APPENDIX

Let us show how to use SMODIC. The commands to launch the tool are as follows:

- SMODIC <option1> < modelfile> < option2 > <formula>

Option1 specifies the input file of SMODIC:

- M: the input is a SM-PDS model.
- B: the input is a binary program

Option2 specifies the model checking strategy:

- L: use the LTL model checking algorithm
- C: use the CTL model checking algorithm
- R1: perform the Reachability Analysis using pre^*
- R2: perform the Reachability Analysis using $post^*$

The model file can be either a binary program or a SM-PDS (.smpds file). The output have three files: one for the Control Flow Graph, one for assembly codes, and one for the generated SM-PDS. A SM-PDS consists of four parts: a finite set of standard PDS transition rules, a finite set of self-modifying transition rules, an initial phase (the initial set of transition rules) and an initial configuration (initial control location equipped with the stack contents).

Fig. 4. The Output of SMODIC

```
0x4ad0161a:     pushl   $0x4ad1f1b2     ; from: 0x4ad0504d(MAY),
0x4ad03fe4(MAY)
0x4ad0161f:     movl    %fs:0, %eax
0x4ad01625:     pushl   %eax
0x4ad01626:     movl    0x10(%esp), %eax
0x4ad0162a:     movl    %ebp, 0x10(%esp)
0x4ad0162e:     leal    0x10(%esp), %ebp
0x4ad01632:     subl    %eax, %esp
0x4ad01634:     pushl   %ebx
0x4ad01635:     pushl   %esi
0x4ad01636:     pushl   %edi
0x4ad01637:     movl    -8(%ebp), %eax
0x4ad0163a:     movl    %esp, -24(%ebp)
0x4ad0163d:     pushl   %eax
0x4ad0163e:     movl    -4(%ebp), %eax
0x4ad01641:     movl    $0xffffffff, -4(%ebp)
0x4ad01648:     movl    %eax, -8(%ebp)
0x4ad0164b:     leal    -16(%ebp), %eax
0x4ad0164e:     movl    %eax, %fs:0
0x4ad01654:     ret     ; targets: 0x4ad03fe9(MAY), 0x4ad05052(MAY)

0x4ad0165a:     movl    0x4ad33b90, %eax        ; from: 0x4ad03fef(MAY)
0x4ad0165f:     testl   %eax, %eax
0x4ad01661:     je      0x4ad04d48      ; targets: 0x4ad04d48(MAY),
0x4ad01667(MAY)
0x4ad01667:     pushl   $0x0    ; from: 0x4ad01661(MAY), 0x4ad04d75(MAY)
0x4ad01669:     call    %eax    ; targets: unresolved
0x4ad0166b:     ret     $0x4    ; targets: 0x4ad03ff4(MAY)      ; from:
0x4ad11d36(MAY)
```

Fig. 5. A Segment of Disassembly Codes

In order to show this, we will use the following command to check whether the program cmd.exe can eventually call the API function `GetModuleA` or not. For this case, we execute the following command:

- SMODIC B malware/cmd.exe L <>getmodulea

Figure 6 is the snapshot of the command to start SMODIC. In this command, "B" is Option1 specifying that the input is a binary program. "L" specifies that the strategy of model checking is LTL. <> getmodulea is the LTL formula **F** $(call\ GetModuleA)$.

```
dist — -bash — 80×24

Xin-MBP:dist Analy$ ./SMODIC B samples/cmd.exe L "<>getmodulea"
SMODIC is working
samples/cmd.exeJakstab STARTS working
Running Jakstab in/Users/Analy/NetBeansProjects/SMODIC/SMODIC
Analysis Manager Istance is generating and Jakstab is working

Analysis Manager is generating and Jakstab is working

ConfigurableProgram is loading and Jakstab is working

/Users/Analy/NetBeansProjects/SMODIC/SMODIC/org/jakstab/analysis

Reading machine specification from pentium.ssl.
Jakstab is working
Analysis Manager Istance is generating and Jakstab is working
```

Fig. 6. An Example to Run SMODIC

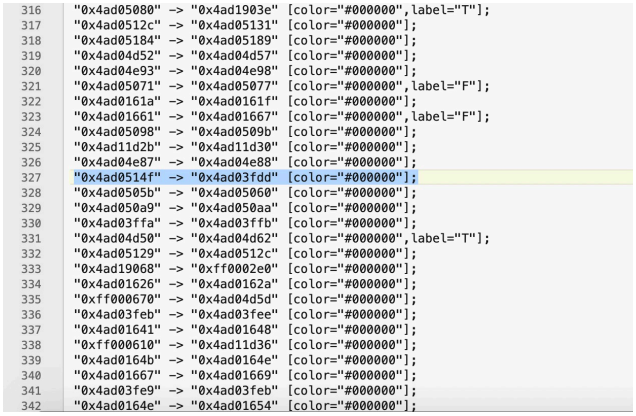
The output have three files: cfg.dot contains the control flow graph (Figures 7 and 8 are two segments of cfg.dot: the control locations corresponding to the instructions are given in Figure 7, and the edges between control locations are given in Figure 8), cmd.asm contains the assembly code equipped with informations about the API functions (Figure 5 is a fragment of this file), and model.smpds contains the SM-PDS (Figure 9 is a segment of the SM-PDS transition rules). This file contains in addition an initial configuration (the initial control point with the stack contents and the initial set of transition rules). The three files are shown in Fig. 4.

```
"0x4ad050cc"[label="0x4ad050cc\nmovl 0x4ad34874, %ecx"];
"0x4ad04e9a"[label="0x4ad04e9a\npushl $0x4ad04eb8"];
"0x4ad04e9a"[label="0x4ad04e9a\npushl $0x4ad04eb8"];
"0x4ad03fe4"[label="0x4ad03fe4\ncall 0x4ad0161a"];
"0x4ad01637"[label="0x4ad01637\nmovl -8(%ebp), %eax"];
"0x4ad0511e"[label="0x4ad0511e\npushl %eax"];
"0x4ad0511e"[label="0x4ad0511e\npushl %eax"];
"0x4ad04e89"[label="0x4ad04e89\nmovl 0x8(%ebp), %esi"];
"0x4ad050aa"[label="0x4ad050aa\norl $0xffffffff, 0x4ad2fa50"];
"0x4ad050aa"[label="0x4ad050aa\norl $0xffffffff, 0x4ad2fa50"];
"0x4ad03fe4"[label="0x4ad03fe4\ncall 0x4ad0161a"];
"0x4ad0514f"[label="0x4ad0514f\ncall 0x4ad03fdd"];
"0x4ad04d70"[label="0x4ad04d70\nmovl %eax, 0x4ad33b90"];
"0x4ad0510d"[label="0x4ad0510d\nleal -36(%ebp), %eax"];
"0x4ad0514f"[label="0x4ad0514f\ncall 0x4ad03fdd"];
"0x4ad01626"[label="0x4ad01626\nmovl 0x10(%esp), %eax"];
"0x4ad03ff4"[label="0x4ad03ff4\nmovl %ebx, -4(%ebp)"];
"0x4ad19049"[label="0x4ad19049\ncmpl $0xe, 0x84(%ecx)"];
"0x4ad0513e"[label="0x4ad0513e\nmovl __initenv@msvcrt.dll, %ecx"];
"0x4ad19049"[label="0x4ad19049\ncmpl $0xe, 0x84(%ecx)"];
"0x4ad01636"[label="0x4ad01636\npushl %edi"];
"0x4ad050db"[label="0x4ad050db\nmovl %eax, 0x4ad2fa54"];
"0x4ad01636"[label="0x4ad01636\npushl %edi"];
"0x4ad0506b"[label="0x4ad0506b\ncmpl $0x4550, (%ecx)"];
"0x4ad04e8c"[label="0x4ad04e8c\nandl $0x0, (%esi)"];
```

Fig. 7. Control Locations with Instructions

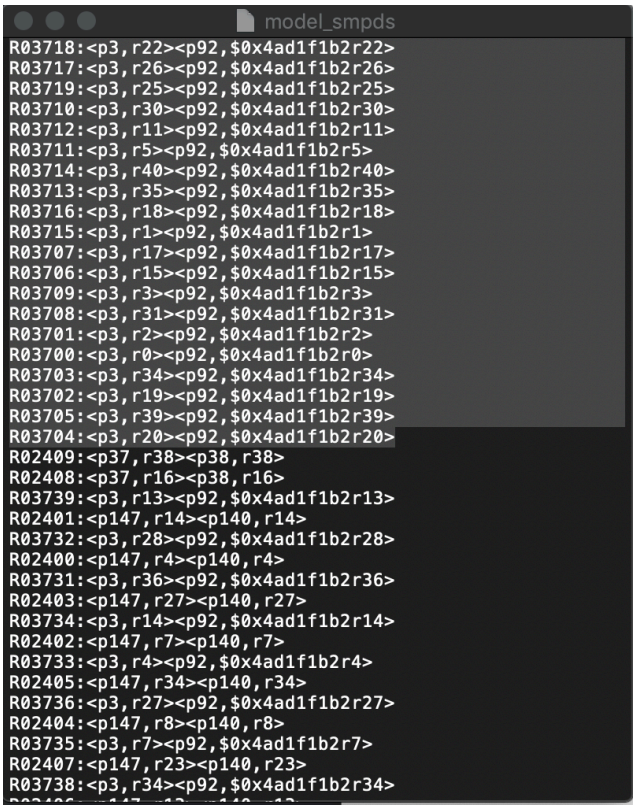
A. Reachability Analysis in SMODIC

Let us show how to use SMODIC to perform reachability analysis on SMPDSs and self-modifying code. To start the reachability analysis, we need to specify the options. Let us consider Option1 B, and Option2 R2(or R1). We also need to specify the sequence of API functions. For example, to perform the reachability analysis on the sequence of API functions "Call GetModuleA", "Call CopyFile" , "call SendFile", we put the names of the functions in lowercase and use the symbol ";" to separate the names. To use the $post^*$ approach to check whether the above sequence of API functions can be reached or not, we use the following command (see Fig. 10):



```
"0x4ad05080" -> "0x4ad1903e" [color="#000000",label="T"];
"0x4ad512c" -> "0x4ad05131" [color="#000000"];
"0x4ad05184" -> "0x4ad05189" [color="#000000"];
"0x4ad04d52" -> "0x4ad04d57" [color="#000000"];
"0x4ad04e93" -> "0x4ad04e98" [color="#000000"];
"0x4ad05071" -> "0x4ad05077" [color="#000000",label="F"];
"0x4ad0161a" -> "0x4ad0161f" [color="#000000"];
"0x4ad01661" -> "0x4ad01667" [color="#000000",label="F"];
"0x4ad05098" -> "0x4ad0509b" [color="#000000"];
"0x4ad11d2b" -> "0x4ad11d30" [color="#000000"];
"0x4ad04e87" -> "0x4ad04e88" [color="#000000"];
"0x4ad0514f" -> "0x4ad03fdd" [color="#000000"];
"0x4ad0505b" -> "0x4ad05060" [color="#000000"];
"0x4ad050a9" -> "0x4ad050aa" [color="#000000"];
"0x4ad03ffa" -> "0x4ad03ffb" [color="#000000"];
"0x4ad04d50" -> "0x4ad04d62" [color="#000000",label="T"];
"0x4ad05129" -> "0x4ad0512c" [color="#000000"];
"0x4ad19068" -> "0xff0002e0" [color="#000000"];
"0x4ad01626" -> "0x4ad0162a" [color="#000000"];
"0xff000670" -> "0x4ad04d5d" [color="#000000"];
"0x4ad03feb" -> "0x4ad03fee" [color="#000000"];
"0x4ad01641" -> "0x4ad01648" [color="#000000"];
"0xff000610" -> "0x4ad11d36" [color="#000000"];
"0x4ad0164b" -> "0x4ad0164e" [color="#000000"];
"0x4ad01667" -> "0x4ad01669" [color="#000000"];
"0x4ad03fe9" -> "0x4ad03feb" [color="#000000"];
"0x4ad0164e" -> "0x4ad01654" [color="#000000"];
```

Fig. 8. A Segment of Edges between Locations



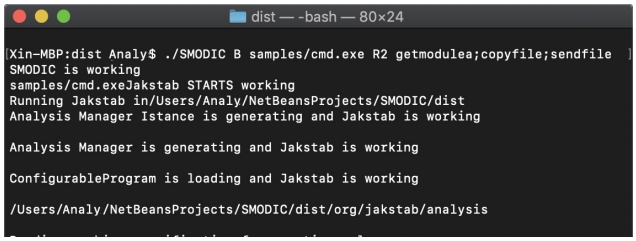
```
model_smpds
R03718:<p3,r22><p92,$0x4ad1f1b2r22>
R03717:<p3,r26><p92,$0x4ad1f1b2r26>
R03719:<p3,r25><p92,$0x4ad1f1b2r25>
R03710:<p3,r30><p92,$0x4ad1f1b2r30>
R03712:<p3,r11><p92,$0x4ad1f1b2r11>
R03711:<p3,r5><p92,$0x4ad1f1b2r5>
R03714:<p3,r40><p92,$0x4ad1f1b2r40>
R03713:<p3,r35><p92,$0x4ad1f1b2r35>
R03716:<p3,r18><p92,$0x4ad1f1b2r18>
R03715:<p3,r1><p92,$0x4ad1f1b2r1>
R03707:<p3,r17><p92,$0x4ad1f1b2r17>
R03706:<p3,r15><p92,$0x4ad1f1b2r15>
R03709:<p3,r3><p92,$0x4ad1f1b2r3>
R03708:<p3,r31><p92,$0x4ad1f1b2r31>
R03701:<p3,r2><p92,$0x4ad1f1b2r2>
R03700:<p3,r0><p92,$0x4ad1f1b2r0>
R03703:<p3,r34><p92,$0x4ad1f1b2r34>
R03702:<p3,r19><p92,$0x4ad1f1b2r19>
R03705:<p3,r39><p92,$0x4ad1f1b2r39>
R03704:<p3,r20><p92,$0x4ad1f1b2r20>
R02409:<p37,r38><p38,r38>
R02408:<p37,r16><p38,r16>
R03739:<p3,r13><p92,$0x4ad1f1b2r13>
R02401:<p147,r14><p140,r14>
R03732:<p3,r28><p92,$0x4ad1f1b2r28>
R02400:<p147,r4><p140,r4>
R03731:<p3,r36><p92,$0x4ad1f1b2r36>
R02403:<p147,r27><p140,r27>
R03734:<p3,r14><p92,$0x4ad1f1b2r14>
R02402:<p147,r7><p140,r7>
R03733:<p3,r4><p92,$0x4ad1f1b2r4>
R02405:<p147,r34><p140,r34>
R03736:<p3,r27><p92,$0x4ad1f1b2r27>
R02404:<p147,r8><p140,r8>
R03735:<p3,r7><p92,$0x4ad1f1b2r7>
R02407:<p147,r23><p140,r23>
R03738:<p3,r34><p92,$0x4ad1f1b2r34>
```

Fig. 9. A Segment of SM-PDS Transition Rules

- ./SMODIC B malware/cmd.exe R2 getmodulea;copyfile;sendfile

The result is shown in Fig. 11.

We also can run reachability analysis on SM-PDSs. Then, we need to specify the options. We make Option1 M, and Option2 R2(or R1). We also need to specify the target configuration. For example, we can execute reachability analysis



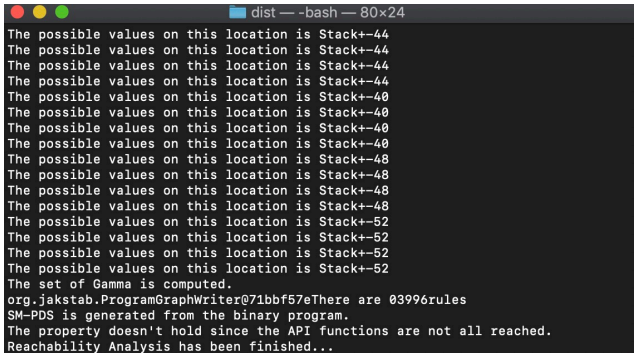
```
dist — -bash — 80×24
Xin-MBP:dist Analy$ ./SMODIC B samples/cmd.exe R2 getmodulea;copyfile;sendfile
SMODIC is working
samples/cmd.exeJakstab STARTS working
Running Jakstab in/Users/Analy/NetBeansProjects/SMODIC/dist
Analysis Manager Istance is generating and Jakstab is working

Analysis Manager is generating and Jakstab is working

ConfigurableProgram is loading and Jakstab is working

/Users/Analy/NetBeansProjects/SMODIC/dist/org/jakstab/analysis
```

Fig. 10. An Example to Start SMODIC for Reachability Analysis

using the $post^*$ approach to check if configuration $\langle p_0, r_0 \rangle$ can be reached or not by the following command:

- ./SMODIC M input.smpds R2 p0:r0

The output of SMODIC is the automaton representing the set of reachable configurations. SMODIC also tells whether the target configuration is reached or not.



```
dist — -bash — 80×24
The possible values on this location is Stack+-44
The possible values on this location is Stack+-44
The possible values on this location is Stack+-44
The possible values on this location is Stack+-44
The possible values on this location is Stack+-40
The possible values on this location is Stack+-40
The possible values on this location is Stack+-40
The possible values on this location is Stack+-40
The possible values on this location is Stack+-48
The possible values on this location is Stack+-48
The possible values on this location is Stack+-48
The possible values on this location is Stack+-48
The possible values on this location is Stack+-52
The possible values on this location is Stack+-52
The possible values on this location is Stack+-52
The possible values on this location is Stack+-52
The set of Gamma is computed.
org.jakstab.ProgramGraphWriter@71bbf57eThere are 03996rules
SM-PDS is generated from the binary program.
The property doesn't hold since the API functions are not all reached.
Reachability Analysis has been finished...
```

Fig. 11. The Result of the Example for Reachability Analysis

B. LTL and CTL in SMODIC

First, we will introduce the syntax of LTL/CTL used in SMODIC. To be able to use SMODIC, you need to be familiar with the syntax of the logics LTL and CTL. The implementation of LTL and CTL operators in SMODIC is as follows:

For LTL:

- Propositional Symbols: true, false and any lowercase string.
- Boolean operators: !(negation), -> (implication), <->(equivalence), && (and), || (or).
- Temporal operators: $[]p$(p always holds), $<>p$ (eventually p holds), pUq (p holds until q holds), and Xp (p holds next time).

For CTL:

- Propositional Symbols: tt(true),ff(false) and any lowercase string.
- Boolean operators: !(negation), -> (implication), <->(equivalence), && (and), || (or).
- Path quantifiers: A (for all paths) and E (there exists a path).

- Temporal operators: Xp (p holds next time), pRq (p holds until q does't hold), pUq (p holds until q holds).

We show how to apply LTL/CTL model checking to malware detection. Let us take a spy worm as example. Such a worm can record data and send it using the Socket API functions. For example, Keylogger is a spy worm that can record the keyboard states by calling the API functions GetAsyKeyState and GetKeyState and send this to the specific server by calling the socket function sendto. This behavior can be specified by the following LTL formula:

$$\phi_{sw} = \mathbf{F}\big((call\ GetAsyncKeyState \vee call\ GetRawInputData) \wedge \mathbf{F}(call\ sendto \vee call\ send)\big).$$

To check whether the program cmd.exe satisfies this formula or not, first, we need to rewrite this formula to the form supported by our tool SMODIC. Because all the propositions are lowercase strings, we rewrite API function calls (like call GetAsyncKeyState) by removing the word "$call$" and changing the name of the function in lowercase string. The operators are in spin syntax. Thus, formula ϕ_{sw} is rewritten as:

$$<> \big((getasynckeystate||getrawinputdata)\&\& <> (sendto||send)\big)$$

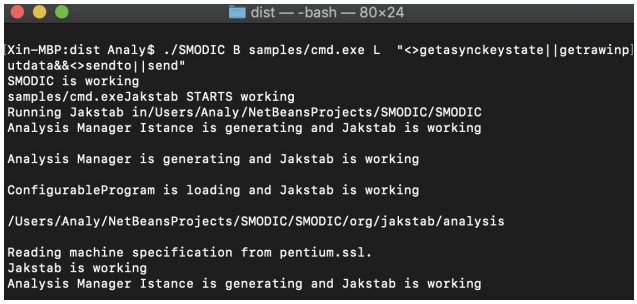


Fig. 12. Command for LTL Model Checking

So, we can check whether the program cmd.exe satisfies this formula or not by using the following command (shown in Figure 12):

./SMODIC B malware/cmd.exe L $<> \big((getasynckeystate|| getrawinputdata)\&\& <> (sendto||send)\big)$.

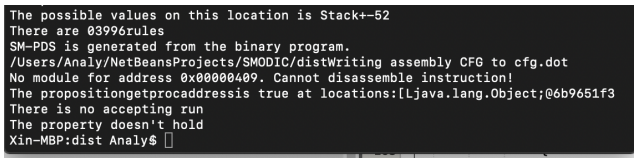


Fig. 13. Result of LTL Model Checking

The result is shown in Figure 13. The result of the computation is that there is no accepting run. The output of the tool is No, i.e. **cmd.exe** is not a spyware.

C. Specifying Malicious Behaviours in LTL

We first show several malicious behaviors in LTL. Then the results to check programs with SMODIC will be given.

Registry Key Injecting: In order to get started at boot time, many malwares add themselves into the registry key listing. This behavior is typically implemented by first calling the API function GetModuleFileNameA to retrieve the path of the malware's executable file. Then, the API function RegSetValueExA is called to add the file path into the registry key listing. This malicious behavior can be described in LTL as follows:

$$\phi_{rk} = \mathbf{F}\big(call\ GetModuleFileNameA \wedge \mathbf{F}(\ call\ RegSetValueExA)\big)$$

This formula expresses that if a call to the API function GetModuleFileNameA is followed by a call to the API function RegSetValueExA, then probably a malware is trying to add itself into the registry key listing.

Data-Stealing: Stealing data from the host is a popular malicious behavior that intend to steal any valuable information including passwords, software codes, bank information, etc. To do this, the malware needs to scan the disk to find the interesting file that he wants to steal. After finding the file, the malware needs to locate it. To this aim, the malware first calls the API function GetModuleHandleA to get a base address to search for a location of the file. Then the malware starts looking for the interesting file by calling the API function FindFirstFileA. Then the API functions CreateFileMappingA and MapViewOfFile are called to access the file. Finally, the specific file can be copied by calling the API function CopyFileA. Thus, this data-stealing malicious behavior can be described by the following LTL formula as follows:

$$\phi_{ds} = \mathbf{F}(call\ GetModuleHandleA\ \wedge \mathbf{F}(call\ FindFirstFileA \wedge \mathbf{F}\ (call\ CreateFileMappingA\ \wedge \mathbf{F}\ (call\ MapViewofFile \wedge \mathbf{F}\ call\ CopyFileA))))$$

Spy-Worm: A spy worm is a malware that can record data and send it using the Socket API functions. For example, Keylogger is a spy worm that can record the keyboard states by calling the API functions GetAsyKeyState and GetKeyState and send that to the specific server by calling the socket function sendto. Another spy worm can also spy on the I/O device rather than the keyboard. For this, it can use the API function GetRawInputData to obtain input from the specified device, and then send this input by calling the socket functions send or sendto. Thus, this malicious behavior can be described by the following LTL formula:

$$\phi_{sw} = \mathbf{F}\big((call\ GetAsyncKeyState \vee call\ GetRawInputData) \wedge \mathbf{F}(call\ sendto \vee call\ send)\big)$$

Appending virus: An appending virus is a virus that inserts a copy of its code at the end of the target file. To achieve

this, since the real OFFSET of the virus' variables depends on the size of the infected file, the virus has to first compute its real absolute address in the memory. To perform this, the virus has to call the sequence of instructions: l_1: call f; l_2: ....; f: pop eax;. The instruction call f will push the return address l_2 onto the stack. Then, the pop instruction in f will put the value of this address into the register eax. Thus, the virus can get its real absolute address from the register eax. This malicious behavior can be described by the following LTL formula:

$$\phi_{av} = \bigvee \mathbf{F}\Big(call \wedge \mathbf{X}(\text{top-of-stack} = a) \wedge \mathbf{G}\neg\big(ret \wedge(\text{top-of-stack} = a)\big)\Big)$$

where the $\bigvee$ is taken over all possible return addresses a, and top-of-stack=a is a predicate that indicates that the top of the stack is a. The subformula $call \wedge \mathbf{X}(\text{top-of-stack} = a)$ means that there exists a procedure call having a as return address. Indeed, when a procedure call is made, the program pushes its corresponding return address a to the stack. Thus, at the next step, a will be on the top of the stack. Therefore, the formula above expresses that there exists a procedure call having a as return address, such that there is no ret instruction which will return to a.

Note that this formula uses predicates that indicate that the top of the stack is a. Our techniques work for this case as well: it suffices to encode the top of the stack in the control points of the SM-PDS. Our implementation works for this case as well and can handle appending viruses.

D. Specifying Malicious Behavior in CTL

We give in what follows several examples of such malicious behaviors.

Kernel32.dll Base Address Malware. DLLs(Dynamic Link Libraries) will be loaded into the process when a program starts. The kernel32.dll module is always loaded into every process handling memory management and interrupts. Therefore, it contains several API functions that can be used by malicious codes. If a malware wants to copy itself to the installation directory of some programs, it will make a call to LoadLibrary that requires the location of LoadLibrary. Because Windows guarantees all DLLs get loaded at the same location, malicious codes only need to find the base address of Kernel32.dll to get the entrance (address) of the API functions. There are several kinds of approaches to implement to determine that base address. One is to find PE header of kernel32.dll in memory to locate Kernel32.dll export section. To aim this, the viruses seek first for the DOS header whose the beginning word is MZ (5A4Dh in hex) and then look for the PE header whose beginning word is PE00 (4550h in hex). In disassemble codes, instruction cmp is used to compare the value in register with 5A4Dh or 4550h. Thus, this behavior can be described in CTL as follows:

$$\phi_{bvi} = \bigvee \mathbf{EG}\big(\mathbf{EF}\ (cmp(r_1, 5A4Dh) \wedge \mathbf{EF}\ cmp(r_2, 4550h))\big)$$

where the $\bigvee$ is taken over all possible data registers r_1 and r_2 that includes eax, ebx, ecx and edx. This CTL formula expresses that the program has a loop that the values in registers r_1 compares to $5A4Dh$ while the value in r_2 compares to $4550h$. This malicious behavior can also be expressed in LTL formula $\psi_{bvi} = \bigvee \mathbf{GF}\ \big(cmp(r_1, 5A4Dh) \wedge \mathbf{F}\ cmp(r_2, 4550h)\big)$. But ψ_{bvi} is stronger than φ_{bvi} and φ_{bvi} is sufficient.

Email-Worms. Email worms are the worms that distributes copies of itself in an infectious executable targets that are attached to fake email messages. $Klez.h$ is such a malicious program. The $Klez$ family use API $GetModuleFileName$ to determine the filename of the executable it loaded from, and then uses another API function $CopyFile$ to copy this file to another location (such as a system directory). In order to retrieve its own file name, the parameter of call $GetModuleFileName$ will contain a zero handle (as a parameter) and the memory location. In disassemble codes, parameters of a call instruction will be pushed onto stack. Then the function $CopyFile$ will be called with the memory address as a parameter to copy this file. This behavior can be expressed by a CTL formula as follows:

$$\phi_{ew} = \bigvee \mathbf{EF}\big((\text{top-of-stack} = m) \wedge \mathbf{EX}\ \mathbf{E}\big[((\text{top-of-stack} = 0)\ \mathbf{U}\ (call\ GetModuleFileNameA \wedge (\text{top-of-stack} = 0))) \wedge \mathbf{EF}(call\ CopyFileA \wedge (\text{top-of-stack} = m))\big]\big)$$

where the $\bigvee$ is taken over all possible memory location m, and top-of-stack$= m$ is a predicate that indicates that the top of the stack is m. This formula means that the value of the top of the stack will be 0 just after the top of the stack is m. Then the top of the stack will be 0 until $GetModuleFileNameA$ is made with another parameter of $GetModuleFileNameA$ 0 on the top of the stack. Then, when function $CopyFileA$ is called, the value of top of the stack is still the value m. Therefore, $CopyFileA$ will copy the file whose name is at the address.

Spyware (Scanning the Disk). Spyware is a malware leading to an information leak by sealing it from the host. One typical behavior is to find a target file by scanning the disk of the host. Once the target found, it will run some procedure to steal it (by Internet), then continue to search the next file. To aim that, function $FindFirstFileW$ will be used to find the first object in a given path. If the result is a directory, it will call $FindFirstFileW$ again and the target file is found, it will call $FindNextFileW$ with the result of $FindFirstFileW$ (the return result will be stored in the register eax) to continue scanning the next target. Otherwise, it will call the function $GetLastError$. This behavior can be expressed as follows:

$$\phi_{spy} = \bigvee \mathbf{EF}\ \big(call\ FindFirstFileW \wedge \mathbf{EX}\ (\mathbf{AF}\ call\ GetLastError \vee call\ FindFirstFilew \vee (call\ FindNExtFileW \wedge (\text{top-of-stack} = a)))\big)$$

where the $\bigvee$ is taken over all possible value of address a obtained by the return value. This formula states that after function $FindFirstFileW$, then, in all the future paths of some immediate successor, the program either calls $GetLastError$ (if $FindFirstFileW$ failed) or calls $FindFirstFileW$ (if a directory is found) or calls $FindNextFileW$ with the top of the stack is the address a. Scanning a disk can be a behavior of a benign program. To avoid false alarm, it can be determined by stealing behaviors such as sending a file. Note that, the formula is branching time and cannot described as a LTL formula.

Appending Viruses. Virus needs to make itself executed. To aim that, one typical approach is to insert themselves codes at the end of target files i.e. appending viruses. Therefore, virus needs to get the location of this and would first compute an absolute address. This process will be performed by the sequence of instructions: l_1: call f; l_2: ....; f: pop eax;. The return address l_2 will be pushed onto the stack when instruction call f is executed. So, the return address can be obtained by instruction pop. Then, the pop instruction in f will put l_2 into the some register. Thus, the virus can get its absolute address from the register. This malicious behavior can be described by the following CTL formula:

$$\phi_{av} = \bigvee \mathbf{EF}\Big(\neg(call \wedge \mathbf{EX}(\text{top-of-stack} = a))\mathbf{U}\big(ret \wedge (\text{top-of-stack} = a)\big)\Big)$$

where the $\bigvee$ is taken over all possible return addresses a, and top-of-stack=a is a predicate that indicates that the top of the stack is a. The subformula $call \wedge \mathbf{X}(\text{top-of-stack} = a)$ means that there exists a procedure call having a as return address. This subformula indicates that in all the immediate successors of the call, the corresponding return address a is on the top of stack. Therefore, at the next step, a will be on the top of the stack. The rest subformula expresses that once the return instruction ret is executed, then the return address a should be on the top of the stack.