Detecting Metamorphic Computer Viruses using Supercompilation

Alexei Lisitsa and Matt Webster*

In this paper we present a novel approach to detection of metamorphic computer viruses by proving program equivalence using a program transformation technique known as supercompilation [6, 5]. 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.

Supercompilation[1] is a semantics-based program transformation technique [6, 5] for functional programming languages proposed by V. Turchin in the late 1960s. 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* of a potentially infinite tree of all possible computations of the parameterised program. The tree of configurations is analysed and *folded* into the finite graph of parameterised configurations and possible transitions between them. To make folding possible a *generalisation* procedure can be used. Finally, the supercompiler analyses the graph and builds the definition of the output program based its analysis. 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 parameterised 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

*Department of Computer Science, University of Liverpool, Liverpool, L69 3BX, UK.

[1] from *super*vised *compilation*

not. The primary purpose of supercompilation is for specialization and optimization of programs. Supercompilation has also been used for program verification [3]. Development of supercompilation has been done mainly in the context of the functional programming language Refal [5]. SCP4 [4] is currently the most advanced supercompiler for Refal.

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

There are many methods of detecting metamorphic computer viruses in the literature. Our approach bears some similarity to the work of Webster & Malcolm [7] 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], as supercompilation essentially rewrites a function corresponding to the execution of a program. Although 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.

1 Supercompilation for Detection

Supercompilation is a program transformation process that traces possible generalised histories of a program in an attempt to reduce redundancy. As we will show, we can use the supercompilation process to produce identical supercompiled versions of metamorphic code fragments that are behaviourally equivalent. This is useful for the detection of metamorphic computer viruses, which can be achieved by proving equivalence of a metamorphic computer

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

virus signature to some suspect code fragment. We understand equivalence of two programs as the equality of the partial functions (mapping inputs to outputs, or inital states to the final states) implemented by those programs.

Our technique uses a supercompiler for Refal called SCP4 [4]. We have defined the semantics of a small subset of Intel 64 instructions using Refal. Essentially, the result is a general-purpose interpreter for the Intel 64 instructions we have defined[3]. 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 the supercompiler 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. We assume that both programs terminate on all inputs. If programs do not terminate on some inputs then equality of residual programs provides only partial evidence for equivalence on a subset of inputs.

Example. *The following three programs have the overall effect of assigning the value* `0` *to the variable* `eax`*,* `1` *to the variable* `ebx` *and* `0` *(or "false") to the zero flag*[4] *of the* `EFLAGS` *register:*

p_1 =
```
mov eax, 0 ; mov ebx, 1 ; cmp eax, ebx
```

p_2 =
```
jmp 1 ; label 1: mov ebx, 1 ; mov eax, ebx ;
mov eax, ecx ; mov eax, 0 ; jmp 2 ; mov eax, ecx ;
jmp 1 ; label 2: cmp eax, ebx
```

p_3 =
```
mov eax, 1 ; mov ebx, 1 ; cmp eax, ebx ; je 1 ;
mov eax, 5 ; label 1: mov eax, 0 ; cmp eax, ebx ;
je 1 ; mov eax, 0
```

We can imagine p_1 as part of the zeroth generation of a metamorphic computer virus, with p_2 and p_3 as some obfuscated forms. Program p_2 inserts unconditional jumps, unreachable code and redundant code in order to obfuscate its behaviour. Program p_3 uses pseudo-conditional jumps and unreachable code. All of these types of code metamorphosis are well-known, and have been observed in metamorphic computer viruses [2, 7].

Applying the supercompiler to the interpreter three times, once for each program, results in the same supercompiled Refal program:

```
$ENTRY Go {
```

[3]Our basic Intel 64 interpreter in Refal can be found online at `http://www.csc.liv.ac.uk/~matt/pubs/refal/1/`.

[4]In this basic interpreter, we have modelled the zero flag only, as it was the only flag required to implement the small instruction set used.

```
(e.101 ) (e.102 ) (e.103 ) (e.104 )  = (eax 0 ) (ebx 1 )
(ecx e.103 ) (Zflag 0 ) ; }
```

In each case, the supercompiler has optimised the interpreter, parameterised with programs p_1, p_2 and p_3, to the same Refal program, which simply assigns the values `0`, `1` *and* `0` *to the variables* `eax`, `ebx` *and* `Zflag` *(zero flag) respectively.*

Essentially, we have translated p_1, p_2 and p_3 into Refal, and the supercompiler has then shown the translated forms to be equivalent. If one of these programs was our signature, and the others were the suspect code samples, then this technique could be used to detect a metamorphic computer virus.

We can also show what happens when we present the supercompiler with another program, p_4, that is not equivalent to p_1, p_2 or p_3:

p_4 = `mov eax, 1 ; mov ebx, 1 ; cmp eax, ebx ; je 1 ;`
`mov eax, 5 ; label 1: mov eax, 0 ; cmp eax, ebx ;`
`je 1 ; mov eax, 1`

In this case, the resulting supercompiled program is different to the one above:

```
$ENTRY Go {
 (e.101 ) (e.102 ) (e.103 ) (e.104 )  = (eax 1 ) (ebx 1 )
 (ecx e.103 ) (Zflag 0 ) ; }
```

This case would correspond to the scenario in which the suspect code is not infected with the metamorphic computer virus. The supercompiler, therefore, shows that the suspect code is non-equivalent and prevents a false positive identification.

2 Conclusion

We have shown how supercompilation can be used to prove equivalence of programs, which can then be used to detect metamorphic computer viruses. We based our method on a interpreter for a fragment of the Intel 64 assembly programming language written in Refal. Applying the supercompilation technique to syntactically different, but behaviourally equivalent code resulted in the same "normalised" form. This can be applied to metamorphic computer virus detection where some virus signature and some suspect code differ syntactically; supercompilation can be applied to prove equivalence and therefore match the signature to the suspect code.

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

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 when the two fragments are transformed into identical programs by the supercompiler.) 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 a real-life metamorphic computer virus. In addition, we intend to establish the theoretical constraints on our approach, i.e., when detection by supercompilation is guaranteed to work, and when it is not.

References

[1] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *IEEE Security & Privacy*, 5(2):46–54, 2007.

[2] Arun Lakhotia and Moinuddin Mohammed. Imposing order on program statements to assist anti-virus scanners. In *Proceedings of Eleventh Working Conference on Reverse Engineering*. IEEE Computer Society Press, 2004.

[3] Alexei Lisitsa and Andrei P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Journal of Programming and Computer Software*, 33(1):14–23, 2007. Translated from Russian: Programmirovanie. No.1 (2007).

[4] Andrei P. Nemytykh. The supercompiler Scp4: General structure. In *Perspectives of System Informatics (PSI 2003): 5th International Andrei Ershov Memorial Conference*, volume 2890. Springer, 2003.

[5] Morten Heine B. Sørensen and Robert Glück. Introduction to supercompilation. In *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*. Springer, 1999.

[6] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[7] 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.