Monitoring Instruction-based Intrusion Detection and Self-healing System

Hironori Shirai, Shoichi Saito, Koichi Mouri, and Hiroshi Matsuo

***Abstract*—Anomaly-based Intrusion Prevention Systems (IPSs) currently protect systems from zero-day attacks. However, since they either only inform administrators about the intrusions and/or just stop services or systems, they cannot stably continue services until the vulnerabilities are fixed by security patches. So self-healing systems (SHSs) are needed because they continue to safely execute services even if the applications have vulnerabilities. However since most SHSs rely on execution flows, e.g., system-call and library-function sequences, these systems cannot recover from non-control-data attacks. Such attacks target data that are not related with execution flow: user inputs, configuration data, etc. In this paper, we propose a novel SHS named RIN, Reactive INstruction-level recovery system, that uses instruction level rules for detection and recovery. RIN detects the falsifications of most data used in target applications and repairs them. We implemented RIN and evaluated it. The results show that it has sufficient functionality to find and fix the vulnerabilities for a function pointer and non-control-data.**

***Index Terms*—Intrusion Prevention System, Self-Healing System, Continuing execution, Invariants, DynamoRIO**

I. Introduction

Zero-day attacks continue to increase. Most attacks are based on the security holes of server applications. Security holes are fixed by patches distributed by various manufacturers, but zero-day attacks frequently happen before the vulnerabilities are fixed. Furthermore, the patches are often distributed too late. As a result, protecting the applications under vulnerabilities is a pressing problem. We must develop anomaly-based intrusion detection systems (IPSs) that can detect new or unknown attacks and self-healing systems (SHSs) that can recover applications and continue to execute them after they detect an anomaly. SHSs need high functionality for both detection and recovery components because they can recover applications only after detecting an anomaly.

Table I represents the detection and recovery types of existing IPSs and SHSs. Most IPSs [1]–[5] detect anomalies by buffer overflow or execution flows like system-call sequences. Consequently, since SHSs [6]–[13] cannot detect attacks that do not modify flows, recovery functions are not called. We must improve the detection methods.

We focus on ClearView [13] because it has the strongest recovery function. It uses an execution rule of operands (usually registers and memory) by instruction-based analysis for recovery. Therefore the system can recover almost all the values used in the target applications including non-control-data: user inputs, configuration data, etc. Contrary to recovery, it cannot detect non-control-data falsification because the system checks the illegal control flow for detection. The system also needs additional executions to collect the recovery information.

Manuscript received December 28, 2011; revised January 16, 2012.

Hironori Shirai, Shoichi Saito, and Hiroshi Matsuo are with the Nagoya Institute of Technology, Japan (e-mail: shirai@matlab.nitech.ac.jp, shoichi@nitech.ac.jp, and matsuo@nitech.ac.jp)

Koichi Mouri is with Ritsumeikan University, Japan (e-mail: mouri@cs.ritsumei.ac.jp)

TABLE I
Comparisons of related works

Detection type / Recovery type	Crash, Buffer overflow	Illegal flow
No-recovery (IDS, IPS)	StackGuard [1], CCured [2]	Belem [3], e-NeXSh [4], Memory-Firewall [5]
Packet filtering	ARBOR [6], Xu et al. [7], Rx [8]	
Checkpoint-rollback	Rx [8]	
Function-based flow control	Tobias H. [9], ASSURE [10]	
Shadow memory	SRS [11]	HookSafe [12]
Instruction-level recovery		ClearView [13]

Other SHSs [6]–[12] can recover from the point where they detected a crash or a flow and a small amount of information (packet logs, error return values, etc.) because they recover by simple ways. On the other hand, ClearView needs more information because it recovers in a more precise way. However, it uses flow-based detection like the other SHSs. As a result, ClearView needs additional executions to collect information for instruction-based recovery.

We use an instruction-based rule not only for recovery but also for detection to improve the detection accuracy and to reduce the number of reboots until recovery has been completed. In this paper, we propose a novel SHS named RIN, Reactive INstruction-level recovery system, that uses an instruction-based rule for both detection and recovery. RIN can detect and recover from non-control-data falsifications that are difficult for existing systems [6]–[13].

This paper is organized as follows. Section II describes the advantages of RIN, and Section III overviews our approach. Section IV describes the implementation, and Section V evaluates RIN. Section VI describes related research. We discuss our approach in Section VII and provide a conclusion in Section VIII.

II. Contributions

RIN can detect and recover all operands in the instructions used in target applications since RIN uses values of operands (registers and memory) as a regular execution rule (RER). Thus RIN can cover not only simple stack overflow and execution flows but also function pointers and non-control-data that are difficult for existing systems to handle. Additionally, since RIN can collect recovery information during a detection process because it uses the same RER for detection and recovery, it can skip execution for collecting recovery information after detection and need just one restart at the the earliest recovery.

RIN provides an environment in which applications can be executed safely for web servers until official recovery patches are distributed.

III. Design

We propose a new SHS named RIN that uses instruction-based rules for detection and recovery. RIN consists of three components: profiling, detection, and recovery. The profiling component generates an RER by executing target applications under safe environments. The detection component compares application states with the RER. The component determines an anomaly and immediately stops the application if it finds many RER violations. The component stores all the violated instructions as a violation list and reboots the application. After rebooting, the recovery component estimates the anomaly's cause in the violation list and generates and inserts a recovery code just before the recovery target instruction. Then the application is re-executed. If the detection component finds many violations, the anomaly's cause may be worng. Thus the recovery component re-estimates the anomaly's cause and recovers it.

RIN repeats this recovery process until the violation number becomes lower than a threshold. The recovery code automatically fixes the value after completing the recovery, even if the application is attacked in the same way. The remainder of this section describes RIN details.

A. *Profiling component*

RIN uses the invariants of the source operands of the instructions for an RER, which is created by the profiling component. The component executes a target application in safe environments and observes and stores the values of the source operands (the registers and memory) of the instruction. Then it analyzes the stored values and generates invariants as an RER. Invariants are formulas that are effective in each instruction. Examples of invariants include operand x takes one of constants c1, c2, ..., cn and operand x is larger than operand y. RIN can judge whether operands are normal because of the invariants and uses them for anomaly detection and recovery.

B. *Detection component*

The detection component checks whether a target application corresponds with an RER. The component compares the values of the operands of each instruction executed in the application with an RER immediately before executing the instruction. RIN can detect the falsifications of values used by instructions. This means that it can cover almost all the values used in the applications. When the component detects an anomaly, it stores all violated invariants and values as a violation list. Then the component reboots the application to restore it to a normal state.

However, some false positive violations occur depending on the amount of learning because RIN uses dynamic profiling to create an RER. To avoid detecting an anomaly at a normal state, RIN only detects one when the frequency of the violation occurrence exceeds a threshold.

C. *Recovery component*

After the detection component detects an anomaly and reboots the application, the recovery component recovers the vulnerabilities.

Our recovery component has three steps: recovery target selection, recovery code generation, and recovery code insertion. After the vulnerability is repaired, an alerted value is automatically fixed and the application can continue to execute. The remainder of this section describes the three recovery steps.

1) Recovery target selection: In this step, the recovery component estimates an anomaly's cause and determines a recovery target using the following three bases:

1) Successfully recovered invariants: This is the first priority basis. The component preferentially selects the invariants that were successfully recovered in past executions. On the other hand, invariants that failed to be recovered are not selected. RIN can learn and detect an actual cause by repetitive executions and this basis.
2) Correlated invariants: This is the second priority basis, which is only checked when the first priority basis cannot select a unique invariant. The component preferentially selects invariants that are more frequently violated when an anomaly was detected because these invariants are highly correlated with anomalies.
3) Previously violated invariants: This is the third priority basis. The component selects previously violated invariants because they may cause later violations.

These bases can determine a unique invariant to be recovered. For example, at the first anomaly detection, no invariants have been recovered yet and every violated invariant has the same degree of correlation. Thus the successfully recovered and the correlated invariant bases cannot reduce the candidates. Then the earliest violated invariant is selected by the previously violated invariant bases.

If the first recovery failed, RIN tries recovery again. This time, the component does not select the invariant that failed to be recovered the first time. The component selects a recovery target using the second and third bases.

2) Recovery code generation: In this step, the component generates a recovery code for the invariant selected by the previous step. For example, operand x and constants $c0, c1, ..., cn$ have relationship $x \ni \{c0, c1, ..., cn\}$. The component fixes x to first element $c0$ if x isn't in $c0, ..., cn$:

$$if(x! \ni \{c0, c1, ..., cn\}) \quad x = c0.$$

For an RER in which operand x and constant c have relationship $(x <= c)$, the component fixes x to c if x takes a larger value than c:

$$if(x > c) \quad x = c.$$

For an RER in which operand x and constant c have relationship $(x < c)$, the component fixes x to $c - 1$ if x equals or exceeds c:

$$if(x >= c) \quad x = c.$$

For RERs $(x > c)$ and $(x >= c)$, the component fixes them in a similar way.

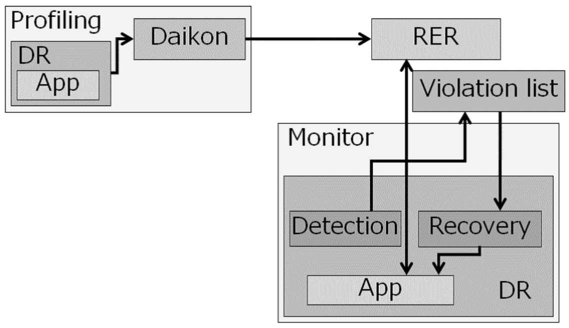


Fig. 1. System overview

For an RER in which operands x, y and constant c have relationship $(x = y + c)$, the component fixes x in the following way:

$$if(x! = y + c) \quad x = y + c.$$

There are many other RER types. But we don't explain them all here.

3) Recovery code insertion: After the component generates a recovery code, it inserts it just before the recovery target instruction. Violated invariants may not decrease much if the component incorrectly recovers the vulnerability. In this case, the component estimates and recovers again using the previous execution and current execution results. RIN continues to try recovery until the violated invariants become lower than a threshold. The value of the target operand is automatically fixed by the recovery code after completing the recovery, even if the application is attacked in the same way.

IV. Implementation

We describe RIN implementation for Linux. We used DynamoRIO [14] and Daikon [15] for RIN implementation. The RIN overview is shown in Fig. 1. DynamoRIO (DR), which is a runtime code manipulation system that supports code transformations on any part of a program, provides efficient, transparent, and comprehensive manipulation of unmodified applications. We used DR for obtaining the values of the operands of the instructions executed in target applications during the profiling mode and inserting monitoring and recovering codes during the monitoring mode. Daikon is an implementation of the dynamic detection of likely invariants; the Daikon invariant detector, which reports likely program invariants, is composed of two components: a front end that monitors applications and extracts the values of variables and a main invariant analyzer. We implemented a new front end that obtains operand-level trace data on DR.

The remainder of this section describes the implementation of each component of RIN: profiling, detection, and recovery.

A. Profiling

We monitor and store the values of the operands of the instructions executed in the target applications using the following DR function: *dr_insert_clean_call()*. This function enables RIN to insert a hook into an arbitrary position to read and write the registers and the memory in the applications in the hook processing. The profiling component outputs the values of the operands of each instruction executed in the application just before executing the instruction using the DR function. Then Daikon analyzes the values and creates an RER. Table II presents part of an RER from Apache 1.3. The first row is part of the assembly code for the analysis. This part was executed up to 98,010 times during the profiling time, and the values were observed by DR from the 2nd to the 4th rows. "op1" represents the first operand of an instruction, and "op2" represents the second. Daikon analyzed these values and the last row presents an RER created by Daikon. We explain the second line as an example. The RER indicates that the operand of the add instruction, register *edi*, has a power of 2. Table II displays only 2 and 16. However Daikon creates this RER because other values, for example 4 and 8 were also observed. On the other hand, Daikon could not create an RER for register *eps* on the last line, since *eps* had random values due to a linux security system named `randomize_va_space` that randomizes the memory space.

TABLE II
RULE

Assembly code	First run	..	98,010th run	RER

add %edi,%edi	op1 = 2
	op2 = 2
			op1 = 16
			op2 = 16
				op1==op2>= 2
				op1==power of 2

mov -16(%ebp) , %eax	op1 = 2		op1 = 5	op1 >= 1

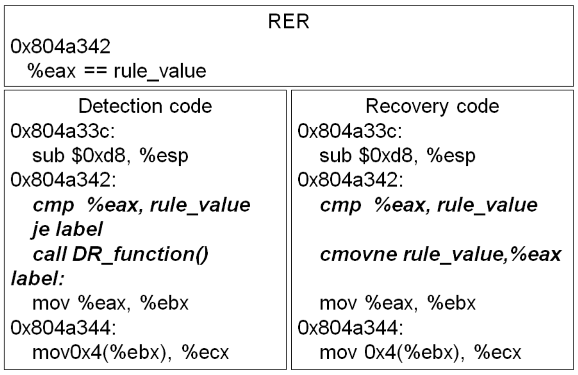
cmp 16(%ebp)
, %eax
	op1 = 2
	op2 = 5
			op1 = 5
			op2 = 5
				op1 >= 1
				op2 == 5
				op1 <= op2

jne 8091b1e	op1=8091b1e		op1=8091b1e	op1==8091b1e
mov %ebx,%eax	op1 = 2		op1 = 6	op1 2, 6
add $0xc,%esp	op2=bfdcee40		op2=bfe8df10	none

B. Detection and Recovery

The detection and recovery components use DR to obtain the application states the same as the profiling. The detection component compares an RER with the operand values observed by DR during the application execution. The component detects an anomaly when a certain amount of invariant violations occurs from a certain amount of invariant checks.

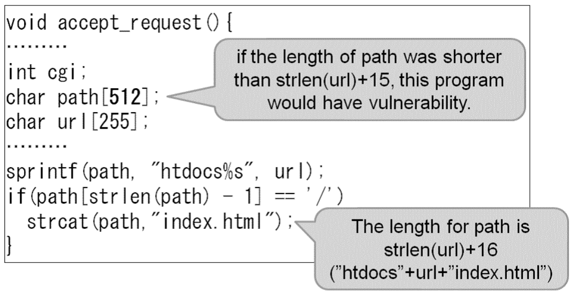
The detection and recovery components create detection and recovery codes and insert them into the application. The detection component generates and inserts detection codes when the target application starts. Similarly the recovery component generates and inserts recovery codes when the target application restarts after anomaly detection. Detection codes compare an RER with the operand values from the execution. If the values do not correspond with an RER, our hook function on DR is called by *dr_insert_clean_call()* and calculates whether the violations exceed a threshold value. We call the DR hook function only when an invariant is violated to reduce overhead. Large overhead is required to call DR functions in the target application because it involves context switches. Recovery codes use an instruction fixing an operand instead of the DR function call that detects an anomaly. Fig. 2 represents examples of detection and recovery codes. Fig. 2 has an RER on the upper side, a detection code on the left side, and a recovery code on the right side. In this example, the RER for an instruction (*mov %eax, %ebx*) at *0x0804342* is that register *eax* equals *rule_value*. The detection component inserts a compare instruction, a conditional jump instruction, and a function call immediately



```
RER
0x804a342
  %eax == rule_value
```

```
Detection code                     Recovery code
0x804a33c:                         0x804a33c:
  sub $0xd8, %esp                    sub $0xd8, %esp
0x804a342:                         0x804a342:
  cmp  %eax, rule_value              cmp  %eax, rule_value
  je label
  call DR_function()                 cmovne rule_value,%eax
label:
  mov %eax, %ebx                     mov %eax, %ebx
0x804a344:                         0x804a344:
  mov0x4(%ebx), %ecx                 mov 0x4(%ebx), %ecx
```

Fig. 2. Check and recovery codes



```
void accept_request(){
………
int cgi;
char path[512];
char url[255];
………
sprintf(path, "htdocs%s", url);
if(path[strlen(path) - 1] == '/')
  strcat(path,"index.html");
}
```

if the length of path was shorter than strlen(url)+15, this program would have vulnerability.
The length for path is strlen(url)+16 ("htdocs"+url+"index.html")

Fig. 3. Tiny HTTPd source vulnerability

before the target instruction as a detection code. If register *eax* violates an RER, the DR function is called. The function determines whether an anomaly exists, and if necessary stores a list of violated invariants and stops the target application. The recovery component inserts compare and conditional move instructions after the detection component detects an anomaly and reboots the target application.

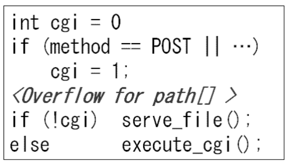
When the detection and recovery components create these codes, these components may insert *push* and *pop* instructions to acquire and release registers and *lahf* and *sahf* instructions to store and restore arithmetic flags. As an example, RIN acquires registers for rule $x = y + a$ because it has to calculate $y + a$. If the target instruction reads the flags, RIN stores the arithmetic flags, e.g., a *cmp* instruction.

V. EVALUATION

We evaluated the detection and recovery functionalities and the overhead. We used a machine with an Intel Core2Duo E4500 processor (2.2 GHz) and 2 GBytes of RAM running Ubuntu 10.04 with kernel 2.6.32. In this evaluation, the detection component detects an anomaly when it finds more than three violations in 1000 invariant checks.

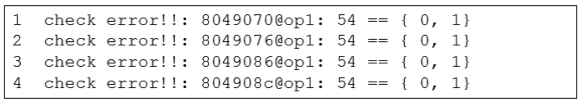
A. Detection and Recovery Evaluation

We manually created two types of vulnerabilities and executed them on our system to show that it can detect and fix them. As stated above, the detection threshold is three in these evaluations, and our system stops the target applications after it finds a third violation in a normal situation. However, we continued to execute the applications to the end and showed all logs for simplicity.



```
int cgi = 0
if (method == POST || …)
    cgi = 1;
<Overflow for path[] >
if (!cgi)  serve_file();
else       execute_cgi();
```

Fig. 4. Tiny HTTPd source overview



```
check error!!: 8049070@op1: 54 == { 0, 1}
check error!!: 8049076@op1: 54 == { 0, 1}
check error!!: 8049086@op1: 54 == { 0, 1}
check error!!: 804908c@op1: 54 == { 0, 1}
```

Fig. 5. Tiny HTTPd attack evaluation result

1) Simple case: We created an overflow vulnerability on *Tiny HTTPd* [16] and evaluated it. An abstract of the *Tiny HTTPd* source is shown in Fig. 3. The max length of string *path* is $url + sixteen$ because string *path* is created from string *url*. Buffer overflow may occur if we set *path* too short. The buffer overflow over-writes variable *cgi* when we use this vulnerability. The variable, which is used to recognize a page type, is set to 0 for a normal HTTP page and 1 for a CGI program (Fig. 4). We can call *execute_cgi()* forcibly by altering *cgi* to a non-zero value.

We executed this vulnerable Tiny HTTPd program on RIN, which detected an anomaly due to the four violations occurred. Fig. 5 shows the result. Here is the log format: {instruction addr}{operand number}: {actual value} {RER} . For example, the first operand of the instruction on *0x08049070* should have 0 or 1, but its actual value was 0x54.

All four violations are related to variable *cgi*. RIN stopped and restarted the application to reset a polluted state after the system detected these violations. The recovery component determined a recovery target invariant using the recovery target selection bases: successfully recovered invariants, correlated invariants, and previously violated invariants (described in Section III-C1). In this case, all four violated invariants had the same priorities for the successfully recovered invariant basis and correlated the invariants basis because the system detected an anomaly for first time for this vulnerability. That is why the component chose the first violation invariant on *0x08049070* as a recovery target by the previously violated invariants basis. The component generated a recovery code that fixed an operand indicated by the invariant and sets it to 0 if the operand is neither 0 nor 1. RIN inserted the code immediately before the instruction to automatically recover the operand. We confirmed that Tiny HTTPd executed normally, and the client displayed a requested page.

2) Complicated case: We created a second case that needed more than one recovery try. We generated a test program that included a format string vulnerability and attacked a function pointer. Fig. 6 shows this program's abstract. The program displays string *str* given by the first parameter and calls function pointer *fp1* or *fp2* depending on integer *num* given by the second parameter. Function pointers *fp1* and *fp2* have fixed addresses. A vulnerability exists on line 8 in Fig. 6 because an arbitrary string was passed to the first parameter of *printf* as a format string. This vulnerability may falsify arbitrary address memory. We

```
 1 int main(int argc, char *argv[]){
 2   void (*fp1)(void)= func1;
 3   void (*fp2)(void)= func2;
 4   char str[256];
 5   int num;
 6   strcpy(str,argv[1]);
 7   num = atoi(argv[2]);
 8   printf(str);          //this has vulnerability
 9   if(num < 0)
10     if(fp1!=NULL) fp1();//fp1 is altered to func2
11   else
12     if(fp2!=NULL) fp2();
13 }
```

Fig. 6. Second evaluation source

```
$ a.out $(printf "¥x98¥xf8¥xff¥xbf")%08x$
  (printf "¥x99¥xf8¥xff¥xbf")%08x$(printf "¥x9a¥xf8¥xff¥xbf")
  %08x$(printf "¥x9b¥xf8¥xff¥xbf").%08x.%08x.%08x.%08x.%08x.
  %08x.%08x.%08x.%08x.%08x.%164x.%n.%91x.%n.%125x.%n.%258x.%n -2
```

Fig. 7. Second evaluation exploit code

```
1   check error!!: 80485d4@op1: -2 == { -1, 0}
2   check error!!: 80485f0@op1: -2 == { -1, 0}
3   check error!!: 80485f7@op1: 8048528 == 8048514
4   check error!!: 80485fb@op1: 8048528 == 8048514
5   check error!!: 80485ff@op1: 8048528 == 8048514
6   check error!!: 8048603@op1: 8048528 == 8048514
7  func2 is called
8   check error!!: 804853b@op2: 8048605 == 8048614
```

Fig. 8. First result for second evaluation

```
1   check error!!: 80485f7@op1: 8048528 == 8048514
2   check error!!: 80485fb@op1: 8048528 == 8048514
3   check error!!: 80485ff@op1: 8048528 == 8048514
4   check error!!: 8048603@op1: 8048528 == 8048514
5  func2 is called
6   check error!!: 804853b@op2: 8048605 == 8048614
```

Fig. 9. Second result for second evaluation

disabled *randomize_va_space*, which randomizes the address spaces to simplify the attack code in this evaluation. We inputted strings for the first parameter and 0 and -1 for the second parameter to the test program running on the profiling mode of our system.

We created an exploit code that alters *fp1* from *func1* address to *func2* address (Fig. 7). The first parameter string in the code changes *fp1* on *0xbffff898* from *0x08048514* where *func1()* is located to *0x08048528* where *func2()* is located.

We used -2 as the second parameter to generate false positives because -2 is not a vulnerable value, but the value was not used during the profiling time.

The first execution result is shown in Fig. 8. "func 2 is called" on line 7 is an output from the application and represents that *func2()* was called instead of *func1()*. Violations on the first and second lines are false positives, since RIN assumes that *num* only takes 0 and -1 due to insufficient learning. Next the application tried to call *fp1* because *num* was minus. However, *fp1* had the address of *func2() 0x08048528* instead of the address of *func1() 0x08048514*. There are four violations for *fp1* from lines 3 to 6. Additionally, a violation on line 8 occurred at the return of *func2()*. *Func2()* should be called from line 12 in Fig. 6 and returned to line 13, but it was called from line 10 and returned to line 11.

RIN generated and inserted a recovery code for an operand

```
1   check error!!: 80485d4@op1: -2 == { -1, 0}
2   check error!!: 80485f0@op1: -2 == { -1, 0}
3  func1 is called
```

Fig. 10. Final result for second evaluation

on the first violated invariant at *0x080485d* (Fig. 9). RIN fixed *num* from -2 to -1, and the first two violations disappeared. -2 for num is not an abnormal value. The violations came from a learning shortage. Therefore real violations for *fp1* remain. RIN found that recovery failed, because many violations remained. Then it tried recovery again.

The recovery component re-chose a recovery target from the violations that occurred in Figs. 8 and 9. A violated invariant on the first line in Fig. 8 was not chosen for recovery because the system failed to recover. Also, the violated invariant on the second line in Fig. 8 was not chosen for recovery because the invariant had a lower degree of association with the anomaly than the other violated invariants. RIN chose a recovery target from the other five violated invariants since they were never used as recovery targets and had the same degree of association with anomalies. As a result, the earliest violation invariant on the third line in Fig. 9 was chosen for a recovery target. Only false positive violations for *num* remained, and *func1()* was called normally (Fig. 10). RIN completed its recovery because the total violations became lower than a threshold.

B. Monitoring overhead

We executed *Apache1.3* and *Apache bench* on different computers on the same LAN to evaluate the monitoring overhead. The average processing times for handling 1000 requests are shown in Table III. Apache execution time was about 0.209 ms without RIN. The time was 311.354 ms when RIN observed the operand values for creating an RER. This routine required larger overhead than the others. However, there are no problems, because we only had to execute this routine a few times before starting a service. The monitoring time was 0.552 ms with heavier overhead than we expected. The main reason for such overhead is from the code cache; DR places application codes into it.

Our preliminary evaluation shows increased overhead when we created unexecuted code regions and inserted big size codes into the regions. In RIN, a checking code that averaged 7.3 instructions for each invariant is only executed. A DR function call code created by *dr_insert_clean_call()* that averaged 60 instructions is not executed when there is no invariant violation. We evaluated the test program that only inserts checking codes without DR function calls. This result is shown on the last line in Table III. *Monitoring* and *Monitoring2* executed the same instructions, because we were not attacked and there are no violations. However these overheads are extremely different. We plan to reduce the amount of inserted codes and optimize the code cache.

C. RER generation overhead

Table IV represents the times for generating RERs and the number of invariants in the RERs and the instructions executed at least one time in the target applications. *Profiling time* is the time during which RIN observes the values.

TABLE III
OVERHEAD EVALUATION

	Time per request [ms]	Ratio
Native Apache	0.209	1.00
Profiling	311.354	1489.73
Monitoring	0.552	2.64
Monitoring2 [a]	0.257	1.22

[a]Monitoring without DR func call

TABLE IV
RULE CREATING TIMES

	Profiling time	Analyzing time	Generated invariants	Target instructions
Tiny HTTPd	3 min	15 sec	1026	991
Test program (Fig. 6)	1 min	3 sec	352	188
Apache	25 min	113 min	16306	18659

Analyzing time is the time during which Daikon analyzes the values observed by the previous step. The numbers of generated invariants are shown on the *Generated invariants* row. *Target instructions* is the number of instructions executed at least one time during the profiling time. They are the targets to generate invariants. In this evaluation, *Apache* needed more time to generate an RER and the total invariants are larger because the program size is bigger than the others. Averages of one or two invariants were generated for one instruction.

VI. RELATED WORK

In this section, we describe the related work of IPSs and SHSs and classify them by detection and recovery types and their usage of dynamic code manipulation (if applicable).

1) Detection types: One of the most popular detection methods are flow-based schemes. Detection methods that check system or/and library function calls may detect attacks to an operating system. E-NeXSh [4] and Belem [3] check the order of the system and library function calls. Hooksafe [12] protects function pointers and return addresses using shadow memory. MemoryFirewall [5] restricts control transfers and defends against binary code injection attacks. These systems can detect attacks using shellcodes, library functions, and system calls, e.g., execute commands, modify files, escalate privilege, etc . However they cannot detect attacks to application states that do not modify the application flow as opposed to instruction-based detection.

Most other systems [6]–[11] use segmentation faults or/and simple memory check systems such as StackGuard [1], CCured [2], etc. These systems trust memory randomization or only target bugs. However, bugs in server applications may be targeted for attacks, and memory randomization cannot protect attacks that do not rely on memory addresses.

2) Self-healing types: ARBOR [6] and Xu et al. [7] recover applications by filtering packets. Both systems protect from buffer overflow. ARBOR, which checks the packet length and identifies the length that causes crashes, filters long packets that cause crashes and keeps applications normal. However ARBOR cannot recovery attacks that have no relationship to packet length. Xu et al. finds instructions that cause crashes and identifies the memory region overwritten by overflow. They identify packets that have the same contents with an overwritten region. However, these systems cannot detect an anomaly if a crash does not occur, and they cannot perform a recovery if the attack type is not packet-based overflow. These systems stop packets, but our system directly fixes falsified data. Therefore if an innocent user accidentally attacks bugs and a user request includes both a malicious code and normal code, these systems omit not only the malicious code but also the normal code. On the other hand, RIN can fix only the malicious parts and executes them.

Michael E. et al. [17], ASSURE [10], and Tobias H. [9] use functions to recover. Michael E. et al. predict function return values for recovery and use previous function return values with context information that includes parent and sibling functions. Tobias H. et al. rollback memory and force return functions. When their system detects an anomaly, it reboots a target application. Then the system creates checkpoints near the detection point. The return values are fixed by types; the system uses -1 for integer types and NULL for pointer types. ASSURE, which uses checkpoints and return functions with an error value obtained by simulations, obtains error return values by passing bad inputs to the function. It also optimizes the checkpoint's places by repetitive executions. These systems assume that a parent function handles an error properly. Thus they may not be able to recover if a parent function does not have an error handling or an error handling leads to an exit from the application. On the other hand, RIN can recover even when a parent function does not have error handling.

Rx [8] recovers the applications by checkpoint-rollback and re-execution. When it detects an anomaly, it restores the application states and re-executes the application under different environments: memory map, scheduling, signal timing, and packet filtering. However, checkpoints-rollback and re-execution recoveries may fail to recover by variance with external processes communicating with the application. In contrast, our system does not need rollbacks.

SRS [11] focuses on independent user requests. SRS creates writing logs for shared memory between threads. When a thread that handles requests crashes, SRS rollbacks the memory written by the crashed thread and recovers the application by isolating the faulty threads from the others. This system only applies to typical server applications, but our system can be applied to other type of applications.

3) Systems using dynamic code manipulation tools: Locasto et al. [17], SRS [11], and ClearView [13] use dynamic code manipulation systems, as does RIN. Locasto et al. use PIN [18] for hooking function enter/exit events, because these event hooks are not influenced by compiler optimizations and signal events. SRS uses DR for shadow memory. ClearView uses DR to obtain the values of the instruction operands like RIN.

VII. DISCUSSION AND FUTURE WORKS

Our concerns are false detection and false recovery. Existing systems ([1]–[13]) have low false positive rates. Thus we plan to use static invariants to reduce false positives. We will detect and recover using both static and dynamic invariants. Static invariants provide less information and are completely accurate. On the other hand, dynamic invariants offer richer information but less accuracy. RIN can detect when one static invariant or many dynamic invariants violations have occurred. Additionally, our invariants do not include rules among more than one instruction because we generate invariants for each instruction. Hence it is possible to change only one operand into a value that is appropriate for invariants. At that time, our system may not detect an anomaly even if the operand is mismatched with other operands. The recovery component has the same problem. It cannot find a unique value for rule `eax>rule_value` (in our current implementation, we fix register eax to `rule_value+1`). We are going to generate invariants that have relationships among operands belonging to different instructions. With these invariants, a detection component may identify an anomaly of an operand from other operands, even if the operand stays within a normal range itself. Similarly, the recovery component probably finds a unique value to fix. We are implementing static and dynamic invariants among more than one instruction.

VIII. Conclusion

We proposed a new type of self-healing system named RIN that uses instruction-based rules for detection and recovery. RIN detects and recovers almost all values used in the target applications because the operands of instructions are targets for detection and recovery. RIN can detect and recover values unrelated to flow that existing systems cannot.

We implemented RIN on Linux and evaluated it. We confirmed that RIN detected and recovered from falsification for a local variable and a function pointer. The overhead for monitoring *Apache1.3* was 2.6 times. One main reason for this overhead is code cache efficiency. When we insert bigger codes, the DR overhead becomes bigger, even if the codes are not executed. Therefore we are going to reduce the amount of codes that we insert and optimize the code cache to reduce the overhead.

Our concerns are false detection and false recovery. Invariant violations occurred if a user does an action that RIN did not profile. That is why RIN ignores a few violations. However, RIN incorrectly detects an anomaly if the learning was too short. So we plan to use static invariants to reduce false positives. We will also implement dynamic invariants among more than one instruction to improve the detection and recovery accuracy.

References

[1] C. Cowan, C. Pu, D. Maier, H. Hinton, and J. Walpole, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security Symposium*, 1998.

[2] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "Ccured in the real world," *Sigplan Notices*, vol. 38, pp. 232–244, 2003.

[3] Y. Kato, Y. Makimoto, H. Shirai, H. Shimizu, Y. Furuya, S. Saito, and H. Matsuo, "Monitoring library function-based intrusion prevention system with continuing execution mechanism," in *Proceedings of the 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. IEEE Computer Society, 2010, pp. 548–554.

[4] G. Kc and A. Keromytis, "e-NeXSh: Achieving an effectively non-executable stack and heap via system-call policing," in *Computer Security Applications Conference, 21st Annual*. IEEE, 2005.

[5] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proceedings of the 11th USENIX security symposium*, 2002, pp. 191–206.

[6] Z. Liang and R. Sekar, "Automatic generation of buffer overflow attack signatures: An approach based on program behavior models," in *Computer Security Applications Conference, 21st Annual*. IEEE, 2005.

[7] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic diagnosis and response to memory corruption vulnerabilities," in *Computer and Communications Security*, 2005, pp. 223–234.

[8] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies - a safe method to survive software failures," in *Symposium on Operating Systems Principles*, 2005, pp. 235–248.

[9] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a reactive immune system for software services," in *USENIX Technical Conference*, 2005, pp. 149–161.

[10] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. Keromytis, "Assure: automatic software self-healing using rescue points," in *ACM SIGPLAN Notices*, vol. 44, no. 3. ACM, 2009, pp. 37–48.

[11] V. Nagarajan, D. Jeffrey, and R. Gupta, "Self-recovery in server programs," in *Proceedings of the 2009 international symposium on Memory management*. ACM, 2009, pp. 49–58.

[12] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.

[13] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. fai Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard, "Automatically patching errors in deployed software," in *Symposium on Operating Systems Principles*, 2009, pp. 87–102.

[14] D. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Citeseer, 2004.

[15] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[16] "Tiny httpd's tiny homepage." [Online]. Available: http://tinyhttpd.sourceforge.net/

[17] M. E. Locasto, A. Stavrou, G. F. Cretu, A. D. Keromytis, and S. J. Stolfo, *Return Value Predictability Profiles for Self-healing*, 2008.

[18] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.