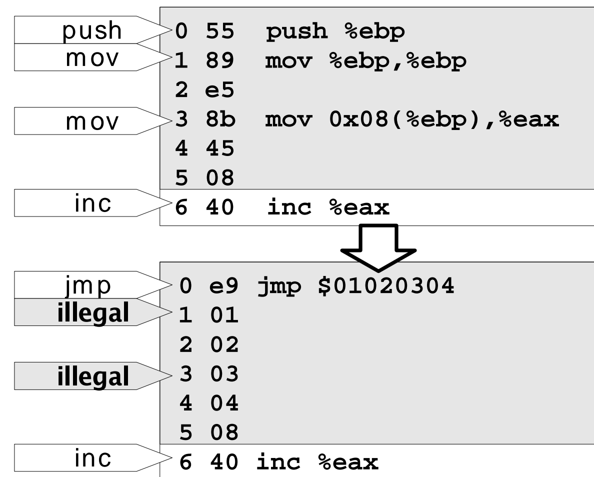

Figure 6: The example of the safety and danger

3.5 Safety check

Safety check confirms that program counters of the all threads in the target don't point the region to be fixed. If a thread's program counter points to such a region,[5] it gets illegal instructions after the instruction to jump is overwritten, as shown in Figure 6. *kaho* has two modes to check the safety. One is the standard mode in which the *kaho* utility program checks. The other is the advanced mode in which the *Activation* IOCTL checks before the instruction to jump to the fixed function is overwritten in kernel space. The interruption of the target in the advanced mode is shorter than that in the standard mode, because the target is not stopped in the advanced mode. However, the number of fixed functions which are applied at one time in the advanced mode is limited to only one.

3.5.1 Standard mode

Safety check in the standard mode consists of *Quick check* and *Forced displacement*. *Quick check* attaches the target with `ptrace` and checks the value of the program counter with `PTRACE_GETREGS`. When the program counters of all threads do not point the region to be overwritten, the check successfully finishes. Otherwise, the check is retried. If the failures continue a few times, *Forced displacement* is executed as the threads are attached.

[5]The first byte of the region is exempted, because some sort of valid instruction should be overwritten.

Forced displacement tries to bring about a safety state using the trap instruction (`int 3`). It first overwrites a trap instruction at the top of the target function. Then it resumes only the threads whose program counters point the region to be fixed with `PTRACE_CONT`. After that, some threads will stop by the trap instruction. Consequently such threads become safe. Other threads including threads which do not run on the code with trap instruction are checked periodically with `PTRACE_ATTACH` and `PTRACE_GETREGS`. If safety of all threads is confirmed, the check successfully finishes. Otherwise, it fails.

3.5.2 Advanced mode

Safety check in the advanced mode is further broken down into four steps. These consist of *Preparation*, *Trap handler*, *Thread safety check*, and *Target safety check*. The basic strategy is to mark the thread which has executed the overwritten trap instruction at top of the target function for safety. This method is inspired by `djprobes` [5]. *Preparation* first makes a checklist in which pointers to the task struct of all threads and corresponding check statusus are contained and adds the pointer of the target's task struct to the list called *trapped-targets*. Then it overwrites the trap instruction at top of the target function. After this, the check in the *Trap handler* becomes active as described below. Finally, it calls `utrace_attach()` and `utrace_set_flags()` to execute the callback specified in `.report_quiesce` for all threads, which is *Thread safety check* in this case.

Trap handler is registered by `register_die_notifier()` when a system is initialized. It is called when any process or a thread in the system executes a trap instruction. Therefore it must confirmed whether the thread which executes the trap instruction and the address are included in the trapped-targets list. If it is included, *Trap handler* marks the thread as safety. Then it changes the program counter of the thread to the address of the fixed function by setting `rip` or `eip` in `struct pt_regs` given by the lower-common layer of kernel, as shown in Figure 7. This means that the thread executes the fixed function.

Thread safety check is called on the context of the target thread. It first reads the check list and confirms that the safety of the thread was already checked by *Trap handler*. If the thread is 'safety,' it finishes immediately.

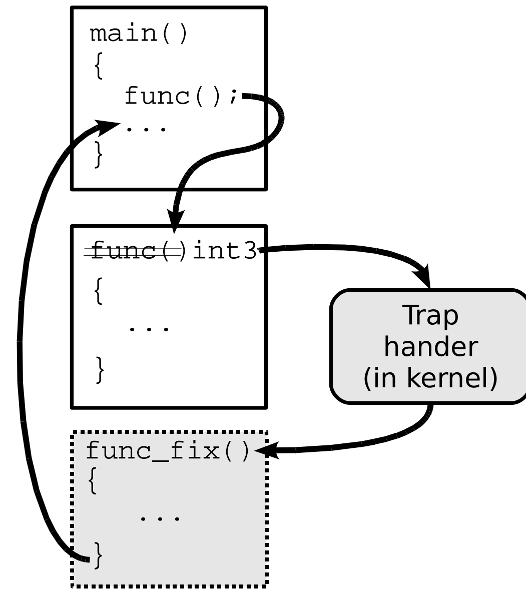

Figure 7: Function call path during safety check in advanced mode

Otherwise, it checks for safety, reading program counters in user space, which is saved in the kernel mode stack. The result is written in the check list. After the *Thread safety check* is executed on the all threads, *Target safety check* reads the check list and deletes safe threads from the list. When the check list becomes empty, *Target safety check* successfully finishes. Otherwise, it repeats *Thread safety check* for the remaining threads in the check list.

3.6 Activation

Activation IOCTL receives the handle and the flag. It overwrites instructions to jump to the fixed function at the top of the target function using `access_process_vm()`. The addresses of the fixed function, address of the target function, and the sizes of the fixed functions are obtained from the patch instance. The examples of the instructions to be overwritten are shown in Figure 8 and Figure 9. In Figure 8, the `jmp` instruction which takes a 32-bit relative address is used and its total size is 5 bytes. The instructions shown in Figure 9 are used only when the architecture is x86_64 and `KAHO_X86_64_ABS_JMP` is set in the flag. The condition in which the instructions in Figure 9 is needed is rare. Code, data, and stack regions in the memory space are usually sparsely placed in memory.

```
        jmpq  $0x12345678
```

Figure 8: Example of instruction to be written

```
    pushq $0x12345678
    movl  $0x9abcdef0, 4(%rsp)
    ret
```

Figure 9: Example of instructions to be written when the flag `X86_64_ABS_JUMP` is specified on the 86_64 architecture

3.7 Usage of *kaho* utility program

We introduce brief usage for the *kaho* utility program named `kaho`. The usage is greatly influenced from that of *pannus*. First of all, environment variable `KAHO_HOME` must be set. It specifies the base directory in which the command file, the patch file, and the map file are placed. Then users perform the following three steps.

1. Prepare a command file, a patch file, and a map file.
2. Execute `kaho` with the load option.
3. Execute `kaho` with the activation option.

3.8 Limitations

3.8.1 Handling of C++ exception

kaho cannot apply a binary patch to code which can potentially generate C++ exceptions. When the exception happens, the code compiled by *g++* tries to find the frame to be caught with the `.eh_frame_section` in the ELF file which contains the target function. However, the information about the fixed function is not in the `.eh_frame_section`. As a result, the exception is not caught correctly.

3.8.2 Static variables

When a target function contains static variables and a patch uses them, the patch may not work correctly. The reason is that the patch refers not to the variable in the target, but in itself. This can be avoided by replacing the `static` keyword with the `extern` keyword and adding the address of the static variable in the target function to the map file.

3.8.3 Loading new shared libraries

kaho fails to load patches when the fixed functions in the ELF file require additional functions which are not in the ELF file itself, the target executable file, or already-loaded shared libraries.

3.8.4 Multiple binary patch in advanced mode

The present algorithm of the safety check in the advanced mode allows only one fixed function to be applied at a time. However, there is a certain situation in which multiple fixed functions should be applied at one time. Therefore, we plan to extend the number of fixed functions which are applied at one time in advanced mode.

4 Evaluation

4.1 Evaluation method

We evaluated performance of RBPs (*livepatch*, *pannus*, and *kaho*) by the interruption time in target execution. This section describes our definition of the interruption time, our measuring method, the target programs used for the measurement, and our evaluation environment.

4.1.1 Interruption time

We defined five categories of interruption; *Allocation*, *Load*, *Check*, *Trap*, and *Setup*. *Allocation* is the interruption due to memory allocation for a binary patch. All RBPs but *pannus* allocate memory in the target context. *pannus* does it from the outside of the target via `mmap3()`. *Load* is the interruption due to loading the binary patch. Only *livepatch* does it in the target context. *Check* is the interruption due to the safety check for the target to be safely patched. *pannus*, *kaho* in the standard mode (*kaho-std*), and *kaho* in the advanced

mode (*kaho-adv*) have safety check mechanisms, which work in the target context. *livepatch* doesn't have such a step. *Trap* is the interruption due to the in-kernel trap handler; it also happens in the safety check step. Only *kaho-adv* uses this functionality. *Setup* is the interruption due to processing the `_init` function in the binary patch. Only *pannus* executes it in the activation.

4.1.2 Measurement

We placed probe points statically in the kernel to determine the interruption time in each category listed in Section 4.1.1. Also, we measured the interruption time in the *Trap* category from the target by getting processor's cycle counter.

The followings are the probe points we placed in kernel.

(A) `ptrace`-triggered stop/cont point (i.e., `TASK_TRACED` stop/cont). The `ptrace` is used for the *Allocation*, *Load*, or *Check* category.

(B) *kaho*'s utrace quiesce handler entry/exit for memory allocation. This is used for the *Allocation* category in *kaho-std* and *kaho-adv*.

(C) *kaho*'s utrace quiesce handler entry/exit for safety check. This is used for the Check category in *kaho-adv*.

(D) Setupper entry and restorer exit of the `_init` function in a patch. This is used used for the *Setup* category in *pannus*.

The following one is in user space (i.e., the target).

(E) Right before calling the fixed function and at the top of the fixed function. This is used for the *Trap* category in *kaho-adv*.

We conducted our measurements a hundred times per target. We took the mean value as a result.

4.1.3 Targets for the evaluation

To determine the performance and the characteristics of each RBP, three typical target programs described below were used.

- Single: Single-threaded application. The target function is continuously called and immediately returns. Because safety checks are very easy, this type of target can be used to determine the shortest time of the interruption.
- Multi-I: Multi-threaded application. The target function is frequently called from all of the threads. A hundred threads simultaneously access the function without any control. There is no outstanding resource contention across the threads in this target. Safety checks are very tough work, so this type of target can be used to determine the longest time. This is the most unfavorable condition for *kaho-adv* because it uses the trap handler for the check.
- Multi-II: Multi-threaded application and the target function is never called from any of the threads. A hundred threads run without accessing the function. There is no outstanding resource contention across the threads in this target. This target is the most favorable condition for *kaho-adv* because no one hits its trap handler. We used this target to estimate the effect of the trap on the interruption time.

4.1.4 Evaluation environment

We conducted our measurements on a Dual 2.66GHz Intel Quad-Core CPU machine with 4GB of RAM, which was installed with the Fedora Core 6 Linux distribution. When testing *pannus*, the 2.6.15.1 kernel plus the *pannus* patch was running on the machine, because the latest patch of *pannus* is against this kernel version. When testing *livepatch* and kaho, the 2.6.24 version of the `utrace` kernel plus the *kaho* kernel module was running on it. Because *livepatch* supports only the i386 architecture, we evaluated *livepatch* in ia32e mode on the x86_64 kernel.

4.2 Results

The evaluation results are summarized in Table 1 and in Figures 10-12. We can see that the results depend largely on the target, at first glance. For any targets, the interruptions by *kaho-adv* were shorter than those by *kaho-std* and *pannus*. The interruptions by *kaho-std* were of the same order as those by *pannus*. Although the results of *livepatch* are also presented, it is naïve to compare the results with *pannus* and *kaho* in terms of

Target	RBP	Total Interupt. (μs)	*Allocation*	*Load*	*Check*	*Trap*	*Setup*
Single	*livepatch*	2486	569 (A)	1916 (A)	–	–	–
Single	*pannus*	48	–	–	29 (A)	–	19 (D)
Single	*kaho-std*	43	5 (B)	–	37 (A)	–	–
Single	*kaho-adv*	10	6 (B)	–	3 (C)	0 (E)	–
Multi-I	*livepatch*	502253	100055 (A)	402197 (A)	–	–	–
Multi-I	*pannus*	4673323	–	–	4657529 (A)	–	15794 (D)
Multi-I	*kaho-std*	1034518	119 (B)	–	1034399 (A)	–	–
Multi-I	*kaho-adv*	1017455	111 (B)	–	594 (C)	1016750 (E)	–
Multi-II	*livepatch*	513139	99107 (A)	414032 (A)	–	–	–
Multi-II	*pannus*	5040145	–	–	5020773 (A)	–	19372 (D)
Multi-II	*kaho-std*	895451	121 (B)	–	895330 (A)	–	–
Multi-II	*kaho-adv*	634	112 (B)	–	522 (C)	0 (E)	–

Table 1: Total and break-down interruption time. '–' means N/A. The characters in parentheses indicate the probe point listed in Section 4.1.2.

the total interruption time. Because *livepatch* doesn't have a safety check, which is an necessary function to prevent unexpected results, the interruptions are short, especially for Multi-I and Multi-II. Therefore, we discuss the results without *livepatch* in the following sections.

4.2.1 Single-threaded target

The interruption-time distribution for the 'Single' case is shown in Figure 10. The interruptions by *kaho-adv*, *kaho-std*, and *pannus* were 10μs, 43μs, and 48μs respectively. The interruption by *kaho-adv* was about a quarter of that by *kaho-std* and *pannus*. We think the reason is that context switch of the target doesn't happen in *Check* by *kaho-adv*, which utilizes `utrace`. On the other hand, because *kaho-std* and *pannus* use `PTRACE_ATTACH` and `PTRACE_DETACH` for *Check*, the context switch happens at least twice.

4.2.2 Multi-threaded target I

The interruption-time distribution for Multi-I is shown in Figure 11. The interruptions by *kaho-adv*, *kaho-std*, and *pannus* were 1.02s, 1.03s, and 4.67s, respectively. This shows that the performance of all RBPs is comparable in this situation.

The interruption due to *Check* by *kaho-adv* was 594μs. This is three orders of magnitude shorter than that of *pannus* and *kaho-std*. However, almost all of the interruption by *kaho-adv* was consumed by *Trap*. When the fixed function was called via *Trap*, the time was 2.2μs longer than the time via direct jump in our evaluation. Even so, *Trap* happened about 440,000 times until *Activation* was completed. As a result, about 1s of interruption happened in total.

In the *kaho-std* and *pannus*, over 99% of interruptions were consumed by *Check*. Although their safety-check algorithms are almost the same, the interruption by *kaho-adv* is about a quarter of *pannus*'s result. The reason is unclear. It may be due to the difference of the base kernel versions.

4.2.3 Multi-threaded target II

The interruption-time distribution for Multi-II is shown in Figure 12. The interruptions by *kaho-adv*, *kaho-std*, and *pannus* were 634μs, 0.90s, and 5.04s, respectively. The interruptions by *kaho-std* and *pannus* differed little from the interruptions for Multi-I. However, the interruption by *kaho-adv* was notably reduced, because *Trap* was not used at all for the target. This means *kaho-adv* is much better than *kaho-std* and *pannus* when the target function is not called frequently.

5 Conclusion

This paper has proposed a runtime code modification method for application programs and an implementation

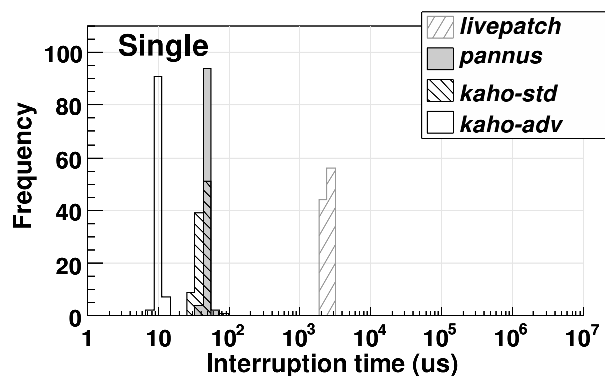

Figure 10: Interruption-time distribution for the single-thread target

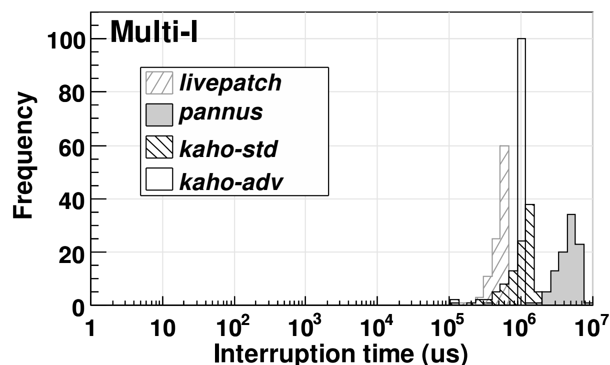

Figure 11: Interruption-time distribution for the multi-thread target I

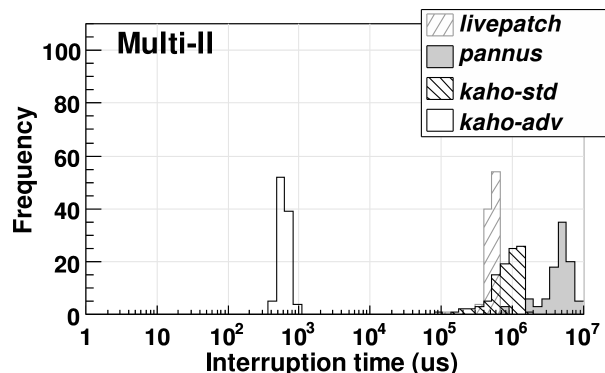

Figure 12: Interruption-time distribution for the multi-thread target II

named *kaho*. Such software is called Runtime Binary Patcher (RBP). The RBP is notably useful for applications used in telecom, which must continue running to keep the required level of system availability. The basic process of *kaho* is based on that of existing open source RBPs, *livepatch* and *pannus*. However, *kaho* has two major advantages. One is short interruption of target execution by the safety check using the trap instruction, which is inspired by `djprobes`. The other is easy maintenance using the `utrace` kernel APIs instead of the `ptrace` system call.

This paper also has shown the evaluation results with three conditions. Although the results depended largely on the condition, the interruptions by *kaho* were comparable to or shorter than interruptions by *livepatch* and *pannus* in all conditions. In a certain condition, the interruption by *kaho* was three orders of magnitude shorter than that by *livepatch* and *pannus*.

Acknowledgement

I wish to express my special thanks to Mr. I. Shikase and Mr. A. Kato of AIR Co., Ltd., who have developed the *kaho* utility program.

References

[1] `http://ukai.jp/Software/livepatch/`

[2] `http://pannus.sourceforge.net/`

[3] `http://people.redhat.com/roland/utrace/`

[4] `http://sourceware.org/gdb/`

[5] `http://lkst.sourceforge.net/djprobe.html`