Supercompilation for Equivalence Testing in Metamorphic Computer Viruses Detection

Alexei P. Lisitsa and Matt Webster*

Department of Computer Science, The University of Liverpool
{A.Lisitsa,M.P.Webster}@csc.liv.ac.uk

1 Introduction

In this paper we present a novel approach to detection of metamorphic computer viruses by using proving program equivalence based on program transformation technique known as supercompilation [7]. Proving program equivalence is an undecidable problem in the general case; however, in specific cases we may find decidable or semi-decidable procedures that can prove that a sub-class of programs are equivalent. This is of relevance for detecting metamorphic computer viruses, which use a variety of semantics-preserving, syntax-mutating methods for code obfuscation. The main purpose of this obfuscation is to avoid detection by signature scanning. An important factor here is that semantics is preserved; therefore, if we can prove using some procedure that two different programs are equivalent, then in principle we can detect metamorphic computer viruses using this procedure.

The supercompilation[1] is a semantic based program transformation technique [7] for functional programming languages proposed by V. Turchin in the early 1970s. A variant of symbolic execution is used for the transformation: the program is executed with a partially defined input and that leads to the *unfolding* a potentially infinite tree of all possible computations of the parameterized program. In the process the tree of configurations is analysed and *folded* into the finite graph of parameterized configurations and posible transitions between them. To make folding possible a *generalization* procedure can be used. Finally, the supercompiler analyses the graph and builds the definition of output program based on that. Thus, a supercompiler implements the mapping $\langle P, e\rangle \mapsto \langle P', e'\rangle$, where P, P' are programs and e, e' are their corresponding parameterized entry points. The result of supercompilation, in general, implements an *extension* of the (partial) function implemented by the original program, i.e. P' produces the same outputs on the inputs for which P terminates, but may terminate on some inputs for which P does not. The primary purpose of supercompilation is for specialization and optimization of the programs. In Lisitsa & Nemytykh [3] it has been shown that it can be used for verification as well.

* A version of this paper has been presented at the Workshop on the Theory of Computer Viruses, 2008, Nancy, 15.05.2008

[1] from *super*vised *compilation*

Here we notice that due to the fact that resulting program is produced from a behavioural graph of possible computations (without referring to the original syntax) supercompilation can be seen also as, *behavior-based normalization procedure*[2], potentially applicable for the equivalence testing.

Development of supercompilation have been done mainly in the context of functional programming language Refal of Turchin [8] and SCP4 of Nemytykh & Turchin [5,6] is the most advanced supercompiler for Refal.

There are many methods of detecting metamorphic computer viruses in the literature. Our approach bears some similarity to the work of Webster & Malcolm [10,9] on detection of metamorphic computer viruses using algebraic specification, in which a specification of Intel 64 was given using Maude. The two approaches are similar in that the specification of Webster & Malcolm and the interpreter here use a notion of stores in the definitions of the semantics of the Intel 64 language. The approaches differ, however, in that the algebraic specification of Webster & Malcolm is based on a formal syntax and semantics of Intel 64, and the values of various variables are queried using rewriting, whereas the semantics of Intel 64 is specified informally in our work, and the supercompiler is used to optimise the evaluation function parameterised by a specific program.

Our approach is also similar to the program rewriting/normalisation approach of Bruschi et al [1,2], as supercompilation essentially rewrites a function corresponding to the execution of a program. Although the supercompilation is not strictly a normalisation procedure, as we cannot guarantee that in all cases two equivalent programs will have the same normal form, the process resembles normalisation as two functions representing different equivalent programs may be rewritten to the same form.

2 Supercompilation for Detection

Supercompilation is a program transformation process that traces possible generalized histories of a program in an attempt to reduce redundancy. As we will show, we can use the supercompilation process to produce supercompiled versions of metamorphic code fragments that are identical. This is useful for the detection of metamorphic computer viruses, which can be acheived by proving equivalence of a metamorphic computer virus signature to some suspect code fragment. We understand equivalence for two programs as equality of partial fucntions (mapping inputs to outputs, or inital states to the final states) implemented by programs.

Our technique uses a supercompiler for Refal called Scp4 [5]. We define the semantics of Intel 64 instructions operationally using Refal. Essentially, the result is a general-purpose interpreter for the Intel 64 instructions[3] we have defined.

[2] At the moment we suggest this reading as semi-formal. Determining precise conditions under which supercompilation would be a normalization procedure is an interesting problem for future investigations.

[3] At the moment only a small subset of instructions is covered.

Our interpreter can be found in the Appendix. If we pass a program as a parameter to the interpreter, the result is an emulation of that Intel 64 program in Refal. We can therefore apply Scp4 to the emulation in order to eliminate redundancy in the program. If two syntactically-different programs are supercompiled to the same form, we can conclude that the programs must be equivalent (under additional assumption that both programs terminate on all inputs). If programs may not terminate on some inputs then equality of residual programs provides only partial evidence for equivalence on a subset of inputs.

Example 1. The following two programs have the overall effect of assigning the value `5` to the variable `eax`, 6 to the variable `ebx` and 1 (or "true") to the zero flag of the EFLAGS register:

$p_1 =$ `mov eax,5; move ebx,5; cmp eax,ebx; move ebx,6`

$p_2 =$ `mov ecx,4; move eax,1; mov ebx,0; label 2: cmp eax,ebx;`
`je 1; mov eax,5; label 1: move ebx,6; loop 2`

We can imagine p_1 as part of the zeroth generation of a metamorphic computer virus, and p_2 as some obfuscated form. Applying the supercompiler to the interpreter twice, once for each program, results in the same supercompiled Refal program:

```
$ENTRY Go {
 (e.101 )(e.102 ) (e.103 ) (e.104 )  =
 (eax 5 ) (ebx 6 ) (ecx  ) (Zflag 1);
}
```

In each case, the supercompiler has optimised the interpreter, parameterised with programs p_1 and p_2 to the same Refal program, which simply assigns the values 5, 6 and 1 to the variables `eax`, `ebx` and `Zflag` [4]. Essentially, we have translated p_1 and p_2 into Refal, and the supercompiler has then shown the translated forms to be equivalent. If one of these programs was our signature, and the other was the suspect code, then this technique could be used to detect a metamorphic computer virus. More examples can be found in [4].

3 Conclusion

In a practical setting, e.g., within an anti-virus software package, we assume that code fragments for equivalence analysis will be extracted and presented before

[4] For simplicity of presentation, as it is the only place where arithmetic involved at the moment, we treat the values of counter register *ecx* differently from other registers. In the interpreter the values of *ecx* are modelled by unary strings and decrement operation is defined accordingly. Under such a convention the residual program assigns the value 0 to *ecx* register (as expected).

supercompilation. The supercompiler will then run with the two fragments as input, and the output of the supercompiler will be analysed in order to determine whether the two fragments are equivalent. This analysis, in the ideal case, is trivial: for example, the supercompiler could simply return the value "true" iff the two fragments are found to be equivalent. In the case where one fragment is a signature of a metamorphic computer virus, and the other fragment is some suspect code, then the positive identification of equivalence will indicate infection of the suspect code by that virus. Of course, this procedure is prone to false negatives in the case where the supercompilation process has not identified equivalence.

Future work will include an expansion of the Intel 64 instruction subset used, and an application to the detection of real-life metamorphic computer viruses. In addition, we intend to establish the theoretical constraints on our approach.

References

1. Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.
2. Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.
3. Lisitsa, A.P., Nemytykh, A.P.: Verification as a Parameterized Testing (Experiments with the SCP4 Supercompiler). Programmirovanie. No.**1** (2007) (In Russian). English translation in J. Programming and Computer Software, Vol. **33**, No.1 (2007) 14–23
4. A. Lisitsa, M. Webster: Detecting Metamorphic Computer Viruses using Supercompilation. In Proceedings of Workshop on Theory of Computer Viruses, 2008 (to appear), 5p
5. A. P. Nemytykh. The Supercompiler Scp4: General Structure. (Extended abstract). *Proceeding of the PSI'03,*, LNCS, vol. 2890, pp: 162-170, 2003
6. A. P. Nemytykh and V. F. Turchin. The Supercompiler Scp4: sources, on-line demonstration. http://www.botik.ru/pub/local/scp/refal5/, 2000.
7. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8:292–325, 1986.
8. V. F. Turchin. Refal-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, Massachusetts, 1989.
(electronic version: http://www.botik.ru/pub/local/scp/refal5/ ,2000.).
9. Matt Webster and Grant Malcolm. Detection of metamorphic and virtualization-based malware using algebraic specification. In *Proceedings of the 17th Annual European Institute for Computer Antivirus Research (EICAR) Conference*. To appear.
10. Matt Webster and Grant Malcolm. Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, 2(3):149–161, December 2006. DOI: 10.1007/s11416-006-0023-z.

4 Appendix. An interpreter of a subset of Intel 64 instruction set in Refal

```
*$MST_FROM_ENTRY;
*$STRATEGY Applicative;
*$LENGTH 3;
*$MATCING ForReapeatedSpecialization;



* A STORE is a list of variable-value pairs, e.g.
* (eax 0) (ebx 1) (ecx 2)


* entry point for the interpreter executing program p_2 from Example 1

$ENTRY Go {(e.1) (e.2)(e.3)(e.4)  =
<Exec ((control)(mov ecx (const I I I))(mov eax (const 1))(mov ebx (const 0))(label 2)
       (cmp (reg eax)(reg ebx))(je 1)(mov eax (const 5))
       (label 1)(mov ebx (const 6))(loop 2))(eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4)>;
}

* execute statement list

Exec {

* Execute jmp
* jump forward
(e.1 (control)(jmp e.label) e.2  (label e.label)  e.3) e.store  =
        <Exec (e.1 (jmp e.label) e.2 (label e.label)(control) e.3) e.store>;

* jump backward
              (e.1 (label e.label) e.2 (control)(jmp e.label) e.3 ) e.store =
<Exec (e.1 (label e.label)(control) e.2 (jmp e.label) e.3) e.store>;

*Execute mov
        (e.1 (control)(mov e.2 e.3) e.4) e.store =
        <Exec (e.1 (mov e.2 e.3)(control) e.4)
        <mov (e.2 e.3) e.store>>;

*Execute cmp and set Zflag
        (e.1 (control)(cmp (e.2) (e.3)) e.4) e.store =
        <Exec (e.1 (cmp (e.2) (e.3))(control) e.4)<cmp ((e.2) (e.3)) e.store>>;

*Execute je

*If Zflag is 1, jump forward
        (e.1 (control)(je e.label) e.2 (label e.label) e.3) e.4 (Zflag 1) =
        <Exec (e.1 (je e.label) e.2 (label e.label)(control) e.3) e.4 (Zflag 1)>;
*If Zflag is 1, jump backward
        (e.1 (label e.label) e.2 (control)(je e.label) e.3) e.4 (Zflag 1) =
        <Exec (e.1 (label e.label)(control) e.2 (je e.label) e.3) e.4 (Zflag 1)>;

*If Zflag is 0 Skip
        (e.1 (control)(je e.label) e.2) e.3 (Zflag 0) =
        <Exec (e.1 (je e.label)(control) e.2) e.3 (Zflag 0)>;

*Skip the label
        (e.1 (control)(label e.label) e.2) e.store =
        <Exec (e.1 (label e.label)(control) e.2) e.store>;


* Execute "loop label1": decrement counter register ecx,
*                        check if counter register is 0, if
*                        yes go to the next instruction, if not
*                        go to label1.
```

```
* The integer value of the counter ecx is presented in the unary form II...III.
* Only positive values are correctly dealt with

* Exit the loop
   (e.1 (control)(loop e.label) e.2) e.3 (ecx I)(Zflag e.5) =
   <Exec (e.1 (loop e.label)(control) e.2) e.3 (ecx)(Zflag 1)>;

* Go to the label backward
   (e.1 (label e.label) e.2 (control)(loop e.label) e.3) e.4 (ecx I I e.ecx)(Zflag e.5) =
   <Exec (e.1 (label e.label)(control) e.2 (loop e.label) e.3) e.4 (ecx I e.ecx)(Zflag 0)>;

* Go to the label forward
   (e.1 (control)(loop e.label) e.2 (label e.label) e.3) e.4 (ecx I I e.ecx)(Zflag e.5) =
   <Exec (e.1 (loop e.label) e.2 (label e.label)(control) e.3) e.4 (ecx I e.ecx)(Zflag 0)>;


*End of the statements list, nothing to execute
        (e.1 (control))e.store = e.store;

}

* Effects of mov execution

mov {
(eax (const e.1))(eax e.2)(ebx e.3)(ecx e.4)(Zflag e.5) = (eax e.1)(ebx e.3)(ecx e.4)(Zflag e.5);
(eax (reg eax))(eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4) = (eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4);
(eax (reg ebx))(eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4) = (eax e.2)(ebx e.2)(ecx e.3)(Zflag e.4);
(ebx (reg eax))(eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4) = (eax e.1)(ebx e.1)(ecx e.3)(Zflag e.4);
(ebx (const e.1))(eax e.2)(ebx e.3)(ecx e.4)(Zflag e.5) = (eax e.2)(ebx e.1)(ecx e.4)(Zflag e.5);
(ecx (const e.1))(eax e.2)(ebx e.3)(ecx e.4)(Zflag e.5) = (eax e.2)(ebx e.3)(ecx e.1)(Zflag e.5);
}

* Effects of cmp execution

cmp {
((reg eax)(reg ebx))(eax e.1)(ebx e.1)(ecx e.2)(Zflag e.3) = (eax e.1)(ebx e.1)(ecx e.2)(Zflag 1);
((reg eax)(reg ebx))(eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4) = (eax e.1)(ebx e.2)(ecx e.3)(Zflag 0);
((reg eax)(const e.1))(eax e.1)(ebx e.2)(ecx e.3)(Zflag e.4) = (eax e.1)(ebx e.2)(ecx e.3)(Zflag 1);
((reg eax)(const e.1))(eax e.2)(ebx e.3)(ecx e.4)(Zflag e.5) = (eax e.2)(ebx e.3)(ecx e.4)(Zflag 0);
}
```