EVALUATION OF CUSTOM VIRTUAL MACHINE INSTRUCTION SET EMULATOR

Jozef KOSTELANSKÝ, Ľubomír DEDERA

Abstract: The main goal of the article is to evaluate performance characteristics of a custom virtual machine instruction set emulator. The instruction set has been designed as part of research aimed at utilization of custom virtual machines in the area of obfuscation techniques for software protection and malware detection, with the aim to efficiently implement the particular algorithm (CRC16). In the paper we compare performance characteristics of two implementations of the CRC16 algorithm – in the emulated custom virtual machine instruction set and the direct C-to-x86-compiled executable. The aim is to show that the emulation process of such a simple virtual machine has only minor influence on execution time in comparison with the C-to-x86-compiled code.

Keywords: Instruction set; Virtual machine; Performance; Compilers; Time measurements.

1 INTRODUCTION

Emulators allows to run code written for one architecture on another architecture. There are many emulators. For example, QEMU (1), Android Emulator (2) and many more. Emulation is also a promising method of software obfuscation (3). In (4) was custom reconfigurable instruction set proposed. The main goal of that custom virtual machine instruction set is to research static properties of reconfigurable instruction set emulators. It concludes with presenting findings that, using presented approach, it is possible to generate binaries with the same dynamic properties, but with different static properties. After reviewing static properties of the proposed emulator, the focus is switched to its dynamic properties. One of the requirements on software obfuscators from dynamic point of view is code efficiency or performance efficiency. It means that obfuscated code should not run much slower than the original code (5). Time measurements are as well used as methods for anomaly detections (6), (7), (8). The goal of this paper is to evaluate performance impact of the proposed custom virtual machine instruction set emulator running on top of the x86 architecture operating system with comparison towards direct x86 implementation. For this evaluation, CRC-16 algorithm implementation is used.

2 VIRTUAL MACHINE INSTRUCTION SET DESIGN

In (4) a custom virtual machine instruction set was proposed. Requirements were focused on simple and extensible instruction set creation. However, it was insufficient for the CRC16 (9) algorithm implementation. That is why the whole emulator has been rewritten specifically for CRC16 implementation with the following changes:

- **Registers –** The number of general-purpose registers has been increased from eight to sixteen registers.
- **Instructions –** The previously proposed instruction set lacked instructions that are necessary for the CRC16 implementation.

Here we sum up the updated RISC-like architecture emulator attributes:

- **Architecture** – virtual machine template design is based on von Neumann architecture with shared memory between data and instructions.
- **Memory size** – virtual machine template design memory size is 32-bit. It consists of 16-bit addresses pointing to 16-bit values.
- **Instruction type** – virtual machine template design uses instructions of fixed length (16 bits per instruction).
- **Registers** – virtual machine template design uses 16 registers. Fifteen of them are general purpose registers. In addition, there is the instruction pointer register.
- **Flags** – virtual machine template design uses 3 flags. It updates them after every instruction execution. The flags are – FLAG_ZERO, FLAG_POSITIVE and FLAG_NEGATIVE. Their names describe theirs purpose.
- **Instructions** – custom virtual machine instruction set consists just of instructions necessary to implement the CRC16 algorithm. Here is a list of previously implemented instructions for reference:
- LA is an instruction used for loading 8-bit value into registers, for example LA R0, 0x00000001 will load value 1 into register R0. This instruction is used for loading smaller constant values or addresses pointing to bigger values in memory.
- LV is an instruction used for loading value pointed by address in one register into another register, for example LV R0, R1 will load value from address where is register R1 pointing into register R0.

- XOR is an instruction used for performing logical xor operation between 2 registers and writing output to another register; for example, XOR R0, R1, R2 will do the following operation: R0 := R1 XOR R2. This instruction is used in addition to XOR also for setting the registry values to 0.
- ADD is an instruction used for adding 2 registers and writing output to another register; for example, ADD R0, R1, R2 will do following operation: R0 := R1 + R2.
- MOV is an instruction used for copying a value from one register to another; for example, MOV R0, R1 will copy value from register R1 into register R0.
- CMP is an instruction used for comparing 2 registers; for example, CMP R0, R1 will compare value in register R0 with value in register R1 and based on the result, is will update flags. If R0 equals to R1 then the instruction will set the FLAG_ZERO. If R0 is greater than R1, the instruction will set the FLAG_POSITIVE and if R0 is smaller than R1, the FLAG_NEGATIVE is set.
- JNZ is an instruction used for conditional branching; for example, JNZ R6 will do conditional jump to the address in register R6 if FLAG_ZERO is not set, and otherwise, it will continue with another instruction. Together with CMP instruction can be used for cycle implementation.
- PRINT is an instruction used for printing the value stored in a register; for example, PRINT R0 will print the value in register R0 on the standard output.
- HALT is an instruction used for halting the custom virtual machine down. If the interpreter encounters this instruction, it will shut down.

Newly added instructions:

- SUB is an instruction used for subtraction; for example, SUB R0, R1, R2 will subtract value in R2 from R1 and write result to R0.
- JNE is an instruction used for conditional branching; for example, JNE R0 will do conditional jump to address stored in register R0 if FLAG_ZERO is not set, and otherwise, it will continue with next instruction. Together with CMP instruction, can be used for cycle implementation.
- JB is an instruction used for conditional branching; for example, JB R0 will do conditional jump to address stored in register R0 if FLAG_NEGATIVE is set, and otherwise, it will continue with next instruction. Together with CMP instruction, can be used for cycle implementation.
- AND is an instruction used for AND logical operation between two registers; for example, AND R0, R1, R2 will do following operation R0 := R1 AND R2.
- OR is an instruction used for OR logical operation between two registers; for example, OR R0, R1, R2 will do following operation R0 := R1 OR R2.
- LSHIFT is an instruction used for bitwise left shift operation; for example, LSHIFT R0, R1, R2 will do following operation R0 := R1 << R2.
- RSHIFT is an instruction used for bitwise right shift operation; for example, RSHIFT R0, R1, R2 will do following operation R0 := R1 >> R2.

3 PROPOSAL OF THE MEASURE

The goal of this paper is to measure execution time needed to perform CRC16 checksum computations. A cyclic redundancy check (CRC) is an error-detecting code that is usually being used for accidental data changes detection. It can be also used for intentional data changes detection as well (9).

CRC is typically implemented using logical shifts for polynomial divisions (10). This kind of implementation was used for our evaluation of the proposed architecture because of its higher computation complexity than the implementation used originally in (3). There are also more effective approaches for CRC computation such as Computation of Cyclic Redundancy Checks via table look-up (11).

CRC16 is easy to implement and can be used for changes detection. The same CRC16 algorithm was implemented in pure C and in the proposed custom virtual machine instruction set.

3.1 CRC16 ALGORITHM DETAILS

There are many CRC specifications and implementations. Description of a specific CRC code needs a characterization by defining a division polynomial. An M-bit long CRC is based on a primitive polynomial of degree *M*, called a generator polynomial. For example, CCITT has chosen the following polynomial: $x^{16} + x^{12} + x^5 + 1$.

This polynomial can also be expressed like 1 0001 0000 0010 0001.

The CRC computation algorithm for an input word I and a given generator polynomial G of degree D is as follows:

First, multiply I by X^M. In binary, it results in adding M zero bits to I. Second, divide G into Ix^M. Since division is done on binary level, all of the subtractions are done as modulo 2. The Modulo 2 subtraction operation is the same as the logical exclusive or (XOR) operation. Third, ignore the quotient. Fourth, the remainder is part of CRC. Let's call it C. C will be a polynomial of degree M-1. If C is of higher degree, the division process has not been finished yet (12). All of these operations can be defined using logical shifts and exclusive OR logical operations.

3.2 CRC16 CUSTOM ARCHITECTURE IMPLEMENTATION

The CRC-16-ANSI specification has been chosen to be used for the purpose of this research. This implementation is specified by the 0x8005 ($x^{16} + x^{15} + x^2 + 1$) (1 1000 0000 0000 0101) polynomial. Below is presented the CRC16 implementation in the proposed custom virtual machine instruction set.

```
LA R0, 0x2
LA R1, 0x4
LA R3, 0x1
LV R2, R3
XOR R3,R3,R3
XOR R4,R4,R4
XOR R5,R5,R5
LA R9, 12
LA R10, 15
LA R11, 1
LA R12, 8
RSHIFT R5,R3,R10
LSHIFT R3,R3,R11
LV R15,R0
RSHIFT R6,R15,R4
AND R6,R6,R11
OR R3,R3,R6
ADD R4,R4,R11
LA R13, 25
CMP R4,R12
JB R13
LA R4, 0
ADD R0, R0, R11
SUBSTRACT R1,R1,R11
LA R13, 29
CMP R5,R11
JB R13
XOR R3,R3,R2
CMP R14, R1
JB R9
LA R9, 34
LA R6, 0
LA R4, 16
RSHIFT R5, R3, R10
LSHIFT R3, R3, R11
LA R13,40
CMP R5,R11
JB R13
XOR R3,R3,R2
ADD R6,R6,R11
CMP R6, R4
JNE R9
LA R7,0
LV R6,R14
LA R8, 0x0001
LA R9, 47
AND R10, R6, R3
LA R13, 52
CMP R10,R11
JB R13
OR R7,R7,R8
RSHIFT R6,R6,R11
LSHIFT R8,R8,R11
CMP R6, R14
JNE R9
PRINT R7
HALT
```

3.3 COMPILATION DETAILS

Both programs were compiled on Linux, Ubuntu 16.04 with 4.4 kernel using gcc compiler v7.5.0 (13) for Linux operating system. Different optimization settings were used for measurements:

- Default configuration;
- - O1;
- - O2;
- - O3;
- - Ofast.

Both pure C implementation and custom virtual machine-based implementation used bitwise shift-based implementations and not precalculated table implementation.

Based on the online gcc compiler documentation (14), the influence of used optimization flags on compilation process is as follows. The -O1 flag turns on optimization during compilation process. Compiler tries to decrease the code size and more notably, it tries to decrease the execution time. It uses the basic set of optimizations but does not use any optimizations that would probably take a lot of compilation time. The -O2 flag instructs the compiler to optimize even more. Using this flag, the gcc compiler performs almost all supported optimizations with exception of those, which outcome in space-speed compromises. The usage of -O3 flag means to turn on further optimizations. It uses all optimizations specified by -O2 flag and some other ones. The largest set of optimizations is applied, when the Ofast flag is set. It uses all -O3 optimizations, but

also it enables optimizations that are not valid for all standard-compliant programs.

Totally, three sets of measurements were made. As an input, different subsets of ROCKOU.txt (15) wordlist have been used. The input subsets consist of 100, 1000, 10 000, 100 000 and 1 000 000 words. The measurements were done on Intel Core i7 with 8 GB of RAM memory.

For execution time measurements, *Time* (16) tool was used. Its results consist of the elapsed time in the User mode, Kernel mode and the real elapsed time. For the purpose of this paper measurements, only real elapsed time was taken into consideration.

4 RESULTS

As it has been mentioned in the previous chapter, totally, three distinct sets of measurements were taken. In all from three distinct sets of measurements, 50 separate measurements were done. All chosen inputs were run against whole set of generated binaries using entire flags.

Tab. 1 Results of measurements of execution time

	c	c -O1	c -O2	c -O3	c -Ofast	vm	vm -O1	vm -O2	vm -O3	vm -Ofast
1000000	32:39.05	32:21.00	32:54.01	32:33.01	32:24.05	33:58.01	32:53.10	32:42.04	32:48.03	32:40.03
100000	03:15.01	03:18.05	03:17.06	03:14.03	03:14.03	03:21.04	03:17.09	03:16.07	03:16.03	03:15.07
10000	19.13	18.89	18.55	19.49	20.18	19.51	19.44	19.98	19.86	19.00
1000	1.81	1.52	1.95	2.00	1.82	2.04	2.00	1.82	2.01	2.00
100	0.17	0.1	0.09	0.09	0.18	0.19	0.09	0.14	0.17	0.1

Table 1 shows dependency of the execution time to the number of processed input words. The first column describes number of processed input words in distinct measure. Table displays results of measurements of execution time in seconds. The header consists of all samples which were used in tests. Mark c stands for clean C implementation and mark vm stands for custom virtual machine instruction set based implementation. Mark is then followed with used optimization flag used in compilation process.

Figures 1 to 5 shows dependency graphs of average execution time of samples in seconds for processing different number of input words.

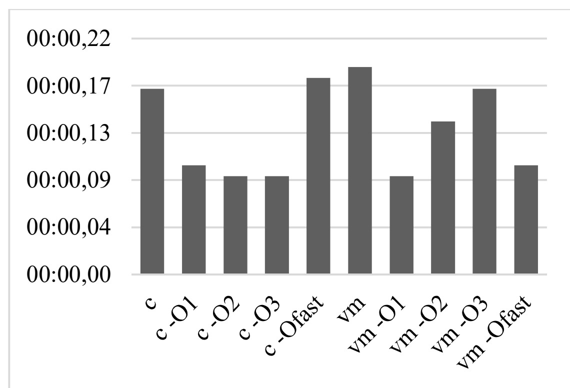

Fig. 1 Average execution time in seconds of samples for processing 100 input words
Source: authors.

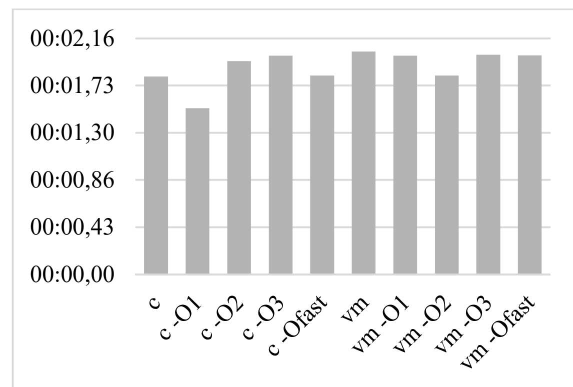

Fig. 2 Average execution time in seconds of samples for processing 1000 input words
Source: authors.

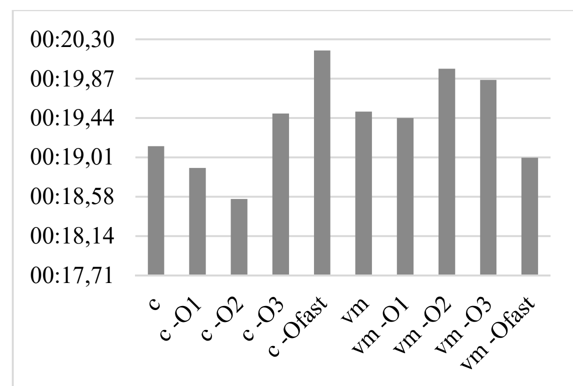

Fig. 3 Average execution time of samples for processing 10 000 input words
Source: authors.

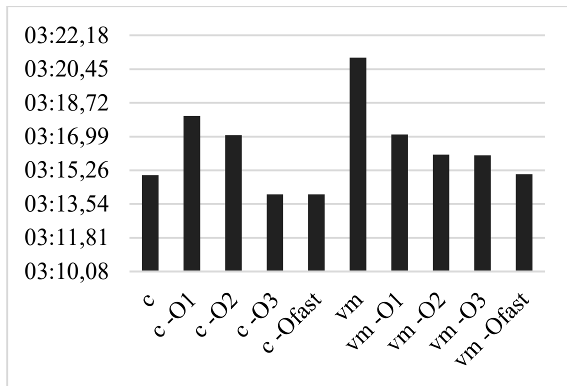

Fig. 4 Average execution time of samples for processing 100 000 input words
Source: authors.

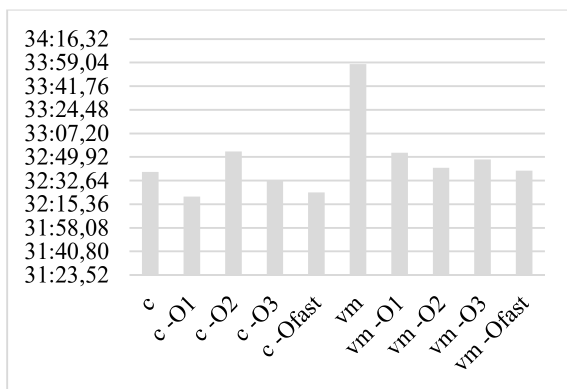

Fig. 5 Average execution time of samples for processing 1 000 000 input words
Source: authors.

It can be observed from the figures that, in general, there is only minor impact on performance between the pure C implementation and the custom emulator implementation for the selected algorithm and the input set. The maximal measured difference between samples without optimizations is only 13 % in favor for the pure C implementation. The maximal measured difference is 89 % in favor for pure C implementation. These results were measured between samples using -O3 optimizations flag on smallest input set. It is a lot, but it was measured on small input set, where operating system itself had huge influence on measures.

On the other hand, it is observable that using the smallest data input, sometimes the custom virtual machine instruction set based sample executes even quicker in some occasions.

Averagely, the difference was only 7 % in favor for the pure C implementation, but if we take into consideration only three biggest input datasets, difference was in average about 1 %.

We assume, that such small average difference of execution time between raw C implementation and custom RISC-based architecture emulator implementation is a result of effectively implemented shellcode for emulator with many mentioned compiler optimizations done by hand.

5 CONCLUSION

The main goal was to evaluate execution time of custom virtual machine instruction set emulator compared to pure C implementation of CRC16 algorithm. The research has been motivated by the utilization of the custom virtual machines in the area of obfuscation techniques for software protection and malware detection.

Measured differences are really small. The average difference is only 7 % in favor to pure C implementation.

Algorithm implementations were in both cases based on bitwise shifts and not on precalculated tables, which is less effective, but leave space for more calculations and further optimizations.

Future measures may be focused on following research questions:

Validation of measures, so the follow-up research may be focused on different algorithms implementations, in order to minimize complier impact on measures.

Measure the similarity score between different compilation flags usage, in order to minimize compiler impact.

To conclude, even though not all measurements results looks optimistic at initial view, but founded on assumptions that measured execution time in measures involving only small number of samples was highly influenced by operating itself, this custom virtual machine instruction set emulator overcame our assumptions, but it needs to be furthermore analyzed. Overall result of 7 % of execution time increase is inconsistent with (7) where was indicated that virtualization-based software will cause timing anomalies that will be detected ”for free” by timing-based attestation. Since our proposed custom instruction set virtual machine meets the requirements mentioned in (5) for code efficiency, it does not have to be detected “for free” as stated above.

References

[1] *QEMU a Fast and Portable Dynamic Translator.* Bellard, Fabrice. s. l. : USENIX Association, 2005. FREENIX Track: 2005 USENIX Annual Technical Conference. pp. 41-46.

[2] *Run apps on the Android Emulator. Android Studio.* [Online] 12. 27, 2019. Available at: <https://developer.android.com/studio/run/emulator>.

[3] YOU, I., KANGBIN, Y.: Malware Obfuscation Techniques: A Brief Survey. In: *Proceedings - 2010 International Conference on Broadband,*

Wireless Computing Communication and Applications, BWCCA 2010.

[4] KOSTELANSKÝ, J., DEDERA, L.: Custom virtual machine implementation and its influence on executable static properties. In: *2019 Communication and Information Technologies* (KIT). [online] Liptovský Mikuláš : Akadémia ozbrojených síl generála M. R. Štefánika, 2019. ISBN 978-80-8040-575-5.

[5] FANG, H., WU, Y., WANG, S., HUANG, Y.: Multi-stage Binary Code Obfuscation Using Improved Virtual Machine. In: Lai X., Zhou J., Li H. (eds) *Information Security*. ISC 2011. Lecture Notes in Computer Science, vol. 7001. Springer, Berlin, Heidelberg, 2011.

[6] LU, S., LYSECKY, R. L., ROZENBLIT, J. W.: Subcomponent timing-based detection of malware in embedded systems. In: *Proceedings - 35th IEEE International Conference on Computer Design, ICCD 2017.* pp. 17-24. [8119185] Institute of Electrical and Electronics Engineers Inc.. https://doi.org/10.1109/ICCD.2017.12

[7] KOVAH, X., KALLENBERG, C., WEATHERS, Ch.: New Results for Timing-Based Attestation. In: *IEEE Symposium on Security and Privacy.* 2012. pp. 239-253.

[8] LU, S., SEO, M., LYSECKY, R.: Timing-based anomaly detection in embedded systems. In: *20th Asia and South Pacific Design Automation Conference, ASP-DAC 2015.* pp. 809-814. [7059110] Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/ASPDAC.2015.7059110

[9] PETERSON, W. W., BROWN, D. T.: Cyclic Codes for Error Detection. Brown. In: *Proceedings of the IRE,* Vol. 49, Issue 1, 1961. pp. 228-235.

[10] *A Commonsense Approach to the Theory of Error-Correcting Codes.* s. l. : The MIT Press; n edition, 1988. 0262010984.

[11] SARWATE, D. V.: *Computation of Cyclic Redundancy Checks via Table Look-Up.* New York : Association for Computing Machinery, 1988. Commun. ACM, Vol. 31, p. 6. 0001-0782.

[12] Press, William, H., et al. *Numerical Recipes in C.* New York : Cambridge University Press, 2002. ISBN 0-521-43108-5.

[13] GRIFFITH, A.: *GCC: The Complete Reference.* New York : McGraw-Hill, Inc., 2002. ISBN 978-0-07-222405-4.

[14] GCC online documentation. *GCC.* [Online] Free Software Foundation, Inc., 11 28, 2019. Available at: <https://gcc.gnu.org/onlinedocs/>.

[15] Passwords. *Skullsecurity.* [Online] 5 18, 2015. Available at: <http://downloads.skullsecurity.org/passwords/rockyou.txt.bz2>.

[16] Time(1) - Linux man page. [Online] Available at: <https://linux.die.net/man/1/time>.

Dipl. Eng. Jozef **KOSTELANSKÝ**
Armed Forces Academy of General M. R. Štefánik
Department of Informatics
Demänová 393
031 01 Liptovský Mikuláš
Slovak Republic
E-mail: jozef.kostelansky@gmail.com

Assoc. Prof. RNDr. Ľubomír **DEDERA**, PhD.
Armed Forces Academy of General M. R. Štefánik
Department of Informatics
Demänová 393
031 01 Liptovský Mikuláš
Slovak Republic
E-mail: lubomir.dedera@aos.sk

Jozef Kostelansky was born in Slovakia. He received his engineer degree in 2016 in Communication and Information systems from the Military Academy in Liptovský Mikuláš with his thesis focused on Android malware analysis. He is currently the PhD. student researching hybrid malware analysis techniques.

Ľubomír Dedera works as an Associate Professor at the Department of Informatics, Armed Forces Academy in Liptovský Mikuláš. He graduated (RNDr.) from the Faculty of Mathematics and Physics, Comenius University in Bratislava in 1990. He received a PhD. degree in Artificial Intelligence from the Military Academy in Liptovský Mikuláš in 1997. His research interests include computer languages, computer security and artificial intelligence.