# The Implementation of Secure Canary Word for Buffer-Overflow Protection

Sirisara Chiamwongpaet
Department of Computer Engineering
Chulalongkorn University
Bangkok 10330, Thailand
Email: g51scm@cp.eng.chula.ac.th

Krerk Piromsopa
Department of Computer Engineering
Chulalongkorn University
Bangkok 10330, Thailand
Email: krerk@cp.eng.chula.ac.th

***Abstract*— Chiamwongpaet and Piromsopa [1] introduced Secure Canary Word, an extension of Secure Bit, as an architectural approach to the protection against buffer-overflow attacks on non-control data (variables and arguments). Secure Canary Word is based on two existing schemes, Secure Bit [2] and Canary Word. The objective of this paper is to propose a new hardware implementation in order to improve the efficiency of Secure Canary Word. To evaluate this design, the hardware simulation is conducted using BOCHS emulator running Linux (Red Hat 6.2) with GCC compiler. Like the prior work, the results confirm that Secure Canary Word can detect buffer-overflow attacks on non-control data. Furthermore, performance is significantly better than the original implementation. This suggests that Secure Canary Word can prevent buffer-overflow attacks on non-control data without any serious performance degradation or storage requirement.**

## I. INTRODUCTION

In 1988, the MORRIS worm [3], which is the notorious worm using buffer-overflow attacks, was discovered. From that time on, this kind of technique is still the most commonly used, although it is so simple that it can be prevented by writing code carefully. Nevertheless, no program is guaranteed to be without any bugs. There have been a lot of buffer-overflow vulnerabilities continuously detected and reported. For example, in US-CERT Vulnerability Notes [4], the latest note about this vulnerability (VU#905281) was published on $19^{th}$ February 2009. Even the Microsoft Windows Vista, which is claimed to be a security-enhanced operating system, was reported that the buffer-overflow problem was found as well [5].

Nowadays, there are many solutions of buffer-overflow attacks. However, they concentrate on protecting control data, such as return addresses and function pointers. A small number of them put an emphasis on non-control data, such as local variables (including pointers and arrays) and function arguments.

Secure Canary Word, proposed on [1], is in the latter class. This concept is the combination between Secure Bit [2] and Canary Word. Nonetheless, the efficiency of the original implementation is difficult to be accepted. The first reason is that the implemented example programs are much slower because they waste time calculating the address of Secure Canary Word in Secure-Canary-Word-verifying functions (`chkSBit`, `read_int` and `read`). The other reason is the big size of program resulting from using `inline` specifier of these functions by the optimization flag `-O3` turned on. Consequently, every call to these functions is expanded into a chunk of code.

Thus, the new implementation, hardware optimization, is dealing with the problems of efficiency and code size. The details are described as follows. The background knowledge section reviews the definition of buffer-overflow attacks and the buffer-overflow protection related to Secure Canary Word. The concept of Secure Canary Word section explains the idea of this approach to preventing these attacks on non-control data. The implementation section suggests how to apply this idea using new CPU instructions. The experiments section shows the evaluation and analyzes the results. Later we conclude that Secure Canary Word can efficiently prevent buffer-overflow attacks on non-control data.

## II. BACKGROUND KNOWLEDGE

To ease understanding the concept of Secure Canary Word, there are some fundamental concepts required.

### A. Buffer-overflow Attacks

According to the definition defined by Piromsopa and Enbody [6], buffer overflow is *the condition wherein the data transferred to a buffer exceeds the storage capacity of the buffer and some of the data overflows into another buffer, one that the data was not intended to go into*.

According to this definition, the definition of **buffer-overflow attacks**, used in this paper and [1], is *an attack caused by overflowing a buffer or writing beyond data boundary with data from another domain which results in malicious or unexpected behavior of a program.*

Buffer-overflow attacks can be classified into several types but two of them are mentioned in this paper.

*1) Stack overflow:* This attack is caused by overflowing a buffer in the stack area of a process. This area may contain control data and non-control data.

*2) Array indexing error:* This attack is caused by indexing beyond the boundary of array. Thus, in a theoretical way, the attackers can write to arbitrary memory location.

978-1-4244-3355-1/09/$25.00©2009 IEEE

In addition to classifying by a location as stated, they can be also divided by the target, control data or non-control data. The **control data**, like return addresses and function pointers, are generated by systems. The **non-control data**, like local variables and function arguments, are declared by users.

The example of buffer-overflow attacks on control data, stack-smashing attack, is presented in [1]. This paper emphasizes only on non-control data with the following example.

**Example 1: Simple buffer-overflow attack on non-control data**

```
#include <string.h>
int main(int argc, char *argv[]) {
    char b = 'X';
    char buffer[3];
    strcpy(buffer,argv[1]);
    printf("%c",b);
    return 0;
}
```

When a string ”AAAA” is passed into this program, the stack will be overflowed. Table I shows the layout of stack before and after the overflow.

TABLE I
STACK LAYOUT OF BEFORE AND AFTER EXECUTING PROGRAM IN EXAMPLE 1

| | | | | | |
|---|---|---|---|---|---|
| Before buffer overflow | Type | Buffer | | | b |
| | Value | - | - | - | X |
| After buffer overflow | Type | Buffer | | | b |
| | Value | A | A | A | A |

From Table I, the variable `b` can be modified by simply overflowing the buffer `buffer`. If this variable is going to be written to a password file, the result might be devastated.

Up to this point, we have learned the seriousness of buffer overflow. Next, we will review the protection schemes being proposed to date.

*B. Buffer-overflow Protection*

A taxonomy of buffer-overflow protection schemes is established by Piromsopa [7]. There are three broad categories (shown in Fig. 1). Static analysis schemes prevent the problem before deploying programs by parsing their source code and warning the programmers of potential threats. Dynamic solutions verify the integrity of data during the execution of programs by creating some metadata. Solutions in the isolation classes only reduce damage from the attacks.

Among these solutions, dynamic solutions sound interesting because they can dynamically detect and prevent the problem in run-time environment. Dynamic solutions can also be partitioned into many subclasses, such as address protection, input protection, bounds checking, and obfuscation.

This paper focuses on two schemes related to Secure Canary Word, namely Canary Word and Secure Bit. They represent address protection scheme and input protection scheme respectively.

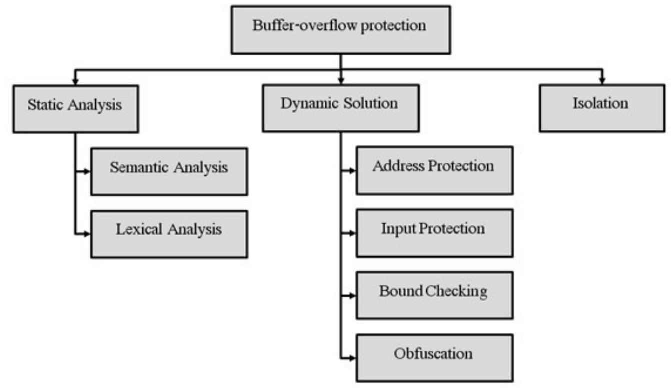



Fig. 1. Taxonomy of solutions against buffer-overflow attacks

*1) Canary Word:* Canary Word is an address protection scheme. The analogy comes from a Canary bird used for detecting toxic gas in a mine. Should a bird die, this would indicate a fatal condition in the mine. The assumption is that corrupting an address will also corrupt the adjacent data. Therefore, a Canary Word is placed as a metadata, adjacent to the address. However, replacing a Canary Word with a valid Canary value can bypass this scheme effortlessly [8]. To illustrate, we demonstrate bypassing a Canary Word in Example 1 (shown in Table II). If a string `"AAAAA"` is passed into the program to overflowing the variable `b` with a character `"A"`, checking the Canary Word will identify that the variable `b` is maliciously modified. On the other hand, passing a string `"AAA0A"` will change the value of the variable `b` without setting the alarm. In summary, the weak point of Canary Word is that there is no mechanism to protect the integrity of the Canary Word itself.

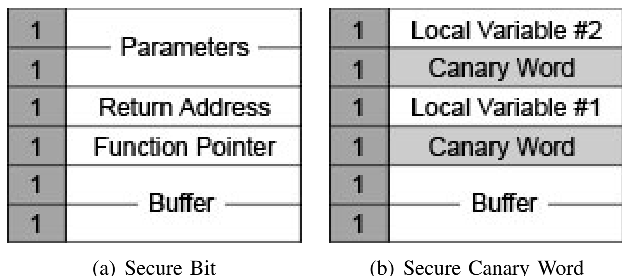
TABLE II
STACK LAYOUT OF THE CANARY WORD

| | | | | | | |
|---|---|---|---|---|---|---|
| Before buffer overflow | Type | Buffer | | | Canary Word | b |
| | Value | - | - | - | 0 | X |
| After buffer overflow | Type | Buffer | | | Canary Word | b |
| | Value | A | A | A | A | A |
| Bypassing Canary Word | Type | Buffer | | | Canary Word | b |
| | Value | A | A | A | 0 | A |

*2) Secure Bit:* Secure Bit [2] is a representative of the input protection schemes. It uses a hardware bit, called Secure Bit, associated with each memory word for tracking data passing across domains (process and kernel). The data with Secure Bit set is either input or derived from input and should not be used as control data. The associated instructions (e.g. call, return and jump) verify the data (the address) before using it. Consequently, the data that is not used by these instructions is ignored from being detected. Thus, it can be attacked. For instance, Table III shows the stack layout of Example 1 with Secure Bit. Although the Secure Bit of the variable `b` is set, no instruction can detect such condition.

## III. THE CONCEPT OF SECURE CANARY WORD

Secure Canary Word combines Secure Bit and Canary Word together. While Canary Word can protect an adjacent address; Secure Bit can provide the integrity for the Canary Word. For this reason, the Secure Bit is used to erase its weakness by detecting an overflow of the Canary Word. From Example 1, a string "AAA0A" cannot be used to bypass this scheme any longer, resulting from setting the Secure Bit of the Canary Word. To ease understanding, Table IV presents the stack layout of this approach in Example 1 and Fig. 2 shows the difference of Secure Bit and Secure Canary Word.




(a) Secure Bit (b) Secure Canary Word

Fig. 2. (a) Secure Bit vs (b) Secure Canary Word

## IV. IMPLEMENTATION

The efficiency evaluation of the previous implementation [1] exhibits that the implemented programs are much slower and their program sizes are also larger than the ordinary ones. Hence, the new implementation is expected to resolve this issue.

For this new implementation, new CPU instructions adding to Secure-Bit-enabled BOCHS [9], the Secure Bit implemented i368 emulator taken from [2], are different from the prior work because of adding verification mechanism in these instructions. Nonetheless, software must be adjusted to verify Secure Canary Word by using these instructions instead of the ordinary ones with no extra function in the software.

In summary, there are two modifications: one to the architecture and another to software.

TABLE III
STACK LAYOUT OF THE SECURE BIT

| | | Buffer | | | b |
|---|---|---|---|---|---|
| Before buffer overflow | Type | Buffer | | | b |
| | Value | - | - | - | X |
| | Secure Bit | 0 | 0 | 0 | 0 |
| After buffer overflow | Type | Buffer | | | b |
| | Value | A | A | A | A |
| | Secure Bit | 1 | 1 | 1 | 1 |

TABLE IV
STACK LAYOUT OF SECURE CANARY WORD

| | | | | | | |
|---|---|---|---|---|---|---|
| Before buffer overflow | Type | Buffer | | | Canary Word | b |
| | Value | - | - | - | 0 | X |
| | Secure Bit | 0 | 0 | 0 | 0 | 0 |
| After buffer overflow | Type | Buffer | | | Canary Word | b |
| | Value | A | A | A | 0 | A |
| | Secure Bit | 1 | 1 | 1 | 1 | 1 |

TABLE V
ORIGINAL AND DEFINED CPU INSTRUCTIONS

| Opcodes | | |
|---|---|---|
| Original | Defined | Description of instructions |
| 0x8B | 0x0F1B | MOV from memory to register |
| 0x3B | 0x0F1C | CMP memory with register |
| 0x39 | 0x0F1D | CMP register with memory |
| 0x83 | 0x0F1E | CMPL immediate value with memory |

### A. Adding New CPU Instructions

For Intel i386 architecture, we observed that there are four CPU instructions that access memory: a MOV instruction with opcode 0x8B, two CMP instructions with opcode 0x39 and opcode 0x3B and a CMPL instruction with opcode 0x83. As a result, four unused opcodes in the i386 architecture are used for replicating the mechanism of these instructions. The selected opcodes are shown in Table V.

These new instructions work like the original ones but they also verify the Secure Canary Word of the address in memory. This solution is expected to considerably reduce the execution time because the overhead is buried deeply in the hardware instead. There are three steps of verification added to these instructions.

*1) Checking the addressing form:* Analyzing the instruction format of the Intel i386 architecture (see Fig. 3 [10]), there are MODR/M and SIB bytes using to indicate the addressing mode (form) and information required to calculate the location of Secure Canary Word. In this step, we must identify such information.

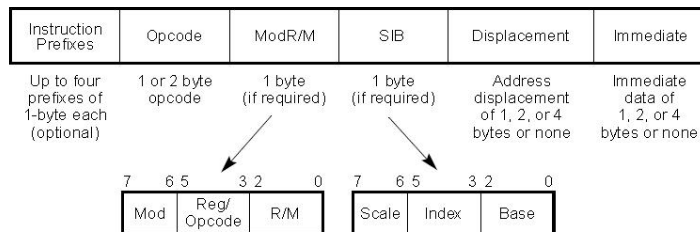



Fig. 3. Intel architecture instruction format

*2) Calculating the address of the adjacent Secure Canary Word:* The Secure Canary Word is adjacent to the address of associated data. However, it may either be in the upper or the lower address. After knowing the addressing form, we can calculate the address of Canary Word using information in the MOD field, the R/M field and the displacement field. If the value in the MOD field is 00 and the value in the R/M field is 100 (in binary), accessing to a byte-array variable is assumed.

Assuming that the base address of the array is A, the address of the Secure Canary Word is A-1. In other cases, assuming that the address of variable is X, if the displacement is positive, the address of the Secure Canary Word is X+4; otherwise, it is X-1.

*3) Verifying the Secure Bit of the Secure Canary Word:* The Secure Bit associated with the Secure Canary Word will be considered. If the Secure Bit is cleared, the processor will continue working regularly. In contrast, it will raise a segmentation fault exception if the Secure Bit is set.

### B. Software Modifications

In addition to adding new CPU instructions, software is necessary to be adjusted by replacing original instructions with the new instructions using in-line assembly and declaring canWord character variable as Secure Canary Word. Two examples of C language programs implementing new MOV and CMP instructions are shown below.

*Example 2: MOV instruction.*

This is the original program which moves value from variable c to variable e.

```
int main(int argc, char* argv[]){
    int c=5;
    int e=3;

    e=c;
    printf("%d",e);
    return 0;
}
```

The Secure-Canary-Word-implemented program of this example will be:

```
int main(int argc, char* argv[]){
    int c=5;
    char canWord_c;
    int e=3;
    char canWord_c;

    asm volatile
    ("
        .long 0xFC451B0F;
        mov %%eax,%0;
    "
    :"=m"(e)
    :/*no input*/
    :"%eax"
    );
    printf("%d",e);
    return 0;
}
```

From the code .long 0xFC451B0F; above, the instruction format is reversely arranged because the Intel i386 architecture uses little-endian format. The opcode 0x0F1B is MOV instruction. The ModR/M byte 0x45 is the addressing form of the register EAX (Accumulator register) and the effective address from register EBP (Extended Base Pointer register) with the 8-bit displacement 0xFC which is the address of the variable c. (see Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte in [10]) As a result, the instruction 0x0F1B45FC is equivalent to MOV 0xfffffffc(%ebp),%eax in assembly langauge.

*Example 3: CMP instruction.*

This is the original program which assigns ”1” to variable e, if variable c is equal to variable z, or assigns ”0” to variable e, if variable c is not equal to variable z.

```
int main(int argc, char* argv[]){
    int c=5;
    int z=9;
    int e=3;

    if(c==z)
        e=1;
    else
        e=0;
    printf("%d",e);
}
```

The Secure-Canary-Word-implemented program of this example will be:

```
int main(int argc, char* argv[]){
    int c=5;
    char canWord_c;
    int z=9;
    char canWord_z;
    int e=3;
    char canWord_e;

    asm volatile
    ("
        .long 0xFC451B0F;
        .long 0xF4451C0F;
        jne FALSE;
      TRUE:
       movl $0x1,%0;
        jmp END;
      FALSE:
       movl $0x0,%0;
      END:
    "
    :"=m"(e)
    :/*no input*/
    :"%eax"
    );
    printf("%d",e);
}
```

From the code .long 0xF4451C0F; above, the instruction 0x0F1C45F4 is equivalent to CMP 0xfffffff4(%ebp),%eax in assembly langauge.

The limitation in this implementation is that only the local variables are protected. The arguments of the functions are still omitted, some of which are relocated by the compiler. The main reason of this limitation is that we are unable to declare the Secure Canary Word between each argument.

## V. Experiments

The experiments are conducted in the improved version of Secure-Bit-enabled BOCHS emulator running Linux (Red Hat 6.2) and using the GCC compiler (version egcs-2.91.66). We evaluate our architecture in two aspects: protection and

TABLE VI

THE EXECUTION TIME

| Example programs | Execution time (Seconds) | | |
|---|---|---|---|
| | Original | Previous | New |
| Bubble sort | 2.106 | 27.885 | 2.316 |
| Ratio | 1 | 13.241 | 1.100 |
| Quick sort | 0.029 | 0.574 | 0.064 |
| Ratio | 1 | 19.793 | 2.207 |
| Binary search using AVL tree | 0.359 | 1.5755 | 0.4725 |
| Ratio | 1 | 4.389 | 1.316 |

TABLE VII

THE PROGRAM SIZE

| Example programs | Size (Bytes) | | |
|---|---|---|---|
| | Original | Previous | New |
| Bubble sort | 11,910 | 12,705 | 12,868 |
| Different (%) | - | +6.675 | +8.044 |
| Quick sort | 12,137 | 13,044 | 12,536 |
| Different (%) | - | +7.473 | +3.287 |
| Binary search using AVL tree | 13,033 | 14,340 | 13,980 |
| Different (%) | - | +10.028 | +7.266 |

efficiency. In particular, we focus on comparing the efficiency with the prior implementation [1].

*A. Protection*

Like the previous work, this implementation detects buffer-overflow attacks on stack on both control and non-control data. It still suffers from Array indexing error on non-control data because they use the common concept of Secure Canary Word.

*B. The Efficiency*

To measure the efficiency of Secure Canary Word, there are three simple programs similar to the previous work: bubble sort, quick sort and binary search using AVL tree. The comparison of the execution time and the program size is made between the previously and newly Secure-Canary-Word-implemented programs (shown in Table VI and Table VII respectively) using the original programs (without Secure Canary Word) as baselines and already subtracting the overhead of reading 1,000 records of data from a file into memory. All results are the average value of 10 executions. The program of binary search using AVL tree is tested using both found and not-found case. All newly-implemented programs are written in C language with in-line assembly for the newly-added instructions. Besides, there are some minor adjustments to the prior programs to produce a comparable result.

According to the Table VI, the results are dramatically different from the previous owing to the specification of the tested computer. The prior one uses Intel Pentium 4 CPU (2.0 GHz) with 1.5 GB DDR RAM but this paper uses Intel Core 2 Duo CPU (2.2 GHz) with 2 GB DDR2 RAM. In addition, we also subtract the overhead of initializing data (reading data from a file) in this work. The results are numerously dissimilar from the former results.

TABLE VIII

THE PROGRAM SIZE OF BUBBLE SORT

| Optimization flags | Size (Bytes) | | |
|---|---|---|---|
| | Original | Previous | New |
| No flag | 11,910 | 12,705 | 12,868 |
| Different (%) | - | +6.675 | +8.044 |
| `-Os` | 11,638 | 12,545 | 12,756 |
| Different (%) | - | +7.793 | +9.606 |
| `-O3` | 11,638 | 12,545 | 12,756 |
| Different (%) | - | +7.793 | +9.606 |
| `-g` and `-O3` | 16,086 | 17,321 | 17,452 |
| Different (%) | - | +7.677 | +8.492 |

TABLE IX

THE PROGRAM SIZE OF QUICK SORT

| Optimization flags | Size (Bytes) | | |
|---|---|---|---|
| | Original | Previous | New |
| No flag | 12,137 | 13,044 | 12,536 |
| Different (%) | - | +7.473 | +3.287 |
| `-Os` | 11,961 | 12,756 | 12,520 |
| Different (%) | - | +6.647 | +4.674 |
| `-O3` | 11,945 | 12,852 | 12,520 |
| Different (%) | - | +7.593 | +4.814 |
| `-g` and `-O3` | 17,237 | 18,460 | 17,412 |
| Different (%) | - | +7.095 | +1.015 |

Moreover, the programs are compiled without any optimization flag turned on. The following results are from the programs with some optimization flags turned on, such as `-Os` (Optimizations designed to reduce code size), `-O3` (Optimizations designed to reduce code size and execution time at level 3 with the `-finline-functions`) and `-g`. For more information, please also see the GCC man page [11]. To make a clear comparison, they are shown in Table VIII, Table IX and Table X for each program. They only affect the program size since there are many unknown opcodes (the compiler cannot optimize the newly-implemented programs).

*C. Evaluation*

It is obviously seen that nearly all results of the newly-implemented program are better than the previous implemen-

TABLE X

THE PROGRAM SIZE OF BINARY SEARCH USING AVL TREE

| Optimization flags | Size (Bytes) | | |
|---|---|---|---|
| | Original | Previous | New |
| No flag | 13,033 | 14,340 | 13,980 |
| Different (%) | - | +10.028 | +7.266 |
| `-Os` | 12,601 | 13,764 | 13,932 |
| Different (%) | - | +9.229 | +10.563 |
| `-O3` | 13,017 | 14,980 | 13,932 |
| Different (%) | - | +15.080 | +7.029 |
| `-g` and `-O3` | 25,717 | 28,548 | 23,036 |
| Different (%) | - | +11.008 | -10.425 |

tation. Especially, the execution time is all decreased many times. However, the program of bubble sort in Table VII is an exception because using in-line assembly in the short program may make it a little bit larger than writing simple C program.

For the programs with optimization flag `-O3` turned on, the sizes of all newly-implemented programs are equal to the programs with optimization flag `-Os` turned on. One of the reasons may be that these optimizations do not use `inline` specifier for optimization like the prior work. In addition, the optimization flag `-g` produces debugging information such as the program code (see also on the GCC man page [11]), causing extra unnecessary size found in the result of the previous work [1].

There is another solution to improve the Secure-Canary-Word approach by eliminating the limitation in this implementation. The solution is to modify a compiler. Such modification can make the Secure Canary Word transparent to programmers and will increase protection by including the protection of functions' arguments in software.

## VI. Conclusion

From the proof of Secure-Canary-Word concept in the former paper [1], the work on this paper demonstrates how to improve the performance of this concept using hardware optimization. Therefore, Secure Canary Word can prevent buffer-overflow attacks on non-control data with few performance penalties and minor storage requirement. Nevertheless, this number can still be improved. There exist other optimizations capable of enhancing Secure Canary Word in a real practice.

## Acknowledgment

The authors would like to thank Graduate School of Chulalongkorn University for giving a Chulalongkorn University Graduate Scholarship to Commemorate the $72^{nd}$ Anniversary of His Majesty King Bhumibol Adulyadej.

## References

[1] K. Piromsopa and S. Chiamwongpaet, "Secure bit enhanced canary: Hardware enhanced buffer-overflow protection," *Network and Parallel Computing, 2008. NPC 2008. IFIP International Conference on*, pp. 125–131, Oct. 2008.

[2] K. Piromsopa and R. J. Enbody, "Secure bit: Transparent, hardware buffer-overflow protection," *Dependable and Secure Computing, IEEE Transactions on*, vol. 3, no. 4, pp. 365–376, Oct.-Dec. 2006.

[3] C. Schmidt and T. Darby, "The what, why, and how of the 1988 internet worm," http://www.snowplow.org/tom/worm/worm.html.

[4] US-CERT, "Us-cert vulnerability notes," http://www.kb.cert.org/vuls/bypublic.

[5] P. Bright, "The sky isn't falling: a look at a new vista security bypass - ars technica," http://arstechnica.com/news.ars/post/20080811-the-sky-isnt-falling-a-look-at-a-new-vista-security-bypass.html, August 2008.

[6] Webopedia, "What is buffer overflow?" http://www.webopedia.com/TERM/B/buffer_overflow.html.

[7] K. Piromsopa, "Secure bit: Buffer-overflow protection," Ph.D. dissertation, Michigan State University, 2006.

[8] Bulba and Kil3e, "Bypassing stackguard and stackshield," *Phrack Magazine*, vol. 10, no. 56, 2000.

[9] T. R. BUTLER, "bochs: The open source ia-32 emulation project (home page)," http://bochs.sourceforge.net/, 2006.

[10] I. Corporation, "Intel(r) architecture software developer's manual, volume 2: Instruction set reference manual," http://www.intel.com/design/intarch/manuals/243191.htm.

[11] Die.net, "gcc(1): Gnu project c/c++ compiler - linux man page," http://linux.die.net/man/1/gcc.