Parallel analysis of polymorphic viral code using automated deduction system

Ruo Ando
National Institute of Information and Communication Technology,
4-2-1 Nukui-Kitamachi, Koganei,
Tokyo 184-8795 Japan

Abstract

As malicious code has become more sophisticated and pervasive, faster and more effective system for forensics and prevention is important. Particularly, quick analysis of polymorphic (partly encrypted) viral code is necessary. In this paper we propose a parallel analysis of polymorphic viral code using automated deduction system. In proposed system, decipher routine and its parameters are detected by parallelized automated theorem proving. We apply the weighting and look-ahead heuristics for parallel analysis. We run several detection programs with different computing strategies for analyzing target viral binary code. When the fastest detection process is finished with computing time T(0), remaining detection processes with T(1..n) can be terminated in T(0). In experiment, computing time for detection is reduced with average rate about 46%. In about a half of all cases, $T(0) * 3 \leq$ *T(max) where T(max) is computing time without our strategy. That is, our parallel system makes detection program faster without appending hardware computing resources. Our system is lightweight and effective for reverse engineering and computer forensics.*

1 Introduction

Malicious mobile code has become more sophisticated. Software encryption and obfuscation is applied for evading signature matching and detection. This kind of code is called polymorphic viral code, of which signature is metamorphically generated. Unfortunately or not, operating system and application has become a large size. As a result, it takes long time to detect polymorphic virus. For forensics and prevention, we need faster and more lightweight detection system. Table 1 and 2 is the example of polymorphic malicious code. Two tables illustrate obfuscating API "GetModuleHandleA" by Win32.Metaphor[*]. These are functionally equivalent, but assembly code of table 2 is complicated so that we cannot detect it. For example,

```
mov dword_4, 32336C65h
```

is obfuscated to 4 instructions.

```
mov edi,32336C65h
lea eax,edi
mov edi,eax
mov dword_4,edi
```

This technique is called register substitution. This is applied for another computer viruses such as Win32.Evol and Win32.Zmist.

To cope with this kind of complicated code, we need to achieve two goals. First, instructions and parameters need to be extracted to reveal the target routine (GetModuleHandleA). Second, extraction of structure and parameters need to be done in reasonable computer time. For these goals, we propose a parallel analysis of polymorphic viral code using automated deduction system. In proposed system, we decuce instructions and parameters from polymorphically obfuscated code by automated theorem proving. For faster prevention, we parallelize our theorem proving system. Overview of proposed system is presented in section 3. Detecting instructions is illustrated in section 4. Detecting parameters is illustrated in section 5. Then, we discuss how to parallelize proposed deduction system in section 6 and 7. We discuss the effectiveness of our parallel analysis by numerical output of theorem prover in section 7.

2 Related work

Theoretical aspect of detecting computer virus is discussed in[1][2]. In 2001, Symantec published the paper about W32.Metaphor[3]. The application of software verification for detecting malware is divided into two fields: emulation based approach[5] and semantic based approach[6]. In [7], model checking is applied for checking program vulnerability. Attack graphs and algebraic specification is used for analyzing malicious code in [9]. Detailed techniques about reordering instructions is discussed in [10].

1	mov dword_3, 6E72654Bh
2	mov dword_4, 32336C65h
3	mov dword_5, 0h
4	push offset dword_3
5	call ds:GetModuleHandleA

Table 1. Assembly code of GetModuleHandleA API.

3	mov dword_1,0h
3	mov cdx,dword_1
3	mov dword_2,edx
3	mov edp,dword_2
2	mov edi,32336C65h
2	lea eax,[edi]
1	mov esi,0A624540h
1	or esi,4670214Bh
2	lea edi,[eax]
2	mov dword_4,cid
3	mov edx,ebp
3	mov dword_5,edx
1	mov dwrod_3,esi
4	mov edx,offset dword_3
4	push edx
5	mov dword_6,offset GetModuleHandleA
5	push dword_6
5	pop dwprd_7
5	mov edx,dword_7
5	call dword ptr ds:0[edx]

Table 2. Obfuscated assembly code of GetModuleHandleA API

3 System overview

Figure 1 show the overview of proposed system. Proposed system extracts four routines (parameter setting, payload transfer, loop/branch and decipher) from binary code. At first stage, we find opcode (instruction) and operand (argument). Second, proposal system classifies transfer instruction and other instructions. On the other hand, we translate operand into registers. At third stage, information of transfer instruction, other instruction and registers is gathered to find four routines.

To implement our model, we use open source software OTTER (Organized Techniques for Theorem-proving and Effective Research). OTTER[17] is a forth-generation of Argonne National Laboratory deduction system to prove theorems stated in FoL with Knuth-Bendix completion, with some strategies for directing and restricting searches.

4 De-obfuscation: detecting decipher instructions

In this section we discuss a way to detect decipher routine using theorem prover OTTER. Theorem prover simplifies obfuscated code by resolution. If theorem prover succeed to deduce clauses of decipher instructions (right side on Figure 1), unit conflict is occurred to terminate reasoning process. Code translation and weighting for faster resolution are also discussed.

4.1 Clause resolution

First, clauses on left side are simplified to two clause on right side as follows:

```
fact: (mov dword_1 0h)
fact: (mov edx dword_1)
conclusion: mov edx 0h.
```

Deduction is done by the resolution of theorem prover. Second, theorem prover occurs unit conflict to terminate the reasoning process. Unit conflict is generated when program get unit clauses with opposite in sign.

Definition: Unit conflict
The unit conflict is a event where two clauses contains a single literal of which signs are opposite and can be unified. These two clauses is called contradictory unit clauses.

In other words, if we succeed to extract decipher routine, unit conflict is occurred. To complete this process, we need three kinds of clauses.

```
set of support:
/* clause set of viral code. */
fact: (mov ebp, dword_2)
fact: (mov edx, ebp)
passive list:
/* clauses we try to find. */
conclusion: -(mov edx dword_2)
usable list:
/* clauses for resolution.*/
-mov(A,B) | -mov(C,A) | mov(C,B).
```

When theorem prover generates the same clauses from ”set of support” with opposite sign as ”passive list”, unit conflict is occurred.

4.2 Code translation

We translate the assembly code into clause expression understandable for theorem prover.

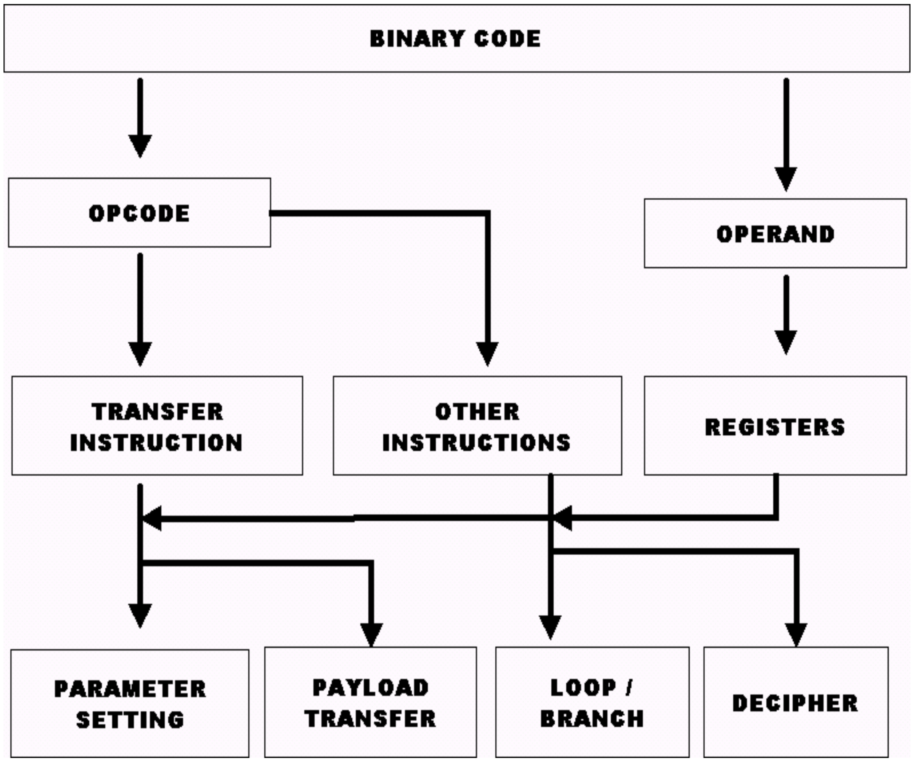

Figure 1. Overview of proposed system.

```
4337:010D  8A 03 mov al,[bp+di]
4337:010F  80 F4 2F xor ah,2Fh
4337:0112  02 07 add al,[bx]
```

The sample above is the output of disassembler. Left side is hex code of executable. Right side is disassembled code. We translate these as follows.

```
mov(reg(al),offset([bp+di]),
86,time(1)).
xor(reg(ah),const(2Fh),87,time(1)).
add(reg(al),offset([bx]),88,time(1)).
```

These clauses are placed on set of support. Theorem prover processes these clauses using transition axioms.

4.3 Weighting strategy

In our method, prover searches combination of clauses so that transition axioms could be applied. The number of possible combination becomes very large number. Then, some strategies are necessary in order to make reasoning process feasible, at least terminated in reasonable computing time. Weighting is technique to make our program focus on important clauses and block redundant paths of reasoning. For example, we set these clauses to make program focus on important instructions like these:

```
weight_list.
weight(xor($(*),$(*),$(*),$(*)),-X).
weight(jmp($(*),$(*),$(*),$(*)),-X).
weight(rotate($(*),$(*),$(*),$(*)),X).
end_of_list.
```

By setting these clauses, xor and jmp is paid more attention compared with rotate.

5 Detecting parameters of decipher routine

There are four parameters of decipher routine: address of encrypted data, key, address of loop entry point, and counter. The basic structure of obfuscated decipher routine is as follows:

```
set A address_of_payload
set B key
set C address_loop_start
set D counter

address_loop_start
   payload_transfer(A)
   decryptor(B)
   parload_transfer(A)
   branch(D)
   goto_start(C)
```

When we detect four parameters A-D, the detection is completed.

5.1 Paramodulation and demodulation

Paramodulation[15] is one of the techniques of equational reasoning. The purpose of this inference rule is for an equality substitution to occur from one clause to another. In the discussion of completeness and efficiency, paramodulation is often compared with demodulation[14]. Demodulation is mainly designed for canonicalizing information and sometimes more effective than paramodulation. However, demodulation does not have power to cope with clauses as follows:

```
fact: f(g(x),x).
fact: equal(g(a),b).
conclusion f(b,a).
```

That is, paramodulation is a generalization of the substitution rule for equality. For searching parameters of obfuscated decipher routine, we should use both paramodulation and demodulation.

```
fact: equal(data_16e,514Bh).
fact: mov(reg(ah),const(data_16e),
63,time(1)).
conclusion : mov(reg(ah),const(514Bh),
63,time(1)).
```

The clauses above is the application of demodulation to deal with constant number defined in the beginning of program. In obfuscating decipher routine, there's another way to hide parameter using mov (data transfer) instruction.

```
fact: mov(reg(ah),const(2Ch),
162,time(1)).
fact: mov(reg(bx),reg(ah),300,time(1)).
/* decrypter */
fact: xor(reg(dx),reg(bx),431,time(1)).
```

In this case, we insert this clause to occur paramodulation.

```
-mov(reg(x),const(y),z,time(1)) |
x=const(y,z).
conclusion:
decrypter(reg(dx),key(const(2Ch,162),
431,time(1)).
```

Conclusion is generated by paramodulation. By using paramodulation, we detect the value of [1]key, [2]address of payload, [3]loop counter (how many times the routine repeats), and [4]entry point of decipher routine.

5.2 Applying hot list strategy

In this paper we apply a heuristics to make paramodulation faster. Hot list strategy, proposed by Larry Wos[15], is one of the look ahead strategies. Look-ahead strategy is designed to enable reasoning program to draw conclusions quickly using a list whose elements are processed with each newly retained clause. Mainly, hot list strategy is used for controlling paramodulation. By using this strategy, we can emphasize particular clauses on hot list on paramodulation.

6 Parallelized reasoning process

In previous section, we discussed two reasoning heuristics, weighting and hot list strategy. Several weighting and hot lists is applied for our program. However, before detection is completed, which strategy is best to reduce the computing time. Computing time is quite different according to which strategy we apply. Then, parallel analysis is neccesary. Figure 3 illustrates proposed parallel analysis. In this figure, reasoning program with strategy 2 is fastest to be finished. Other processes are terminated when program 2 is finished. Let N(1), N(2) and N(3) be computing time of program 1, 2 and 3. Proposed parallel analysis is effective particularly when

```
N(2) * 3 < N(1) + N(2) + N(3)
```

As we discussed later, numerical output of our system achieves this comdition with probability rate of 50 %.

7 Numerical results

In this section we validate the effectiveness of our system by numerical output of theorem prover. First, we briefly discuss the result of weighting startegy.

7.1 SMEG

In experiment, we use SMEG (Simulated Metamorphic Encryption Generator)[15] to generate sample programs of obfuscated decipher routine. SMEG can generate hundreds of executables including obfuscated decipher routine. There are three types of SMEG mutations as shown in Table 3. Type A and C uses mov and xchg (exchange) to transfer the encrypted data. Type B uses indirect addressing (xor [address] key) to execute payload transfer and decryption at the same time. In type D, stack operation is applied for data transfer (fetch) and loop II (push / retf).

7.2 Weight lists

In this section we present the numerical output of theorem prover according to several weighting strategies. Table

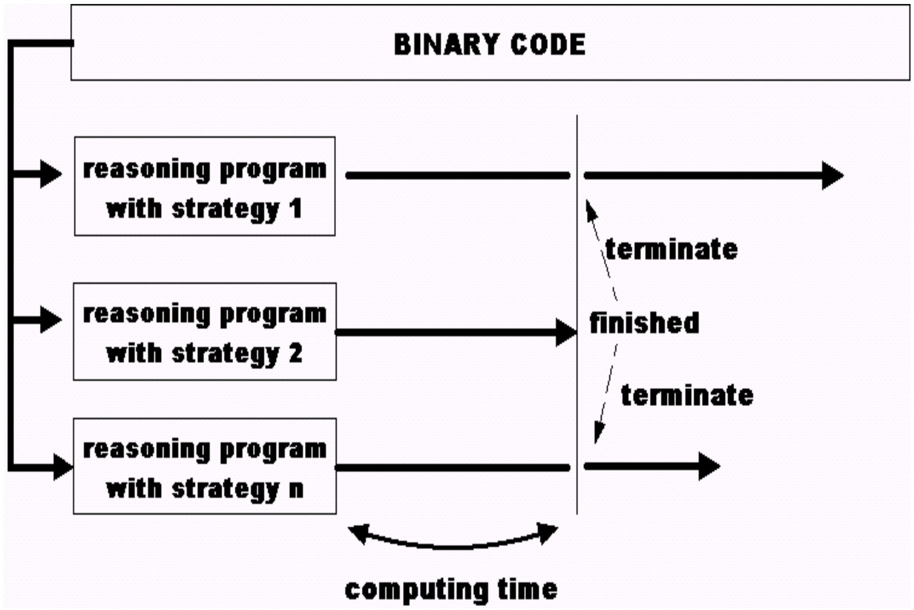

Figure 2. Proposed parallel analysis.

	Type A	Type B	Type C	Type D
loop I	set loop_start	loop_start	set loop_start	set loop_start
transfer I	mov data address	decrypt [address] key	xchg address data	push data / pop data
decrypt I	decrypt data key	decrypt [address] key	decrypt data key	decrypt data key
transfer II	mov address data	decrypt [address] key	xchg address data	mov address data
decrypt II	inc address	inc address	inc address	inc address
branch II	dec counter	dec counter	dec counter	dec counter
branch	test counter counter	test counter counter	test counter counter	test counter counter
loop II	jmp loop_start	jmp loop_start	jmp loop_start	push / retf

Table 3. Three kinds of assembly code generated by SMEG

4 and 5 show the number of generated clauses for detecting four types of SMEG mutation. + means that theorem prover focus on that instruction. - means not. It is shown that without proper weightings, computation time becomes too much longer according to Table 4 and 5. Among three types, weighting (-,+,+) results in good performance. Both tables indicates other instructions should not be focused. However, in Table 5, branch instructions should not be paid much attention. In both tables, Weighting (-,+,+) results in good performance.

7.3 Hot list strategy

To detect parameters, clauses are generated by reasoning program for paramodulation as follows.

```
-mov(reg(x),const(y),z,w) |
x=const(y,z).
```

Clauses on right side are called paramodulant. Pamodulant is used by theorem prover for equality substitution (paramodulation). We make two hot lists. In other words, we set hot list clauses about registers EAX, EBX, ECX EDX and ESI, EDI, EBP, ESP.

```
# hot list group I :
calculation registers
list(hot).
ax=const(x,y). bx=const(x,y).
cx=const(x,y). dx=const(x,y).
end_of_list.
```

```
# hot list group II :
memory registers
list(hot).
di=const(x,y). si=const(x,y).
bi=const(x,y). bp=const(x,y).
end_of_list.
```

generated code type A (mov for payload transfer)				generated code type C (xchg for payload transfer)			
MOV	DECRYPT	OTHER	generated clauses	MOV	DECRYPT	OTHER	generated clauses
+	+	-	1091841	+	+	-	1172590
+	-	-	1091912	+	+	-	1172590
-	+	-	1624209	-	+	-	1373584
-	-	+	707	-	-	+	666
-	+	+	778	-	+	+	604
+	-	+	707	+	-	+	666
generated code type B (using indirect addressing)				generated code type D (stack for paylaod transfer)			
MOV	DECRYPT	OTHER	generated clauses	MOV	DECRYPT	OTHER	generated clauses
+	+	-	640888	+	+	-	1172779
+	-	-	402426	+	-	-	1172842
-	+	-	745398	-	+	-	2426631
-	-	+	778	-	-	+	806
-	+	+	769	-	+	+	779
+	-	+	778	+	-	+	806

Table 4. Weighting strategies. + means that detection program focus on the instruction. - means not. Weighting (-,+,+) works well.

Type A (no weighting)		Type A (with weighting)	
HOT LIST	all clauses	HOT LIST	all clauses
no heat	915	no heat	707
EAX	677	EAX	677
EBX	670	EBX	602
ECX	799	ECX	541
EDX	756	EDX	540
EDI	1078	EDI	822
ESI	1055	ESI	801
EBI	1055	EBI	801
EBP	1055	EBP	801
Group I	468	Group I	366
Group II	1510	Group II	1206

Table 6. Hot list strategies for Type A. Paramodulation for detecting parameters into register E* is speeded up by hot list. We set 10 hot lists for each register and two groups.

Type B (no weighting)		Type B (with weighting)	
HOT LIST	all clauses	HOT LIST	all clauses
no heat	1592	no heat	769
EAX	915	EAX	605
EBX	1561	EBX	494
ECX	497	ECX	490
EDX	519	EDX	593
EDI	1921	EDI	1164
ESI	1724	ESI	843
EBI	1724	EBI	685
EBP	1724	EBP	685
Group I	463	Group I	242
Group II	2422	Group II	1807

Table 7. Hot list strategies for Type B.

Table 6, 7, 8 and 9 are result of applying hot lists for four types of SMEG generation. We make 10 hot list (list(hot)) accorging to eight registers and two groups {eax, ecx, ebx, edx} and {edi, esi, ebi, ebp}. Among 8 hot lists (eax, ecx, ebx,edx, edi, esi, ebi, ebp), which hot list increase performance best depends on types of generated code. As a whole, hot list group of calculation registers {eax, ecx, ebx, edx} results in good performance compared with group {edi, esi, ebi, ebp}. In some bad cases hot list of group II increase the number generated clauses compared with no hot list.

Let T(group I) be computing time with hot list group I. Let T(no weight) and T(group II) computing time with no heat and hot list group II. In experiment, our system achieves condition.

```
T(group I) < T(no weight) < T(group II)
or
T(group I) < T(group II) < T(no weight)
```

Particularly in type A and D, our system achieves this condition.

```
T(group I) *3 < T(no weight)
where T(group II) < T(no weight)
```

generated code type A (mov for payload transfer)				generated code type C (xchg for payload transfer)			
MOV	BRANCH	OTHER	generated clauses	MOV	BRANCH	OTHER	generated clauses
+	+	-	1836	+	+	-	1908
+	-	-	1091912	+	-	-	617600
-	+	-	2239	-	+	-	2538
-	-	+	1135508	-	-	+	630
-	+	+	780	-	+	+	604
+	-	+	1135395	+	-	+	586
generated code type B (using indirect addressing)				generated code type D (stack for payload transfer)			
MOV	BRANCH	OTHER	generated clauses	MOV	BRANCH	OTHER	generated clauses
+	+	-	2138	+	+	-	2481
+	-	-	402426	+	-	-	1172842
-	+	-	4620	-	+	-	2906
-	-	+	788	-	-	+	1673800
-	+	+	769	-	+	+	779
+	-	+	780	+	-	+	1673691

Table 5. Weighting strategies. + means that detection program focus on the instruction. - means not. Weighting (-,+,+) works well.

Type C (no weighting)		Type C (with weighting)	
HOT LIST	all clauses	HOT LIST	all clauses
no heat	976	no heat	604
EAX	1018	EAX	605
EBX	720	EBX	494
ECX	946	ECX	490
EDX	976	EDX	593
EDI	1592	EDI	1164
ESI	1272	ESI	843
EBI	1114	EBI	685
EBP	1114	EBP	685
Group I	738	Group I	463
Group II	2284	Group II	1807

Table 8. Hot list strategies for Type C.

Type D (no weighting)		Type D (with weighting)	
HOT LIST	all clauses	HOT LIST	all clauses
no heat	1877	no heat	801
EAX	1444	EAX	587
EBX	1675	EBX	587
ECX	870	ECX	599
EDX	1877	EDX	737
EDI	7406	EDI	1462
ESI	2028	ESI	876
EBI	2028	EBI	876
EBP	2028	EBP	876
Group I	563	Group I	259
Group II	8186	Group II	1891

Table 9. Hot list strategies for Type D.

```
or
T(group I) *3 < T(group II)
where T(no weight) < T(group II)
```

We can conclude that proposed parallel analysis model is effective. Particularly in type A and D, it is shown that we can make deduction system faster without appending hardware computing resources. c

8 Conclusion

As malicious code has become more sophisticated and pervasive, faster and more effective system for forensics and prevention is required. Software encryption and obfuscation is applied for geratate new malicious code of which signature is unknown (polymorphic). Quick analysis of polymorphic (partly encrypted) viral code is on demand. In this paper we propose the parallel analysis of polymorphic viral code using automated deduction system. In proposed system, decipher routine and its parameters are detected by parallelized automated theorem proving. Decipher instructions are detected by resolution. Parameters are detected by paramodulation and demodulation. We apply the weighting and look-ahead heuristics (hot list strategy) for parallel analysis. On parallel analysis system, several programs with different strategies for the target code. When the fastest detection process is finished with computing time T(0), remaining detection processes with T(1..n) can be terminated in T(0). In experiment, computing time for detection is reduced with average about 46%. In about a half of all cases, $T(0) * 3 \leq$ T(max) where T(max) is the longest computing time without our strategy. In these cases, our parallel system makes detection program faster without appending computing hardware resources.

References

[1] Computer viruses: from theory to applications. IRIS International series, Springer Verlag, ISBN 2-287-23939-1,2005.

[2] Diomidis Spinellis, "Reliable identification of bounded-length viruses is NP-complete", IEEE Transactions on Information Theory, 2000.

[3] Peter Szor and Peter Ferrie," Hunting for Metamorphic", Virus Bulletin Conference, 2001.

[4] Stephen Pearce, "Viral Polymorphism", paper submitted for GSEC version 1.4b, 2003.

[5] Michalis Polychronakis, Kostas G. Anagnostakis and Evangelos P. Markatos, "Network level polymorphic shellcode detection using emulation", Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2006.

[6] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, Randal E. Bryant, "Semantics-Aware Malware Detection", IEEE Security and Privacy, 2005.

[7] Mihai Christodorescu and Somesh Jha, "Static Analysis of Executables to Detect Malicious Patterns", USENIX Security Symposium, 2003.

[8] Hao Chen, Drew Dean, and David Wagner, "Model checking one million lines of C code", Annual Network and Distributed System Security Symposium (NDSS), 2004.

[9] O.Sheyner, J.Haines, S.Jha, R.Lippmann, and J. M. Wing, "Automated Generation and Analysis of Attack Graphs", IEEE Symposium on Security and Privacy , 2002.

[10] Arun Lakhotia, Moinuddin Mohammed, "Imposing Order on Program Statements to Assist Anti-Virus Scanners", Working Conference on Reverse Engineering, 2004.

[11] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, Koen De Bosschere, "Software Protection through Dynamic Code Mutation", Workshop on Information Security Applications, 2005.

[12] Larry Wos, George A. Robinson, Daniel F. Carson, Leon Shalla, "The Concept of Demodulation in Theorem Proving", Journal of Automated Reasoning, 1967

[13] W. McCune, "33 basic test problems: A practical evaluation of some paramodulation strategies", Preprint ANL/MCS-P618-1096, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1996

[14] Larry Wos, Gail W. Pieper, "The Hot List Strategy", Journal of Automated Reasoning, 1999

[15] Simulated Metamorphic Encryption Generator available at
http//vx.netlux.org/vx.php?id=es06

[16] IDA Pro disassembler avaibale at
http//www.datarescue.com/

[17] OTTER automated deduction system available at
http//www.mcs.anl.gov/AR/otter/

[18] Intel Corporation: IA-32 IntelR Architecture Software Developer's Manual, Volume 2A: Insruction Set Reference A-M,2004.

[19] Intel Corporation: IA-32 IntelR Architecture Software Developer's Manual, Volume 2B: Insruction Set Reference N-Z,2004.