Trusted Computing Dynamic Attestation Using a Static Analysis based Behaviour Model

Tong Li[1],Fajiang Yu[1] [2], Yang Lin*[1], Xueyuan Kong[1], Yue Yu[1]
[1] School of Computer Science, Wuhan University,
Wuhan, Hubei, P.R.C. 430072
[2] Key Laboratory of Aerospace Information Security and Trusted Computing,
Ministry of Education in China
e-mail: linyang1117@xnmsn.cn/litong662008@163.com

***Abstract*—Current technology in trusted computing cannot comply with the requirement of trusted behaviour. One method for trusted computing dynamic attestation is proposed in this paper. This method uses a behaviour model based on the static analysis of binary code. One same source code may have several different binary versions, therefore one method is proposed for building almost the same core function model for different binary versions. This research also overcame the difficulty where some dynamic behaviours could not be obtained by static analysis. The paper also provides solutions for dynamic attestation of some complex programs, such as recursion, library link and multi threads programs.**

Keywords- trusted computing; dynamic attestation; behaviour model; static analysis

I. Introduction

Trusted computing is an information system security solution for basic computing security problems [1] [2]. The technology which trusted computing platforms currently adopts guarantees the integrity of its feature code. Its configuration data is the same as expected, before the components of the computing platform take control of the main CPU, which is called trusted computing static attestation, but which does not comply with the requirements that the behaviours are trusted [3]. We need to verify the dynamic behaviour of components as well, which is termed trusted computing dynamic attestation.

The related research mainly includes MCC (Model Carrying Code), PCC (Proof Carrying Code), semantic remote attestation, etc.

MCC [4] [5] was proposed by Sekar et al., its key idea is: The code producer generates behaviour information about the program security (model), a consumer receives both the model and the program from the producer. The consumer checks whether the model satisfies the consumer's security policy by formal reasoning. References [6] and [7] have carried out some implementations of MCC on a JVM (Java Virtual Machine). The MCC developer should know the program's source code, but this assumption is not always true, and many applications on trusted computing platform do not open their source code.

PCC [8] was proposed by Necula et al., its key idea is: The producer carries out analysis on the code and generates formal safety proofs, which are based on the consumer's policy. In addition, the proofs are bound to the source code, which usually is implemented by the compiler. The consumer uses type-based logic to automatically check the program, which is based on the same policy and refers to the safety proofs. The implementation of PCC also needs to know the program's source code.

Reference [9] proposed one semantic remote attestation (SRA) framework. SRA is based on a trusted Java virtual machine (Trusted VM) on the client side, and a server attests the Java program's hierarchies, restricted interfaces, runtime state, input information, etc. But there are no good solutions for building the semantic model from one program.

This paper mainly makes the following contributions: 1) The behaviour model is built based on the static analysis of binary code, which can cover all possible program execution paths. 2) Due to the availability of different compilers and different compiling options, one same source code may have several different binary versions, this paper proposed one method for building almost the same core function model for different binary versions. 3) By referring to the dynamic behaviour of an "empty program", this paper also overcame the difficulty that some dynamic behaviour cannot be obtained by static analysis. 4) The paper also provided the solutions for the dynamic attestation of some complex programs, such as recursion, link library using and multi-thread programs.

II. STATIC ANALYSIS-BASED PROGRAM BEHAVIOUR MODEL BUILDING

Program runtime behaviour attestation is the main feature of trusted computing dynamic attestation. The first step of program behaviour attestation is the building of the program behaviour model. Generally, there are two methods for program behaviour model building: dynamic training and static analysis. Dynamic training means it is hard to build a training set which can cover all possible program execution paths. This paper uses a behaviour model based on the static analysis of binary code, which can cover all possible program

This work is supported by the National Natural Science Foundation of China (Grant Nos. 60673071, 60970115, 91018008), the Fundamental Research Funds for the Central Universities in China (Grant No. 3101044), and the Open Foundation of the Zhejiang Key Laboratory of Information Security in China.

execution paths. It can be generated by the platform manufacturer or an administration centre.

A. Model Building Procedures

The process of constructing a static analysis-based program trusted behaviour model includes the following seven stages (see Fig. 1):

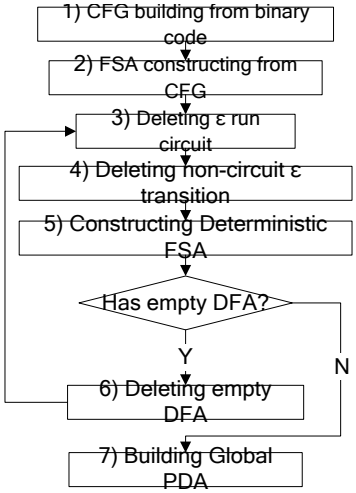


Figure 1. Flow of constructing a static analysis-based behaviour model

1) CFG building from binary code

We use one Interactive Dissembler (IDA) plug-in, named "wingraph32", to generate a Control Flow Graph (CFG) for every sub-function of the PE file. CFG is a directed graph, which can be represented by $G = \langle V, E \rangle$, where V is a finite set, element of V is vertex $v \in V$, which is a linear sequence of instructions, and E is a sub set of $E(V)$, $E \subseteq E(V)$, $E(V) = \{(u,v) | u,v \in V\}$.

2) FSA constructing from CFG

After being preprocessed, the original CFG is changed to $G' = \langle V', E' \rangle$. Every $v' \in V'$ is one of the following two vertex types: One where there are no instructions in it, the other is one in which there is only one call instruction. The exit of G' has no instruction. And thus we translate the CFG G' into a FSA (Finite State Automata) $M = (Q, \Sigma, \delta, S, F)$.

3) Deleting ε run circuit

If there is a state transition function $\delta(u', \varepsilon) = v'$ in M, we call it the "ε transition". Multiple ε transitions may form ε run circuits. An ε run circuit can lead to the failure of trust attestation. The FSA whose ε run circuits have been deleted is denoted as $M' = (Q', \Sigma', \delta', S', F')$.

4) Deleting non-circuit ε transition

After ε run circuits are deleted, there are still some non-circuit ε transitions in M', which may have an influence on the efficiency of trust attestation execution. The FSA whose all non-circuit ε transitions have been deleted is denoted as $M'' = (Q'', \Sigma'', \delta'', S'', F'')$.

5) Constructing Deterministic FSA

M" may be a NFA (Non-deterministic FSA). We can translate M" into an equivalent DFA to improve the efficiency of the trust attestation, which is denoted as $M_D = (Q_D, \Sigma_D, \delta_D, S_D, F_D)$.

6) Deleting empty DFA

If M_D has only one vertex $Q_D = \{q\}$, which is a start state and also a final state (the transition function set $\delta_D = \phi$) then M_D is an empty DFA. It is absolutely useless for trust attestation. When we remove one empty DFA, the corresponding sub function name should be deleted from the input alphabet of other non-empty DFAs, and the state transition whose input symbol is the function name should also be changed as an ε transition. The process of deleting empty DFA can lead to new ε run circuits appearing in other DFAs, we need to repeat the procedures in stages 3), 4), 5) and 6) until there are no empty DFAs in the model.

7) Building Global PDA

The object of trusted computing and dynamic attestation is one entire application program, we must construct one global DFA from all local sub-function DFAs. Since one sub function may be called in multiple positions, in order to ensure the sub-function can return correctly, the global DFA must be a PDA (Push down Automata), otherwise there may be some "impossible paths" [10] [11].

Information on the detailed stages and algorithms for building a trusted behaviour model based on the static analysis of a PE binary file can be seen in our previous work [12] [13].

B. Variance between Debug and Release Version

The same code is compiled with same compiler, but by choosing different compiling options different binary versions can be obtained, among which the most typical is the Debug and Release versions. Program behaviour model construction should not only support static analysis of the Release version, but also the Debug version.

We use Visual C++ 6.0 for compiling one instance to a Debug version, and then disassemble it using IDA. It can be observed that the program entry point is `_mainCRTStratup`, and the forms of API for calling in some sub-functions are of the form:

```
call ds:__imp__GetVersion@0;
call ds:__imp__OpenFile@12;
```

We use Visual C++ 6.0 to compile the same code to a Release version. It can be observed that the disassembled code's entry point is `start`, and the API calling of the sub-functions is very intuitionistic:

```
call ds: GetVersion;
call ds: GetCommandLineA;
call esi: OpenFile;
```

Thus, when we start constructing a program behaviour model for the Debug version, we need carry out some

preprocessing, such as removing prefixes like `__imp__` and suffixes like @*.

In addition, there is also a big difference in the number of sub-functions between the Debug and Release versions.

Both Debug and Release versions have a quite large number of sub-functions in the disassembled result of the corresponding binary program. The main cause is that the compiler will add some essential additional codes. The reason why the Debug version has more sub-functions than Release is that the Debug version contains a lot of debug information, and must include more API calling, such as `DebugBreak`, `InterlockedIncrement`, etc.

Although there is a big difference, the core function behaviour model based on the Debug version should be fairly consistent with the Release version.

C. Variance between Different Compilers

The same code is compiled with a different compiler, and different binary versions could be obtained. We use one empty Win32 console program as an instance (Example 1). This program is compiled with Visual C++ 6.0, Visual Studio 2005 and Visual Studio 2010. We then disassemble these versions, and the sub-function numbers in the disassembled results of the corresponding binary versions are shown in Table I.

Example 1: empty.c

```
void main(int argc, char* argv[])
{ return; }
```

Table I
SUB-FUNCTION NUMBER OF EXAMPLE 1'S MULTI-BINARY VERSIONS

	Numbers of sub-function	Numbers of sub-function in optimised model
Visual C++ 6.0 Release	55	30
Visual Studio 2005 Release	180	146
Visual Studio 2010 Release	145	109

Different compilers cover API in different wraps. VS2010 covers most API in wraps as sub-functions, all implementations are based on sub-function calling, such as `sub_401E83` only wrap one API `EncodePointer`, `sub_401E83` only wrap one API `TlsAlloc`. So the behaviour models of different binary versions have a different number of sub-functions.

To ensure the program runs safely some compilers place more emphases on initialisation, which also leads to the variance of sub-function number in different binary versions' behaviour models. For example, VS2010 invokes `HeapSetInformation` to set stack information, while VS2005 and VC6 do not. VS2005 invokes `__security_init_cookie` to initialise Cookies for preventing buffer overflow. VS2010 wraps the following API sequence into a sub-function `sub_40250F`, to accomplish the same initialisation of `__security_init_cookie`. However, VC6 does not carry out this work.

```
GetSystemTimeAsFileTime->GetCurrentProcessId-
>GetCurrentThreadId->GetTickCount-
>QueryPerformanceCounter
```

Besides which there are some different API callings in different program versions from different compilers. This is because some of the same functions are implemented by different APIs. Such as VC6 and VS2005 using `GetStartupInfoA` to obtain the information in the initialising stage, while VS2010 uses `GetStartupInfoW`. Some new compilers use extended API to replace the old ones, such as the `GetVersion` used by VC6 has already been replaced by `GetVersionEx`.

D. Modeling Management for Different Binary Versions

Although there is a large variance between different binary versions of the same code, their core function is the same.

We obtain the corresponding binary program (B_ϕ^{cdr}) with a specific compiler and compiling options to the compiler "empty program" (P_ϕ, see Example 1), its program behaviour model is denoted by M_ϕ^{cdr}, which can act as a reference template when we construct the model for other normal programs with the same compiler and compiling options.

At the time of behavioural modelling of a certain normal binary program (B_n), we can get the specific compiler name and determine that B_n is a Release version or Debug version by performing a static analysis of B_n, and then B_n can be specified as B_n^{cdr}. We follow the procedures in subsection II-A and obtain the optimised model of \$ B_n^{cdr}, denoted as M_n^{cdr}. Referring to the corresponding "empty program" behaviour model M_ϕ^{cdr}, we can remove the relevant parts about the initialisation and exit operations in M_n^{cdr}, the core function behaviour model $M_{nc}^{cdr} = M_n^{cdr} - M_\phi^{cdr}$. Then the different binary versions of the same P_n could have a fairly similar core function behaviour model.

We use Visual C++ 6.0 for compiling Example 1's code into a Release version. After the model construction and optimisation, there are 30 sub-functions in the optimised model, which can be seen as the reference template of a VC6 Release version program (M_ϕ^{6R}).

III. PROGRAM DYNAMIC BEHAVIOUR ATTESTATION

The framework of the program dynamic behaviour attestation using a static analysis based model is shown in Fig. 2. After building a model of the program's expected behaviour, we also need to monitor the program's running behaviour. We use the library of Microsoft Detours to monitor the program's behaviour, and monitor 311 core API functions in Ntdll.dll.

A. Preprocessing Program Behaviour

When one program runs on the operating system, certain Win32 APIs called by the program cannot be obtained by static analysis of the program. We need to conduct some program behaviour preprocessing, then we can use the static analysis-based program behaviour model to do the attestation.

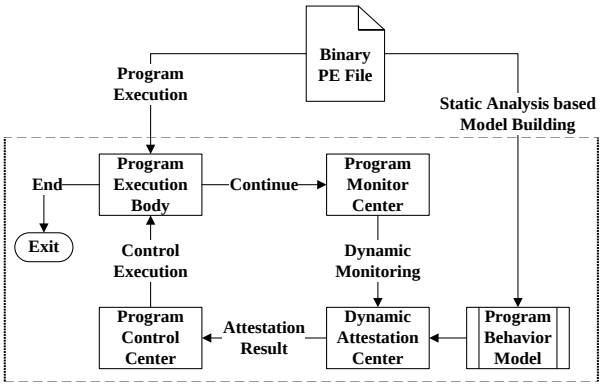


Figure 2. Framework of program dynamic behaviour attestation using a static analysis-based model

1) Preprocessing program initialisation and exit behavior

When running the console program which is compiled from Example 1 on Windows XP SP3, we can monitor the following API sequence:

GetFileType->LockResource->GetCommandLineA

The API `LockResource` cannot be obtained by static analysis of the corresponding binary program.

When we run the same program on Windows 7, the monitored API sequence is:

GetFileType->SetHandleCount->GetCommandLineA

Obviously, the same program runs on different operating systems, some Win32 APIs called by the program are also different.

We take the API sequence called by "empty program" as a standard, which is used to verify the program's actual API sequence during the process of initialisation and exit. And then the other API sequences left can be verified by using a static analysis-based program behaviour model.

2) Preprocessing complicated Win32 API behaviour

When one program calls some complicated Win32 APIs, the system will call other relevant APIs to complete the complicated function. These relevant APIs also cannot be obtained by static analysis of the program.

For example, when running the console program which is compiled from Example 1, we can monitor the following API sequence:

OpenFile->SearchPathA->SearchPathW->CreateFileA->CreateFileW->GetFileTime->FileTimeToDosDateTime

While carrying out a static analysis of the corresponding binary program, we can only obtain the API of `OpenFile`. So we firstly need to preprocess the relevant APIs, then we can use the static analysis-based program behaviour model to conduct attestation.

3) Preprocessing Unicode API behaviour

On the Windows NT-based operating system, the Win32 API calling related char operation (including ANSI char and Unicode char) will ultimately call Unicode API. For example, if the API is `GetModuleHandleA` in the static analysis based behaviour model of one program, then when running the program, we can monitor the two APIs: `GetModuleHandleA` and `GetModuleHandleW`. So we need to preprocess the program behaviour of Unicode API, then we can use the static analysis-based program behaviour model to conduct attestation.

B. General Program Dynamic Attestation

The preprocessed program behaviour is denoted as $w = a_1 a_2 \cdots a_n$, where $a_i (i = 1, 2, \cdots, n)$ is the name as the Win32 API. Now we can use the static analysis-based program behaviour model to verify w, just to see whether the constructed global PDA $M_G = (Q_G, \Sigma_G, \Gamma_G, \delta_G, q_{G0}, Z_{G0}, F_G)$ ($Z_{G0} = \varepsilon$) can accept w.

Whether M_G can accept w depends on whether M_G can be transformed from the initial Instantaneous Description $\mathrm{ID}(q_{G0}, w, \varepsilon)$ to $\mathrm{ID}(p_G, \varepsilon, \varepsilon)$ ($p_G \in F_G$) by making some moves, which is denoted as:

$$(q_{G0}, w, \varepsilon) \vdash_{M_G}^{*} (p_G, \varepsilon, \varepsilon), p_G \in F_G$$

$\vdash_{M_G}$ denotes that M_G makes a move, including ε move and non-ε move.

If $(p_G, \gamma) \in \delta(q_G, a_i, Z_G)$, then M_G can make a non-ε move, which is denoted as:

$$(q_G, a_i w', Z_G \beta) \vdash_{M_G} (p_G, w', \gamma\beta)$$

This idicates that when M_G is in state q_G and the symbol of the stack top is Z_G, M_G reads a_i, transforms its state to p_G, pops out Z_G, and pushes in γ.

If $(p_G, \gamma) \in \delta(q_G, \varepsilon, Z_G)$, then M_G can make an ε move, which is denoted as:

$$(q_G, w', Z_G \beta) \vdash_{M_G} (p_G, w', \gamma\beta)$$

This represents that when M_G is in state q_G and the symbol of the stack top is Z_G, M_G reads nothing, transforms its state to p_G, pops out Z_G, and pushes in γ.

C. Single Thread Program Attestation

On Window XP SP3, we use VC6 to compile the code in Example 2, and then build the behaviour model of the corresponding binary program. After being simplified by following the procedures described in subsection II-D, we get the core function model of Example 2's program, $M_s = (Q_s, \Sigma_s, \Gamma_s, \delta_s, q_{s0}, Z_{s0}, F_s)$, which is shown in Fig. 3.

Example 2: file.c

```
void main(int argc, char* argv[]) {
 ... ...
 pf1 = (HANDLE)OpenFile(fn1,&of,OF_READWRITE);
 if(pf1) {
  rt=ReadFile(pf1,bf1,sizeof(bf1),&rsize, NULL);
  if (rt) {
   pf2=(HANDLE)OpenFile(fn2,&of,OF_READWRITE);
   if(pf2) {
    rt=WriteFile(pf2,bf2,strlen(bf2),&wsize,NULL);
    CloseHandle(pf2);
   }}
  CloseHandle(pf1);
 }}
```

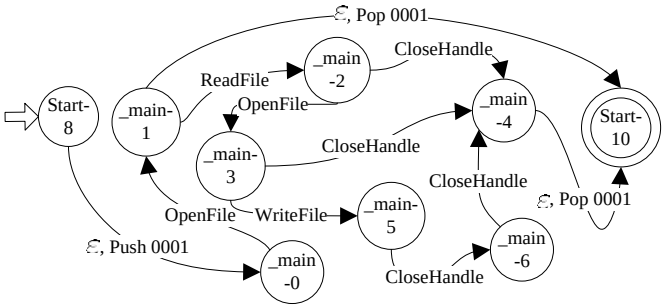


Figure 3. Core function behaviour model of Example 2

We run the console program which is compiled from Example 2 on Windows XP SP3, and preprocess the monitored Win32 API sequence by following the procedures described in subsection III-A. The preprocessed API sequence (w_s) is:

Openfile->ReadFile->OpenFile->WriteFile->CloseHandle->CloseHandle->CloseHandle

The initial instantaneous description of M_s is $\mathrm{ID}(\text{Start-}8, w_s, \varepsilon)$. Let us see whether M_s can accept w_s:

$(\text{Start-}8,\ \mathrm{w}_s, \varepsilon) \vdash_{M_s} (\text{\_main-}0, w_s, \text{`0001'})$
$(\text{\_main-}0,\ \text{`OpenFile'}\ \mathrm{w}_s^1, \text{`0001'}) \vdash_{M_s} (\text{\_main-}1, w_s^1, \text{`0001'})$
. . . . . .
$(\text{\_main-}4,\ \varepsilon, \text{`0001'}) \vdash_{M_s} (\text{Start-}10, \varepsilon, \varepsilon)$

`Start-10` is one final state, so w_s can be accepted by M_s. It means that the program behaviour during this run time passed the dynamic attestation.

D. Recursion Program Attestation

We use one instance to illustrate how to conduct dynamic attestation for a recursion program, whose source code is shown in Example 3.

Example 3: Recursion.c

```
int Recu(int i , HANDLE pfile , DWORD rsize);
void main(int argc, char* argv[]) {
 ... ...
 pfile=(HANDLE)OpenFile(FPATH, &of,
OF_READWRITE);
 if(pfile) {
  Recu(i, pfile, rsize);
  CloseHandle(pfile);
 }
}
int Recu(int i, HANDLE pfile, DWORD rsize) {
 if(i<=0) { ... }
 else {
  WriteFile(pfile,buf,strlen(buf),&rsize,NULL);
  Recursion (i-1, pfile, rsize);
  ReadFile(pfile, buf, sizeof(buf), &rsize, NULL);
 }
 return 0;
}
```

We build the behaviour model of the corresponding binary program by following the procedures described in section II, and obtain the core function model of the recursion program, $M_r = (Q_r, \Sigma_r, \Gamma_r, \delta_r, q_{r0}, Z_{r0}, F_r)$, which is shown as Fig. 4.

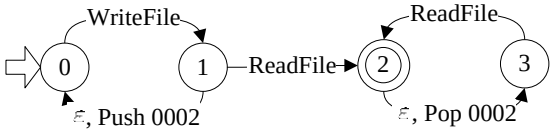


Figure 4. Core function model of recursion program

We run the recursion program which is compiled from Example 3, and preprocess the monitored Win32 API sequence by following the procedures described in subsection III-A. The preprocessed API sequence (w_r) is:

OpenFile->WriteFile->WriteFile->WriteFile->ReadFile->ReadFile->ReadFile

The first API `OpenFile` and the last API `CloseHandle` are called by `main()`. We only use the model of recursion function to verify the API sequence (w_r') between the first API `OpenFile` and the last API `CloseHandle`.

The initial instantaneous description of M_r is $\mathrm{ID}(0, w_r', \varepsilon)$. Let us see whether M_r can accept w_r':

$(0,\ \text{`WriteFile'}\ \ \mathrm{w}_r'^1, \varepsilon) \vdash_{M_r} (1, w_r'^1, \varepsilon)$
$(1,\ \mathrm{w}_r'^1, \varepsilon) \vdash_{M_r} (0, w_r'^1, \text{`0002'})$
. . . . . .
$(3,\ \text{`ReadFile'},\ \varepsilon) \vdash_{M_r} (2, \varepsilon, \varepsilon)$

2 is one final state, so w_r' can be accepted by M_r. This means that the program behaviour during this run time passed the dynamic attestation. This shows our method can solve the difficulty of dynamic attestation for a recursion program.

E. Library Link Program Attestation

We use one link library instance (source code is in Example 4) to illustrate how to conduct dynamic attestation for a library link program.

Example 4: Export function FOp in one link library

```
void FOp(char *PathS , char *PathD) {
 ... ...
 pfS=(HANDLE)OpenFile(PathS,&of,OF_READWRITE);
 if(pfS) {
  rt=ReadFile(pfS,tmp,sizeof(tmp),&rsize,NULL);
  if(rt) {

pfD=(HANDLE)OpenFile(PathDt,&of,OF_READWRITE);
   if(pfD) {
    WriteFile(pfD,tmp,sizeof(tmp),&rsize, NULL);
    CloseHandle(pfD);
   } }
  CloseHandle(pfS);
 }}
```

One program uses one link library in two ways: static link and dynamic link. The source code of one program is bound with the static link library (Example 4), as is shown in Example 5.

Example 5: Bounding with a static link library

```
#pragma comment(lib, "verDll.lib")
_declspec(dllimport)void FOp(char *ps,char *pd);
void main(int argc, char* argv[]) {
  ... ...
  FileOp(Src,Des);
  return;
}
```

We build the behaviour model of the corresponding binary program from Example 4 and Example 5 by following the procedures described in section II, and obtain the core function model, $M_{sl} = (Q_{sl}, \Sigma_{sl}, \Gamma_{sl}, \delta_{sl}, q_{sl0}, Z_{sl0}, F_{sl})$, which is shown in Fig. 5.

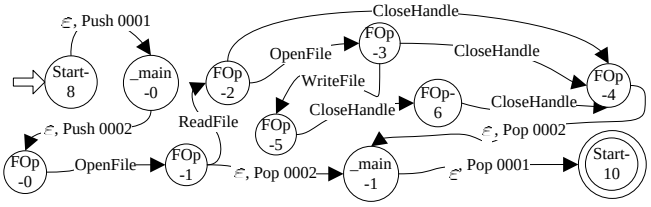


Figure 5. Behavior model of static link library program

We run the program which is compiled from Example 5, and preprocess the monitored Win32 API sequence by following the procedures described in subsection III-A. The preprocessed API sequence (w_{sl}) is:

```
OpenFile->ReadFile->OpenFile->WriteFile->CloseHandle->CloseHandle
```

The initial instantaneous description of M_{sl} is ID(Start-8, w_{sl}, ε). Let us see whether M_{sl} can accept w_{sl}:

$$(\text{Start-8}, w_{sl}, \varepsilon) \vdash_{M_{sl}} (\_\ \text{main-0}, w_{sl}, \text{`0001'})$$

$$(\_\ \text{main-0}, w_{sl}, \text{`0001'}) \vdash_{M_{sl}} (\text{FileOp-0}, w_{sl}, \text{`0002' `0001'})$$

$$\cdots\cdots$$

$$(\_\text{main-1}, \varepsilon, \text{`0001'}) \vdash_{M_{sl}} (\text{Start-10}, \varepsilon, \varepsilon)$$

`Start-10` is one final state, so w_{sl} can be accepted by M_{sl}. This shows our method can solve the difficulty of dynamic attestation for a static link library program.

The source code of one program bound with the dynamic link library (Example 4}) is shown in Example 6.

Example 6: Dynamic link library program

```
typedef void (FOp)(char *PS, char *PD);
void main(int argc, char* argv[]) {
  ... ...
  hDLLDrv=LoadLibrary("verDll.dll");
  if(hDLLDrv) {
    file=(FOp *)GetProcAddress(hDLLDrv,"FOp");
    if(file) file(Src, Des);
    FreeLibrary(hDLLDrv);
  }
  return;
}
```

Due to the act that the link library is dynamically loaded to the program space, the actual address of the export functions in the library cannot be obtained by static analysis. We build the behaviour model of the corresponding binary program from Example 6 by following the procedures described in section II, and obtain the core function model, $M_{dl} = (Q_{dl}, \Sigma_{dl}, \Gamma_{dl}, \delta_{dl}, q_{dl0}, Z_{dl0}, F_{dl})$, which is shown as Fig. 6.

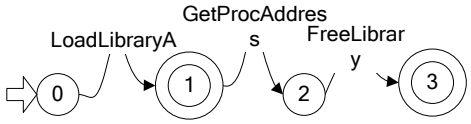


Figure 6. Behaviour model of dynamic link library program

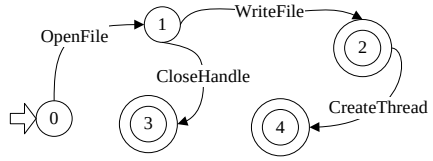
Based only on M_{dl}, it is impossible to conduct behaviour validation for a dynamic link library program. In our future work, we will obtain the name of the export function in the link library by carrying out an analysis of Win32 API arguments (such as `FOp`, the argument of `GetProcAddress` in Example 6). Then the behaviour model of `FOp` can be embedded into M_{dl}, and the attestation for the dynamic link library program can be completed.

F. Multi-thread Program Attestation

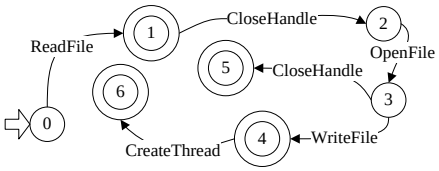
We build the behaviour model of one multi-thread binary program by following the procedures described in section II, and obtain the core function model, $M_m = (Q_m, \Sigma_m, \Gamma_m, \delta_m, q_{m0}, Z_{m0}, F_m)$, which is shown in Fig. 7.

The current behavioural model does not include API argument value, so there is no way to embed the automaton of each sub-thread into that of the main thread in order to form a complete global automaton.

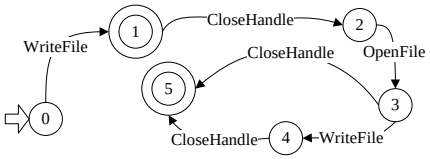
Due to the irregularity of parallel program execution in the operating system, the API calls for each thread to appear alternately, and the appearance order of the API is also different at each run-time. In addition to recording the API name, we should also record the thread ID who calls the corresponding API when monitoring the dynamic behaviour of a multi-thread program.



(a)_main



(b) Thread_1



(c) Thread_2

Figure 7. Behaviour model of multi-thread program

We independently conduct the dynamic attestation for every sub-thread's behaviour. The specific method is similar to single thread program attestation (see III-C). The difference from the single thread program is that we need to try to determine the corresponding relationship between the actual behaviour and sub-function's behavioural model of a certain thread by making multiple attempts.

In our future work, we will obtain the name of the sub-thread function by carrying out ab analysis of the arguments in `CreateThread`, and then the behaviour model of the sub-thread function can be embedded into the model of the main thread.

IV. ANTI-ATTACK EXPERIMENT

We use two typical attacking experiments to prove that the method of dynamic attestation proposed in this paper is effective.

A. DLL Hijacking

When one Windows program calls the API in one system DLL, the system will search the corresponding DLL in the system directory. This experiment uses one pseudo DLL in the system directory to launch the attack.

The object being attacked is a socket program (P_s). We build the behaviour model of the corresponding binary file of P_s by following the procedures described in section II, and obtain its function model, $M_s = (Q_s, \Sigma_s, \Gamma_s, \delta_s, q_{s0}, Z_{s0}, F_s)$. Valid API sequence in M_s is as follow:

> WSAStartup->socket->htons->bind->listen->accept->send->recv->closesocket

Since the socket program has to call the APIs from `ws2_32.dll`, we use one "malicious" DLL to replace the original `ws2_32.dll`. Except for `send()`, all other functions in the pseudo DLL are completely the same as the original `ws2_32.dll`. `send()` is modified using the following API sequence to steal sensitive information.

We run the program (P_s) on the platform with the malicious `ws2_32.dll`, and preprocess the monitored Win32 API sequence by following the procedures described in subsection III-A. The preprocessed API sequence (w_s) is:

> WSAStartup->socket->htons->bind->listen->accept->OpenFile->WriteFile->CloseHandle->send->recv->closesocket

Let us see whether M_s can accept w_s. When M_s reads in `OpenFile`, there is no path to complete the transition, denoted as:

$$(\mathrm{p}_s, \texttt{'OpenFile'}w'_s, Z_s\beta) \nvdash^*_{M_s} (q_s, w'_s, \gamma\beta)$$

At this time, $p_s \notin Q_s, \texttt{'OpenFile'}w'_s \neq \varepsilon, Z_s\beta \neq \varepsilon$. This means that M_s cannot accept w_s, and P_s's run behaviour cannot pass dynamic attestation. We can see our method can protect the system against a DLL Hijacking attack.

B. Buffer Overflow

The object being attacked is program (P_b) for file content copy. P_b reads some data from the first file, and writes the data into the second file. We build the behaviour model of P_b by following the procedures described in section II, and obtain its function model, $M_b = (Q_b, \Sigma_b, \Gamma_b, \delta_b, q_{b0}, Z_{b0}, F_b)$. Valid API sequence in M_b is as follow:

> Openfile->ReadFile->OpenFile->sprintf->strcpy->WriteFile->CloseHandle->CloseHandle

We build a piece of Shell Code, which will call `MessageBox`, and then write the Shell Code to the first file. When P_b uses `strcpy`, the Shell Code will be called. We run program (P_b) on the platform with the Shell Code file, and preprocess the monitored Win32 API sequence by following the procedures described in subsection III-A. The preprocessed API sequence (w_b) is:

> Openfile->ReadFile->OpenFile->sprintf->strcpy->MessageBoxA->WriteFile->CloseHandle->CloseHandle

Let us see whether M_b can accept w_b. When M_b reads in `MessageBoxA`, there is no path to complete the transition, denoted as:

$$(\mathrm{p}_b, \texttt{'MessageBoxA'}w'_b, Z_b\beta) \nvdash^*_{M_b} (q_b, w'_b, \gamma\beta)$$

At this time, $p_b \notin F_b, \texttt{'MessageBoxA'}w'_b \neq \varepsilon, Z_b\beta \neq \varepsilon$. This means that M_b cannot accept w_b, and P_b's run behaviour cannot pass dynamic attestation. We can thus see our method can protect the system against a buffer overflow attack.

Although the experiments carried by us are well known, our methods are also effective against other unknown attacks.

V. CONCLUSION AND FUTURE WORK

Our method for trusted computing dynamic attestation uses a behaviour model based on the static analysis of binary code, which can cover all possible program execution paths. One source code may have several different binary versions, this paper proposed one method of building almost the same core function model for different versions. This paper also overcame the difficulty that some dynamic behaviours cannot be obtained by static analysis, by referring to the dynamic behaviour of an "empty program". The paper also gave some solutions for the dynamic attestation of some complex programs, such as recursion, link library using and multi-thread programs.

Our current method cannot protect programs against mimicry attack [14] [15]. Some researchers have proposed methods to protect programs against mimicry attack. Based on this research, we will build program behaviour models using EFSA (Extended FSA) to describe the argument values. This needs to combine static analysis and dynamic training, because some specific argument values can only be obtained during run time. The behaviour model including arguments

also can help solve the difficulty of the dynamic link library program's behaviour attestation.

The method used in this paper cannot ensure the security of mobile code programs (such as Web script), which can only ensure the security of the script execution host program (such as Browser). We will carry out further research on the dynamic attestation of mobile code programs and parallel programs.

Acknowledgment

The authors wish to thank the reviewers for their useful comments.

References

[1] C. Shen, H. Zhang, H. Wang, J. Wang, B. Zhao, et al., "Research on trusted computing and its development," Science in China Series F: Information Sciences, vol. 40, no. 2, pp. 405–433, 2010.[online] available at: http://www.springerlink.com/content/a44nt6xg44801533/. accessed:2010.9 .

[2] C. Shen, H. Zhang, D. Feng, Z. Cao and J. Huang., "Survey of information security," Science in China Series F: Information Sciences, vol. 50, no. 3, pp. 273–298, 2007.

[3] Trusted Computing Group, "TCG specification: Architecture overview: Specification," revision 1.4, June 2010, [online] available at: http://www.trustedcomputinggroup.org/files/resource_files/AC652DE1-1D09-3519-ADA026A0C05CFAC2/TCG_1_4_Architecture_Overview.pdf, accessed:2010.9 .

[4] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney, "Model-carrying code: A practical approach for safe execution of untrusted applications," Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles ACM, Bolton Landing, New York, USA: ACM Press, 2003, pp. 15–28.

[5] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S.A. Smolka, "Model-carrying code (MCC): A new paradigm for mobile-code security," Proceedings of the 2001 Workshop on New Security Paradigms, Cloudcroft, New Mexico, USA: ACM Press, 2001, pp. 23–30.

[6] WEI Da, JIN Ying, ZHANG Jing, ZHENG Xiao-juan, LI Zhuo, et al., "Enforcing Security Policies in Open Source JVM," ACTA Electronica Sinica, vol. 37, no. 4A, pp. 36–41, 2009 (in Chinese).

[7] JIN Ying, LI Ze-Peng, ZHANG Jing, LIU Lei., "Static Checking of Security Related Behavior Model for Multithreaded Java Programs," Chinese Journal of Computers, vol. 32, no. 9, pp. 1856–1868, 2009 (in Chinese).

[8] G. C. Necula, "Proof-carrying code," Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, pp. 106–119, 1997.

[9] V. Haldar, D. Chandra, M. Franz, "Semantic remote attestation: A virtual machine directed approach to trusted computing," Proceedings of the 3rd Conference on USENIX Virtual Machine Research and Technology Symposium, San Jose, California, USA, pp. 29–41, 2004.

[10] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller .et al., Environment-sensitive intrusion detection. Lecture Notes in Computer Science 3858, Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, RAID 2005, Seattle, Washington, USA, pp. 185–206, 2005.

[11] Feng, H.H., Giffin, J.T., Yong Huang, Jha, S., Wenke Lee, et al., "Formalizing sensitivity in static analysis for intrusion detection," Proceedings of the 2004 IEEE Symposium on Security and Privacy, Oakland, California, USA, pp. 194–208, 2004.

[12] Y. Fajiang and Y. Yue, "Static analysis-based behavior model building for trusted computing dynamic verification," Wuhan University Journal of Natural Sciences, vol. 15, no. 3, pp. 195–200, 2010.

[13] Y. Yue, Y. Fajiang, and K. Yanan, "Optimizing Behavior Model Building for Trusted Computing Dynamic Verification," Wuhan University Journal of Natural Sciences, vol. 32, no. 20, pp. 121–125, 2010 (in Chinese).

[14] D. Wagner and P. Soto, "Mimicry attacks on host based intrusion detection systems," Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, DC, USA, pp. 255–264, 2002.

[15] W. Li, Y. Dai, Y. Lian, and P. Feng, "Context sensitive host-based IDS using hybrid automaton," Journal of Software, vol. 20, no. 1, pp. 138–151, 2009 (in Chinese).

.