LETTER *Special Section on Cryptography and Information Security*

Redundancy in Instruction Sequences of Computer Programs

Kazuhiro HATTANDA[†*], ***Nonmember and*** **Shuichi ICHIKAWA**[†,††a)], ***Member***

SUMMARY There is redundancy in instruction sequences, which can be utilized for information hiding or digital watermarking. This study quantitatively examines the information capacity in the order of variables, basic blocks, and instructions in each basic block. Derived information density was 0.3% for reordering of basic blocks, 0.3% for reordering instructions in basic blocks, and 0.02% for reordering of global variables. The performance degradation caused by this method was less than 6.1%, and the increase in the object file size was less than 5.1%.

key words: *information hiding, reordering, basic block, digital watermark*

1. Introduction

There are many instruction sequences that correspond to a program, any of which serves users equally as long as they are functionally equivalent. In other words, there is redundancy in constructing instruction sequences of a program. Such redundancy can be utilized for watermarking or information hiding [1].

Davidson and Myhrvold [2] invented a method to generate a signature on a computer program. They pointed out that the basic blocks of a computer program can be reordered arbitrarily without changing the behavior of the program if *jump* instructions are correctly inserted and maintained. Their idea is to embed a signature as a reordered sequence of basic blocks. Davidson's scheme was quantitatively evaluated by Hattanda and Ichikawa [3].

El-Khalil and Keromytis [4] mentioned that the order of arguments and the register allocation choice are usable for information hiding. They also stated that they can encode information by ordering functions and various tables in an object file, and that the estimated encoding rate of this method is 2.8% of the code size. Here, it should be noted that the redundancy in an *object file* is not equivalent to the redundancy in an *object code*.

Though other objects are similarly usable to embed a signature, no quantitative evaluations have been reported to date. In this study, we concentrated on evaluating the redundancy in instruction sequences, particularly in the order of variables and instructions.

It is very difficult to count all redundancies, since there are many options. Therefore, in this study, only four options are examined: (1) reordering global variables, (2) reordering local variables, (3) reordering basic blocks, and (4) reordering instructions in each basic block. The performance and object size of benchmark programs are also examined before and after the reordering. All measurements were made with ELF object files for Intel x86 architecture [5], which were generated from C programs using GCC 2.95.3 and binutils 2.13.

2. Reordering of Variables

Generally, users are unaware of the addresses of variables. Therefore, we can construct functionally-equivalent programs by reordering the variables in the main memory. Since there are $n!$ options to arrange n elements, we can generate $n!$ functionally-equivalent instruction sequences with n variables.

In C language, variables are categorized into global and local variables. Global variables are further categorized into sub-categories; external variables, uninitialized variables, and initialized variables. Uninitialized and initialized variables are registered in *.bss* and *.data* sections, respectively, and thus can be reordered by changing the order of definition in assembly files after compilation. To reorder external variables, we had to add a new feature to the linker (*ld*) to arbitrarily change the addresses of external variables.

Local variables are allocated on a stack or on registers. Local variables on a stack are accessed via the EBP register with the offset values that are assigned by compiler. Therefore, it is possible to reorder local variables on a stack by adding a new feature to the C compiler. Further redundancy in register allocation could also be utilized by enhancing the C compiler. However, this redundancy was not examined in this work, leaving it for future studies.

3. Reordering of Instructions

An instruction sequence is divided into basic blocks, each of which is a sequence of instructions that is executed straight from beginning to end. The addresses of basic blocks can be arbitrarily reordered without changing the behavior of the program if the order of execution is maintained by adding unconditional jump instructions properly (Fig. 1). This re-

Manuscript received March 22, 2005.
Manuscript revised June 18, 2005.
Final manuscript received August 4, 2005.

[†]The authors are with the Department of Knowledge-based Information Engineering, Toyohashi University of Technology, Toyohashi-shi, 441-8580 Japan.
[††]The author is with the Intelligent Sensing System Research Center, Toyohashi University of Technology, Toyohashi-shi, 441-8580 Japan.
[*]Presently, with Hitachi Information and Control Systems, Inc.
a) E-mail: ichikawa@tutkie.tut.ac.jp

DOI: 10.1093/ietfec/e89–a.1.219

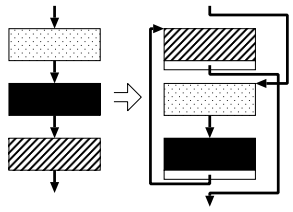

Fig. 1 Reordering basic blocks.

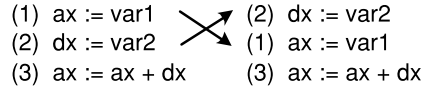

Fig. 2 Reordering instructions in each basic block.

dundancy may be used to embed a digital signature into a program [2]. The obvious drawbacks of this method are the increase in program size and the performance degradation caused by additional jump instructions. These overheads were quantitatively evaluated and reported by the authors using MIPS architecture [3].

In this current study, the authors developed a program that counts the redundancy in the order of basic blocks. This program reads an assembly file, divides it into basic blocks, and reorders these basic blocks, while reporting the degree of redundancy.

It is also possible to reorder instructions in a basic block if the resulting instruction sequence is functionally equivalent to the original one. For example, in Fig. 2, the instructions (1) and (2) are exchangeable, because both sequences yield the same result.

Let $Ref[x]$ and $Sto[x]$ be the sets of read operands and write operands of an instruction x, respectively. Here, $Ref[x]$ and $Sto[x]$ include both *explicit* and *implicit* operands. Explicit operands are designated as arguments of an instruction, while implicit operands are referred or updated without explicit designation. For example, `PUSH EAX` [5] instruction pushes the value of EAX register onto stack. Though its explicit operand is an EAX register, `PUSH` instruction implicitly refers to SS (stack segment) and ESP (stack pointer) registers to modify memory and ESP register. Consequently, the read and write operands of `PUSH EAX` are $\{EAX, SS, ESP\}$ and $\{ESP, memory\}$, respectively.

Instructions x and y are independent if the following condition holds: $Ref[x] \cap Sto[y] = Sto[x] \cap Ref[y] = Sto[x] \cap Sto[y] = \phi$. If two adjacent instructions are independent, these two instructions are mutually exchangeable. In this study, another program was developed to reorder instructions in each basic block and to count possible combinations of functionally equivalent instruction sequences in an assembly file.

It should be noted that the instructions (1) and (2) in Fig. 3 are *not* exchangeable by the rule in the previous paragraph, because both instructions are arithmetic instructions that implicitly affect status flags in the EFLAGS register; i.e., $Sto[(1)] \cap Sto[(2)] \neq \phi$. Actually, status flags are immediately overwritten by the next instruction (3), and therefore (1) and (2) could be exchangeable. Although flow analysis is required to recognize such redundancy, we left this issue for future studies. Hence, there are still some redundancies left in the order of instructions in each basic block.

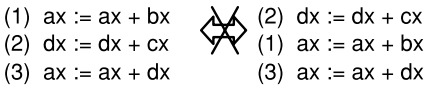

Fig. 3 Unexchangeable instructions.

4. Experiments and Results

The options to arrange n items are $n! = O(n^n)$, which is difficult to handle when n is large. Thus, PPS (Partial Permutation Scheme) [6] was adopted in this study. Items are divided into chunks, each of which includes 6 items (6! = 720 options). Odd items were excluded from measurements. The redundancy derived from m chunks is thus 720^m; i.e., m chunks can carry maximally $m \log_2 720 \approx 9.49m$ bit of information (*information capacity*).

Table 1 lists some open source programs in C language which were selected for experiments in this study. As readily seen from Table 1, the information capacity of basic block reordering and instruction reordering is larger than that of variable reordering, because the number of instructions is usually larger than the number of variables. In optimized object codes, local variables are allocated to registers, and practically no information capacity is derived from local variables. It is thus necessary to examine the redundancy of register allocation, but that is also left to future studies.

Figures 4 and 5 display the relationships between object file size and the information capacity in reordering basic blocks and instructions in basic blocks, respectively. Information density, which is defined by information capacity divided by the corresponding object file size, was estimated to be 0.3% for reordering of basic blocks (Fig. 4), 0.3% for reordering of instructions in basic blocks (Fig. 5), and 0.02% for reordering of global variables.

These techniques may have some negative impact on the performance and size of the instruction sequences. The overhead of reordering depends on the order taken; i.e., it depends on the data embedded in the program. Thus, we measured the average performance and size of 100 reordered sequences, each of which embeds a random number. Target programs are three simple benchmark programs: dhrystone, linpack, and whetstone.

In reordering basic blocks, a maximally 6.1% performance degradation was observed on an Intel Xeon 2.8 GHz system for three benchmark programs; the performance degradation was less than 3.1% in the other three reordering techniques. Basic block reordering incurs (maximally) a 5.1% increase in object file sizes for three benchmark programs; no increase was observed in the other three reordering techniques.

Table 1 Information capacity in sample programs with four reordering methods.

Program	Compile option	#Func.	#Line	Object file size [byte]	Information capacity [byte]			
					Global	Local	Basic block	Instruction
dhry_1.c	-DHZ=100 -DTIME	6	385	7464	2.37e+00	1.19e+00	2.37e+01	1.54e+01
dhry_2.c		6	192	1936	0.00e+00	0.00e+00	7.12e+00	4.29e+00
linpackc.c	-DDP -DUNROLL	12	907	15856	2.37e+00	4.75e+00	6.53e+01	6.84e+01
whetstone.c		4	433	5984	1.19e+00	4.75e+00	1.90e+01	1.79e+01
dhry_1.c	-DHZ=100 -DTIME -O2	6	385	7064	2.37e+00	0.00e+00	2.02e+01	1.35e+01
dhry_2.c	-O2	6	192	1600	0.00e+00	0.00e+00	5.93e+00	1.62e+00
linpackc.c	-DDP -DUNROLL -O2	12	907	11848	2.37e+00	0.00e+00	5.46e+01	5.32e+01
whetstone.c	-O2	4	433	4096	1.19e+00	0.00e+00	1.42e+01	1.32e+01
ed/main.c	default option (-O)	26	1684	38704	9.49e+00	0.00e+00	1.29e+02	3.21e+01
ed/regex.c	default option (-O)	26	5171	28428	0.00e+00	0.00e+00	1.40e+02	1.19e+02
bzip2.c	default option (-O2)	43	2103	31936	3.56e+00	0.00e+00	7.24e+01	5.10e+01
gzip.c	default option (-O)	23	1744	28132	9.49e+00	0.00e+00	5.58e+01	4.76e+01
gcc.c	default option (-O2)	50	5840	77448	1.90e+01	0.00e+00	2.56e+02	2.50e+02
ldmain.c	default option (-O2)	20	1376	20512	1.19e+00	0.00e+00	3.68e+01	3.66e+01
ldlang.c	default option (-O2)	126	5525	48128	5.93e+00	0.00e+00	1.44e+02	1.06e+02

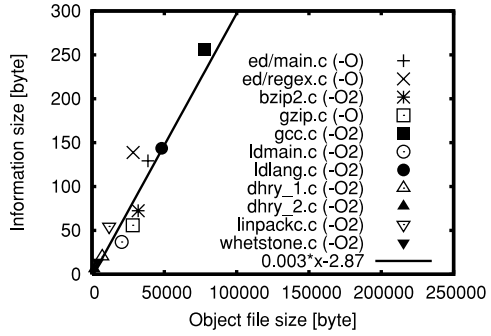

Fig. 4 Information density by reordering basic blocks.

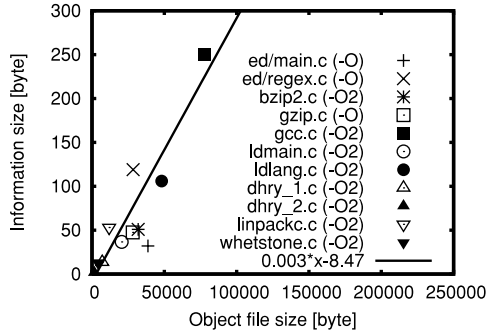

Fig. 5 Information density by reordering instructions in each basic block.

5. Conclusion

The encoding rate of our techniques can be improved, as stated in Sects. 3 and 4. Though the information density by variable reordering is smaller than that by instruction reordering, it is still important, because these two methods are mutually independent. Two independent techniques might be used together to avoid forgery of the digital signature [6].

El-Khalil and Keromytis [4] presented a system, named Hydan, which embeds information in x86 binaries. In their scheme, an instruction sequence is selected from a predefined set of functionally equivalent sequences. For example, "add %eax, $50" and "sub %eax, $-50" are functionally equivalent, and thus one bit of information could be embedded by selecting one of them. They reported that their encoding rate is 0.9% of the code size, which corresponds to about 0.7% of the object file size. This rate is almost comparable to ours. The scheme of Hydan is basically different from ours, because we only change the arrangement of objects without replacing them. This suggests that these two schemes could be used together to increase information density or encoding rates.

Cooperation with other techniques should also be examined for higher encoding rates. There are many other schemes to embed information in computer programs. More extensive survey and quantitative investigations are required.

Acknowledgment

This work was partially supported by a Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science (JSPS). Support for this work was also provided by the 21st Century COE Program "Intelligent Human Sensing" from the Ministry of Education, Culture, Sports, Science and Technology.

References

[1] C.S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—Tools for software protection," IEEE Trans. Softw. Eng., vol.28, no.8, pp.735–746, Aug. 2002.

[2] R.I. Davidson and N. Myhrvold, "Method and system for generating and auditing a signature for a computer program," U.S. Patent 5,559,884, Sept. 1996.

[3] K. Hattanda and S. Ichikawa, "The evaluation of Davidson's digital signature scheme," IEICE Trans. Fundamentals, vol.E87-A, no.1, pp.224–225, Jan. 2004.

[4] R. El-Khalil and A.D. Keromytis, "Hydan: Hiding information in program binaries," Proc. 6th Int'l Conf. Information and Communications Security (ICICS'04), LNCS 3269, pp.187–199, Springer, 2004.

[5] Intel Corporation, Intel Architecture Software Developer's Manual, 1997, http://www.intel.com/

[6] S. Ichikawa, H. Chiyama, and K. Akabane, "Redundancy in 3D polygon models and its application to digital signature," J. WSCG, vol.10, no.1, pp.225–232, 2002.