Base Address Recognition with Data Flow Tracking for Injection Attack Detection

Satoshi Katsunuma†, Hiroyuki Kurita†, Ryota Shioya†, Kazuto Shimizu†, Hidetsugu Irie‡,†, Masahiro Goshima†, and Shuichi Sakai†

†Graduate School of Information Science and Technology, The University of Tokyo
‡Japan Science and Technology Agency
{katsu05, kuri, shioya, kshi, ern, goshima, sakai}@mtl.t.u-tokyo.ac.jp

Abstract

Vulnerabilities such as buffer overflows exist in some programs, and such vulnerabilities are susceptible to address injection attacks. The input data tracking method, which was proposed before, prevents I-data, which are the data derived from the input data, being used as addresses. However, the rules to determine address injection attacks are vague, which produces many false-positives and false-negatives in detection results. Generally, the data used as an address consist of a base address and an address offset. We propose an architectural technique to prevent I-data overwriting B-data, which are the data used as base addresses in this paper. It dynamically recognizes the I-data and the B-data. Address injection is detected if I-data that are not B-data are used as addresses. We implemented the proposed technique on a Pentium-based Bochs emulator and investigated its detection capability. I believe that the technique is the most accurate injection detection technique proposed thus far.

Keywords: security, vulnerability, injection attack, base address, data flow tracking

1. Introduction

Security intrusions over the Internet have caused considerable damage through the acquisition and falsification of information, and abuse of resources. Majority of attacks exploit program vulnerabilities such as buffer overflows [11] and format string errors [10]. These vulnerabilities have a significant effect on computer security. Recently, Internet worms such as Code Red and W32/Nimda Worms have spread world wide and have attacked computers on the Internet by exploiting program vulnerabilities.

In particular, the attacks that exploit the vulnerabilities inject arbitrary data into programs. Although there are several types of injection attacks [14][7], the most frequent one is the injection of addresses. Code injection attacks are examples of such an attack. These attacks inject codes and the addresses of the codes, and then acquire the root privileges of programs. In this paper, the injection attacks that input addresses are referred to as address injection attacks. Address injection attacks can be divided into two types. The first type executes arbitrary codes by injecting instruction addresses. The second type loads and stores arbitrary data by injecting data addresses.

Input data tracking method [12][8][6] is an architectural method to detect address injection attacks. The method is transparent to the program. The method tracks I-data, which are the data derived from the input data in this paper, on the data flow and then detects the I-data being used as addresses. Since the method collectively detects address injection attacks, it prevents a larger number of injection attacks than conventional techniques such as data execution prevention [3] and return address protection [13]. Because I-data are often used as parts of addresses, the method of input data tracking considers compare instructions to be data range checking, and allows compared I-data to be used as addresses. However, since the safety of the I-data cannot be wholly determined by compare instructions, the method produces many false positives and false negatives in detection results.

Our proposed architecture accurately detects the injection attacks, based on address calculation; an address offset is added to or subtracted from a base address, and the result is used as an address. The proposed technique dynamically recognizes B-data, which are the data used as base addresses in this paper, and detects the injection attacks if I-data overwriting B-data are used as addresses.

The rest of this paper is organized as follows. Section 2 describes various types of injection attacks. Section 3
overviews the method of input data tracking. Section 4 presents our technique. Section 5 describes the evaluation of the proposed technique. Section 6 concludes the paper.

2. Injection Attacks

2.1. Program Vulnerabilities

Injection attacks exploit program vulnerabilities. This section describes stack buffer overflows [11], heap buffer overflows [4], and format string errors [10], which are the common injection attacks.

Buffer overflows occur in stack buffers, heap buffers, and static area buffers. If the data inputted to a buffer is larger than its allocated size, the input data will overwrite the data around the buffer. Figure 1 (a) shows a stack buffer overflow. A buffer _buf_ is allocated on a stack with 100 bytes. The function _scanf_() sends input data to the buffer _buf_. If data of more than 100 bytes are read by the function _scanf_(), the data overwrite other data on the stack such as the return address and local stack variables. Figure 1 (b) shows a heap buffer overflow. If the data sent to a heap buffer _buf_ are larger than the allocated area size of 100 bytes, the coming data overwrite other heap data. A heap is implemented as a doubly linked list, and contains the addresses of forward and backward nodes. If these addresses are overwritten with buffer overflows, then arbitrary memory areas can be accessed.

Format string errors are vulnerabilities in standard C library functions, notably the family function _printf_. Figure 1 (c) shows a format string error. Although the function _printf_() should be described as _printf_("%s", _buf_), it is described as _printf(_buf_)_ in Figure 1 (c). If a string _abcd%*x%x%n_ is inputted, the string _abcd_ can be used as the address data in the function _printf_. Thus, arbitrary memory areas can be accessed.

2.2. Address Injection

Injection attacks such as code injection attacks [14] exploit program vulnerabilities. Most of the injection attacks are carried out by injecting addresses. Instruction address injection attacks execute arbitrary codes by overwriting instruction addresses. Similarly, data address injection attacks load and store arbitrary data by overwriting data addresses.

Instruction address injection attacks overwrite instruction addresses with input data, and execute arbitrary code in code areas or injected code. Code injection attacks are the most frequently implemented injection attacks among instruction address injection attacks. A code injection attack injects a code, and overwrites an instruction address with the address of the injected code, leading the program to execute the injected code. Many code injection attacks exploit stack buffer overflows. An attacker overwrites the return address of a function, and when the function returns, the program starts to execute the injected code.

Data address injection attacks overwrite data addresses with the input data. If the overwritten addresses are used as store addresses, the memory data are unexpectedly overwritten, leading to corruption of important data. If the overwritten addresses are used as load addresses, the memory data are unexpectedly used. For instance, data address injection attacks may overwrite user ID, or print local files. These attacks can result in the acquisition of root privileges.

Some injection attacks are implemented without injecting addresses. Some attacks overwrite the flags used in condition branching and change the program controls. Other attacks directly overwrite important data on the memory with buffer overflows.

3. Related Work

Data execution prevention [3], return address protection [13], the methods of input data tracking are the architectural methods that dynamically detect injection attacks without static compiler analyses and static code translations. Since these techniques do not require the static analyses of the programs, they can be performed transparently by the users. Therefore, the users rely on these techniques to verify whether the program is safe or not. Such techniques are useful when the program may not be trustworthy since the creator of the program is unknown or cannot be trusted.
3.1. Data Execution Prevention

Data execution prevention does not allow the data in the non-code areas to be used as codes and thereby prevents code injection attacks. It adds a No-eXecute (NX) bit to each entry in the page table. The NX bit indicates whether the data in a page are executable or not. The technique decides that the data in the code areas are executable, while those in the other areas are not executable. Thus, even if injected addresses are used, the method prevents the injected data from being executed as codes. The NX bit concept has been employed in Windows XP as well as recent versions of Linux and openBSD.

3.2. Return Address Protection

Secure Return Address Stack (SRAS) [13] protects the return addresses and detects the instruction address injection attacks that exploit stack buffer overflows. When a function is called, SRAS created the redundant copy of the return address. When the function returns, SRAS checks the integrity of the return address with the copy. If the return address is changed, SRAS detects the instruction address injection attack.

3.3. Method of Input Data Tracking

Input data tracking methods such as Dynamic Information Flow Tracking (DIFT) [12], Minos [8], Pointer Taintedness Detection (PTD) [6] can detect address injection attacks. These methods dynamically recognize I-data, and do not allow them to be used as addresses. DIFT and Minos do not allow the I-data to be used as instruction addresses, and thus can prevent instruction address injection attacks. PTD does not allow the I-data to be used as data addresses as instruction addresses, and detected data address injection attacks similarly. Our proposed technique is based on the PTD technique. The PTD technique and its problems are as follows.

PTD adds a bit, which is referred to as a taintedness bit, to each byte of the data. Basically, the taintedness bits of I-data are set to 1, and those of the other data are set to 0. If the tainted data are used as jump/branch addresses and load/store addresses, PTD considers the tainted data as an injection attack. Input data may be received through network, file system, and keyboard. When the operating system inputs the data, it initializes the taintedness bits of the data. The taintedness bits are then propagated from the source operands to the destination operands for data-transfer and arithmetic instructions. If any of the source operand data are tainted, the destination operand data also become tainted. On the other hand, if all the source operands are not tainted data, the destination operand data does not become tainted data.

I-data are often used as parts of addresses. For example, I-data can be used as offsets from array base addresses. Thus, PTD considers compare instructions as data range checking and permits the use of the compared I-data as addresses. However, PTD produces false positives because it is difficult for the compare instructions to determine whether the I-data can be safely used as addresses or not. In addition, PTD also produces false negatives because the compare instructions falsely consider the I-data as safe data.

False Positives When the I-data are used as parts of addresses without data range checking, as shown in Figure 3(a), PTD produces false positives. In Figure 3(b), the program copies the I-data, and the copy is checked for data range. However, if the original I-data are used as an address offset, PTD still recognizes it as an injection attack. In Figure 3(c), when the I-data are compared by the subtract instruction, the programs have false positives. In this way, PTD does not permit the use of the I-data whose data range has not been checked as parts of addresses, and demands that the data range checking must be directly applied to I-data with specific compare instructions.

\[
\begin{align*}
\text{IN} & \text{ R_in} \\
\text{ADD} & \text{ R1} \leftarrow \text{ R_base} + \text{ R_in} \\
\text{MOV} & \text{ R1} \leftarrow \text{ R_in} \\
\text{MOV} & \text{ R3} \leftarrow [\text{ R1}] \\
\text{CMP} & \text{ if } \text{ R1} < 0x10 \\
& \text{ then } \text{FLAG} \leftarrow 1 \\
\text{ADD} & \text{ R2} \leftarrow \text{ R_base} + \text{ R_in} \\
\text{CMOV} & \text{ if } \text{FLAG} = 1 \\
& \text{ then } \text{R3} \leftarrow [\text{ R2}] \\
(a) & \\
\text{IN} & \text{ R_in} \\
\text{SUB} & \text{ R1} \leftarrow \text{ R_in} - \text{ R2} \\
\text{CMP} & \text{ if } \text{ R1} < 0x00 \\
& \text{ then } \text{FLAG} \leftarrow 1 \\
\text{ADD} & \text{ R3} \leftarrow \text{ R_base} + \text{ R_in} \\
\text{CMOV} & \text{ if } \text{FLAG} = 1 \\
& \text{ then } \text{R4} \leftarrow [\text{ R3}] \\
(c) & \\
\text{IN} & \text{ R_in} \\
\text{SUB} & \text{ R1} \leftarrow \text{ R_in} - \text{ R2} \\
\text{CMP} & \text{ if } \text{ R1} < 0x00 \\
& \text{ then } \text{FLAG} \leftarrow 1 \\
\text{ADD} & \text{ R3} \leftarrow \text{ R_base} + \text{ R_in} \\
\text{CMOV} & \text{ if } \text{FLAG} = 1 \\
& \text{ then } \text{R4} \leftarrow [\text{ R3}] \\
(b) & \\
\end{align*}
\]

Figure 2. Examples of false positives in PTD

False Negatives PTD produces false negatives when programs incompletely perform data range checking, or when programs execute those compare instructions that are not intended for data range checking. Programs have many compare instructions used for the purposes other than data range checking. For instance, in inputting the data, the programs frequently check whether the I-data are equal to the line
feed, null, and other characters, or that they are not negative. In the string function, programs also check that the I-data are not equal to null.

```c
void func4(){
    char buf[10];
    while( (c=getchar() ){
        buf[i++] = c;
    }
}
```

```c
void func5()
{
    FILE * f = fopen("INPUT", "r");
    fscanf(f, "%s", buf);
}
```

```c
void func6(){
    char buf[100];
    scanf("%s", buf);
    char cbuf[100];
    strcpy(cbuf, buf);
    cbuf = strtok(buf, "/");
}
```

```c
void func7(){
    char buf[100];
    char * cbuf;
    scanf("%s", buf);
    cbuf = strtok(buf, "/");
}
```

*Figure 3. Examples of false negatives in PTD*

For example, in Figure 3(a), a function `getchar()` receives input data from keyboard. Since the I-data are compared with a line feed, the input data can be used as addresses. In Figure 3(b), a function `fscanf()` read data from a file. Since the function `scanf()` checks for the end of input, the data are mistakenly allowed to be used as addresses. In Figure 3(c), when a function `strcpy()` copies a string `cbuf` from a string `buf`, the I-data are compared with null. Thus, the taintedness bits of I-data become 0. Similarly, string functions such as the function `strcat()` check for the end of the buffer. In Figure 3(d), a function `strtok()` separates a buffer `buf` with `/`. Since the I-data on the buffer `buf` are compared with `/`, it is possible to use the data as addresses. In this way, since PTD determines the safety of the I-data with compare instructions, it produces many false negatives.

4. Proposed Detection Technique

The conventional methods of input data tracking produce many false positives and false negatives. This is because the methods ignore the fact that data used as addresses consist of base addresses and address offsets. Before the address is used, the address offsets are added to or subtracted from the base address as shown in Figure 4. I-data are often used as address offsets, but not as base addresses. The technique does not consider execution of compare instruction as an indication for the safety of I-data. To recognize I-data and B-data on the data flow, the technique newly adds 2-bit tag per byte on actual memory words, registers, and caches, as shown in Figure 5. The two bits are named as I-bit and B-bit. These bits are initialized, and when instructions are executed, the bits are propagated from the source to the destination operands. If the I-data that are not added to or subtracted from the B-data are used as addresses, the B-data overwritten by I-data are detected.

4.1. Concept

The technique is realized through supports from processor and operating system. To detect the overwriting of B-data, the technique dynamically detects B-data. The I-data is also tracked on the data flow in a similar manner to the conventional methods of input data tracking. However, the technique does not consider execution of compare instruction as an indication for the safety of I-data. To recognize I-data and B-data on the data flow, the technique newly adds 2-bit tag per byte on actual memory words, registers, and caches, as shown in Figure 5. The two bits are named as I-bit and B-bit. These bits are initialized, and when instructions are executed, the bits are propagated from the source to the destination operands. If the I-data that are not added to or subtracted from the B-data are used as addresses, the B-data overwritten by I-data are detected.

4.2. I-Data Recognition

To detect the overwriting of I-data, the proposed technique recognizes I-data by using I-bits. The technique initializes the I-bits of the data received from the network, file system, and keyboard, as shown in Table 1. The I-bits are then propagated from the source to the destination operands,
Table 1. Tag initializations

<table>
<thead>
<tr>
<th>tag</th>
<th>instructions</th>
<th>initialization methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-bit</td>
<td>system calls read, recv</td>
<td>I(input data) ← 1</td>
</tr>
<tr>
<td></td>
<td>in program start</td>
<td>B(SP) ← 1</td>
</tr>
<tr>
<td>B-bit</td>
<td>system calls brk, mmap</td>
<td>B(return values) ← 1</td>
</tr>
<tr>
<td></td>
<td>add/subtract instructions of I-data and non-I-data</td>
<td>If R3 is a non-I-data, B(R3) ← 1</td>
</tr>
<tr>
<td></td>
<td>that are not B-data:</td>
<td>with static address checking.</td>
</tr>
<tr>
<td></td>
<td>ADD R1,R2,R3 (R1 ← R2 + R3)</td>
<td>After the propagation, B(R3) ← 0</td>
</tr>
</tbody>
</table>

as shown in Table 2. Namely, the I-bit of the destination operand is the OR of the I-bits of the source operands.

Figure 6. Virtual memory layout

4.3. B-data Recognition

In order to detect the overwritten B-data, the proposed technique recognizes the B-data by using B-bits. If I-data are B-data, their B-bits are set to 1; otherwise, they are set to 0. The B-bits of dynamic and static B-data are initialized through different means, as shown in Table 1. The B-bits are then propagated from source to destination operands for data-transfer instructions and arithmetic instructions, as shown in Table 2.

Dynamic B-data Initialization A dynamic area are a stack or a mapped area such as a heap, as shown in Figure 6. The memory locations where the data are allocated in the dynamic areas are determined during program execution. Therefore, the data used as the base addresses of the dynamic areas are dynamically determined, and these data are referred to as dynamic B-data in this paper. The dynamic B-data have their origins. Stack addresses originate from the Stack Pointer (SP) initialized at program start. Heap addresses originate from the return values of the system call brk, and the addresses of the mapped areas originate from the return values of the system call mmap.

As shown in Table 1, in order to recognize the dynamic B-data, the B-bits are initialized by the operating system; the B-bits of the data in SP are set to 1 at program start. The B-bits of the return values of system calls brk and mmap are set to 1. For example, when the mmap() call returns, The B-bit of the return value is set to 1. Thus, the dynamic B-data that point to the mapped area are initialized. In this manner, the dynamic B-data are recognized for both I-data and non-I-data.

Static B-data Initialization Static areas are static data areas and code areas, as shown in Figure 6. The memory locations where the data are allocated in the static areas are determined at program compilation time. Therefore, the data used as base addresses of the static areas are statically determined, and these data are referred to as static B-data in this paper. Since the memory locations of static data areas and code areas are fixed after program compilation, the data range of the static B-data are known before program execution. Therefore, the verification of the data that originate from the static data and immediate data can determine whether the data are the static B-data.

The technique initializes the B-bits by verifying whether the data point to the static areas. For this purpose, the operating system preserves the top and bottom addresses of the static areas in programs and dynamically linked libraries. The static B-data are recognized when the data are I-data. As shown in Table 1, if the I-bit of a destination operand becomes 1, the B-bits of the source operands are initialized; subsequently, after the tag propagation, the B-bits of these source operands become 0 since these source operands are not I-data. For example, when ADD R1,R2,R3 (R1 ← R2 + R3) is executed, if R2 is I-data and R3 is not I-data, it is verified whether the value of R3 points to static areas. If the value indicates these areas, the B-bit of R3 is set to 1; subsequently, after the B-bit of R1 is set, the B-bit of R3 is set to 0.

B-data Propagation In order to discriminate the B-data from the other data accurately, our technique propagates the B-bits on the data flow according to the address properties, as given in Table 2. For each instruction, the technique de-
terminates the B-bit of a destination operand, based on the B-bits of source operands and the opcode.

For data transfer instructions, the B-bit of the destination operand is equal to that of the source operand. For add/subtract/AND instructions, the B-bit of the destination operand is determined as follows. Usually, an address offset is obtained if a base address is subtracted from another base address, and a base address is obtained if an address offset is added to or subtracted from another base address. Thus, for add/subtract instructions, if either but not both of the source operands is a B-data, the B-bit of the destination operand is set to 1; otherwise, the B-bit is set to 0. For AND instructions, when the high-order (or low-order) bits of a base address are extracted, the destination operand becomes a base address (or an address offset). Therefore, if AND instruction has a B-data and the data whose highest-order bit is 1 as source operands, the B-bit of the destination operand is set to 1; otherwise, the B-bit is set to 0. For example, when AND $R1,R2,0x11..00 (R1 ← R2 & 0x11..00)$ is executed, the B-bit of R1 is set to 1 if the B-bit of R2 is 1. On the contrary, when AND $R1,R2,0x00..11 (R1 ← R2 & 0x00..11)$ is executed, the B-bit of R1 is set to 0 even if the B-bit of R2 is 1.

For other arithmetic instructions, including two-operand instructions such as multiplication, division, and OR, and one-operand instructions such as shift and NOT, the B-bit of the destination operand is always set to 0 regardless of the values of the B-bits in the source operands.

### 4.4. Injection Attack Detection

When the memory is accessed through jump/branch and load/store instructions, our proposed technique determines whether an injection attack exists or not, as shown in Table 3. If the data used as either the instruction address or the data address has its I-bit set and B-bit unset, the data is B-data overwritten by I-data. Therefore, the technique recognizes the access as the result of an address injection attack. For other combinations of the values of I-bit and B-bit, the accesses are recognized as safe accesses.

### 5 Evaluation

We implemented the proposed technique on a Pentium-based Bochs emulator [1] and investigated its detection capability and its performance overhead.

#### 5.1 Simulation Setting

We implemented an instruction emulation using a Pentium emulator -Bochs- mounting Red Hat Linux 6.2 (kernel 2.2). Bochs can emulate x86 instruction set, and execute a real operating system. We extended Bochs and implemented the proposed technique. We also implemented PTD technique for comparison purpose. We used a network application and test programs containing the vulnerabilities to investigate the detection of injection attacks. Further, we used integer programs from SPEC 2000 benchmark to investigate the detection accuracy and performance overhead.

#### 5.2 Detection of Injection Attacks

First, we implemented address injection attacks for the test programs shown in Figure 1(a), (b), (c). These programs have stack/heap buffer overflows and format string errors. As the result, the proposed technique successfully detected the injection attacks in all of the programs.

Next, we compared the detection ability of the proposed technique with that of PTD. We used an HTTP server GHTTPD [2], which has a buffer overflow in the function GHTTPD [2], which has a buffer overflow in the function

![Image of Table 2. Tag propagations]

<table>
<thead>
<tr>
<th>I-bit</th>
<th>arithmetic instructions: $MUL R1,R2,R3 (R1 ← R2 * R3)$</th>
<th>propagation methods: $I(R1) ← I(R2) \text{ or } I(R3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>data transfer instructions: $MV R1,R2 (R1 ← R2)$</td>
<td>$I(R1) ← I(R2)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B-bit</th>
<th>add/subtract instructions: $ADD R1,R2,R3 (R1 ← R2 + R3)$</th>
<th>propagation methods: $B(R1) ← B(R2)$ xor $B(R3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AND instruction extracting high bits: $AND R1,R2,0x11..00 (R1 ← R2 &amp; 0x11..00)$</td>
<td>$B(R1) ← B(R2)$</td>
</tr>
<tr>
<td></td>
<td>the other arithmetic inst.: $OR R1,R2,R3 (R1 ← R2 \or R3)$</td>
<td>$B(R1) ← 0$</td>
</tr>
<tr>
<td></td>
<td>data transfer instructions: $MV R1,R2 (R1 ← R2)$</td>
<td>$B(R1) ← B(R2)$</td>
</tr>
</tbody>
</table>

#### Table 3. Determining attacks in memory access

<table>
<thead>
<tr>
<th>I-bit</th>
<th>B-bit</th>
<th>used as address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>attack detection</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

---
log()). An attacker can overwrite the address of the data containing the name of a public access file with the address of the injected data containing the name of a local file. When the program loads the injected address, the attacker is able to access the local file. The proposed technique successfully detected the address injection attacks. On the other hand, PTD was unable to detect all of the attacks because the program performs checking of null data in the string function and PTD incorrectly interpreted such checking as data range checking for the input data to be safe.

We also used the test programs shown in Figure 3. In these programs, the I-data are compared in data input or these string functions. The proposed technique detected the address injection attacks in all of the programs. However, PTD cannot detect any of the attacks because the tainted bits of compared I-data became 0, as shown in Table 4.

### 5.3 Detection Accuracy

We investigated the detection accuracy by using the integer programs of SPEC 2000 benchmark. We executed gzip, parser, vpr (place), vpr (route), gcc, mcf, vortex, bzip2, and twolf until not more than 1G instruction. We evaluated the following data: (1) the number of false positives in PTD, (2) the number of false positives in the proposed technique, and (3) the number of B-data recognition errors.

The result showed that the proposed technique did not produce false positives in any program, while PTD produced many false positives, as shown in Table 5. The number of PTD false positives is different from that evaluated in a previous PTD study [6] because both methods use different data range checking definitions. For example, alpha binaries were used in the previous study. In x86 binaries, an instruction such as TEST R1,R1 often determines whether R1 is equal to 0. However, since alpha binaries do not have TEST, PTD studied in the paper regarded such instructions as those for data range checking. In this manner, in the study, the false positives were decreased by extending the data range checking, which led to an increase in the false negatives in PTD.

We also monitored following B-data recognition errors: (1) the dynamic B-data were used as the addresses of static areas, (2) the non-B-data were used as the addresses of the dynamic areas, (3) the static B-data were used as the addresses of the dynamic areas. If these B-data recognition errors occur frequently, the proposed technique can produce false positives and false negatives. As the result, the programs didn’t have any errors as shown in Table 5.

### 5.4 Performance Overhead

Our proposed technique adds 2-bit tag per byte on all memory data and registers. The technique initializes and propagates the tags in each instruction, and detects injection attacks in memory access. Operations on the tags can be executed in parallel with normal instructions. Further, since the technique operates on 2-bit data, the performance overhead is low.

However, when the technique initializes the static B-data, multiple comparisons of word-size data for static address checks are required and may have performance overhead. Using the integer programs of SPEC 2000 benchmark, we then measured (1) the number of the static address checks and (2) the number of the static address checks that need multiple comparisons. Since the programs can have multiple static areas, the multiple comparisons can be needed. However, if the checked value is smaller than the lowest static address, the check needs only one comparison.
As shown in Figure 7, the number of the static address checks was up to 17% of the number of instructions. The number of those checks requiring multiple comparisons was less than 2% in all of the programs. Therefore, by adding a comparator performing one arithmetic comparison per cycle, the technique can keep the performance overhead at low.

6 Conclusions

This paper proposed an address injection attack detection technique that monitored I-data, which are the data derived from the input data, being used as base addresses. The proposed technique recognized I-data and B-data, which are the data used as base addresses, on the data flow and detected injection attacks if the I-data that were not the B-data were used as addresses. The data were recognized by propagation of newly added tags from the source to the destination operands of the executed instructions.

The proposed technique is evaluated and compared with PTD, which is the most effective one among the conventional methods of input data tracking. The results are: (1) the proposed technique detected the injection attacks that PTD was unable to detect, (2) the proposed technique did not produce any false positives, while PTD produced many false positives, (3) the proposed technique accurately recognized B-data. Therefore, the proposed technique is the most accurate injection detection technique proposed thus far. In future work, we are going to minimize the performance and area overheads.

Acknowledgement

This research is partially supported by CREST project of Japan Science and Technology Corporation, and by 21st century COE project of Japan Society for the Promotion of Science.

References