Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications Workshops

Trusted Computing Dynamic Attestation by Using Static Analysis based Behavior Model

Fajiang Yu [*] [†], Xianglei Tang[‡], Yue Yu[*], Tong Li[*] and Tong Yang[*]
[*]*School of Computer, Wuhan University,*
Wuhan, Hubei, P.R.C. 430072
Email: qshxyu@126.com
[†]*Key Laboratory of Aerospace Information Security and Trusted Computing,*
Ministry of Education in China
[‡]*Zhejiang Provincial Testing Institute of Electronic Products,*
Hangzhou, Zhejiang, P.R.C. 310012

***Abstract*—Current technology of trusted computing can not comply with the requirement of trusted behavior. One method for trusted computing dynamic attestation was proposed in this paper. This method uses the behavior model based on static analysis of binary code. One same source code may have several different binary versions, this paper proposed one method of building almost the same core function model for different binary versions. This paper also overcame the difficulty that some dynamic behaviors can not be obtained by static analysis. The paper also gave out some solutions of dynamic attestation for some complex programs, such as recursion program, library link program and multi threads program.**

***Keywords*-trusted computing; dynamic attestation; behavior model; static analysis**

I. Introduction

Trusted computing is an information system security solution for the basic computing security problem [1]. The technology which trusted computing platforms currently adopted guarantees the integrity of its feature code and 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 it does not comply with the requirements that the behaviors are trusted. We need to verify the dynamic behavior of components as well, which is called trusted computing dynamic attestation.

The related researches mainly include MCC (Model Carrying Code), PCC (Proof Carrying Code), semantic remote attestation, and etc.

MCC [2] [3] was proposed by R. Sekar *et al.*, its key idea is: Code producer generates behavior information about 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. Document [4] and [5] have done some implementations of MCC on JVM (Java Virtual Machine). The MCC developer should know program's source code, but this assumption does not always be true.

PCC [6] was proposed by George C. Necula *et al.*, its key idea is: The producer carries analysis on code and generates formal safety proofs, which is based on the consumer's policy. The proofs are bounded in addition to source code, which usually is implemented by the compiler. The consumer uses type based logic to check the program automatically. The implementation of PCC also needs know program's source code.

Document [7] proposed one semantic remote attestation (SRA) framework . SRA is based on trusted Java virtual machine (Trusted VM) on client, and server attests Java program's hierarchies, restricted interfaces, runtime state, input information, and etc. But there is no good solutions of building the semantic model of one program.

This paper mainly makes the following contributions: 1) Behavior model is built based on static analysis of binary code. 2) This paper proposed one method of building almost the same core function model for different binary versions of same code. 3) This paper also overcame the difficulty that some dynamic behaviors can not be obtained by static analysis. 4) The paper also gave out the solutions of dynamic attestation for some complex programs.

II. Static Analysis based Program Behavior Model Building

Program runtime behavior attestation is the main part of trusted computing dynamic attestation. The first step of program behavior attestation is building program behavior model. You can see the detail stages and algorithms about building trusted behavior model based on static analysis of PE binary file in our previous work [8].

A. Variance between Debug and Release Version

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

978-0-7695-4429-8/11 $26.00 © 2011 IEEE
DOI 10.1109/ISPAW.2011.14

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

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

And we use Visual C++ 6.0 for the same code to a Release version. It can be observed that the disassembled code's entry point is `start`, and the API calling in some sub-functions is very direct:

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

So when we start constructing program behavior model for Debug version, we need do some preprocessing, such as removing prefixes like `__imp__` and suffixes like `@*`.

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

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

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

B. Variance between Different Compilers

The same code is compiled with different compiler, 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 respectively. And then we disassemble these versions, the sub function numbers in the disassembled results of corresponding binary versions are different.

Example 1: empty.c

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

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 behavior models of different binary versions have different number of sub functions.

To make sure program running safely, some compilers lay more emphases on initialization, which also lead to the variance of sub function number in different binary versions' behavior models. For example, VS2010 invokes `HeapSetInformation` to set stack information, while VS2005 and VC6 do not. VS2005 invokes `__security_init_cookie` to initialize Cookie for preventing buffer overflow. VS2010 wraps the following API sequence in a sub function `sub_40250F`, to accomplish the same initialization of `__security_init_cookie`. But VC6 does not carry this work.

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

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

C. Modeling Management for Different Binary Versions

Although there is big variance between different binary versions of one same code, they should have fairly same core function behavior model.

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

At the time of behavioral modeling of a certain normal binary program (B_n), we can get the specific compiler name and determine 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 get the optimized model of B_n^{cdr}, denoted as M_n^{cdr}. Referring to the corresponding "empty program" behavior model M_{ϕ}^{cdr}, we can remove the relevant parts about initialization and exit operation in M_n^{cdr}, the core function behavior model $M_{nc}^{cdr} = M_n^{cdr} - M_{\phi}^{cdr}$. Then the different binary versions of the same P_n could have fairly same core function behavior model.

III. Program Dynamic Behavior Attestation

After building the model of the program expected behavior, we also need to monitor the program's running behavior. We use the library of Microsoft Detours to monitor the program behavior, and monitor 311 core API functions in Ntdll.dll.

A. Preprocessing Program Behavior

When one program runs on the operating system, some Win32 APIs called by the program can not be obtained by static analysis of the program self. We need to do some program behaviors preprocessing, then we can use the static analysis based program behavior model to do attestation.

1) Preprocessing program initialization 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
```

And the API `LockResource` can not be obtained by static analysis of the corresponding binary program self.

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 initialization and exit. And then the other API sequences left can be verified by using static analysis based program behavior model.

2) Preprocessing complicated Win32 API behavior

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

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 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 behavior model to do attestation.

3) Preprocessing Unicode API behavior

On the Windows NT-based operating system, the Win32 API calling related char operation (including ANSI char and Unicode char) ultimately will call Unicode API. For example, if the API is `GetModuleHandleA` in the static analysis based behavior 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 behavior of Unicode API, then we can use the static analysis based program behavior model to do attestation.

B. General Program Dynamic Attestation

The preprocessed program behavior is denoted as $w = a_1a_2\cdots a_n$, where $a_i(i = 1, 2, \cdots, n)$ is the name of Win32 API. Now we can use the static analysis based program behavior model to verify w, just 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 ID(q_{G0}, w, ε) to 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 make a move, including ε move and non-ε move.

C. Single Thread Program Attestation

On Window XP SP3, we use VC6 to compile the code in Example 2, and then build the behavior model of the corresponding binary program. After being simplified by following the procedures described in subsection II-C, 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 as Figure 1.

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);
 }
}
```

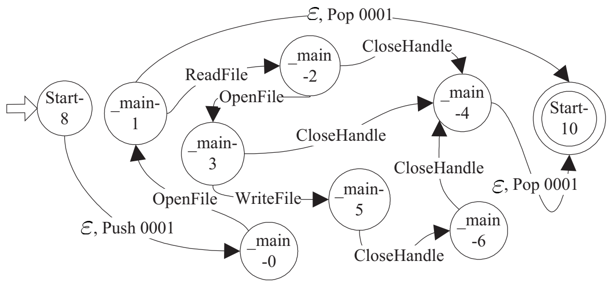

Figure 1. Core function behavior model of Example 2

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
```

The initial instantaneous description of M_s is ID(Start-8, w_s, ε). Let us see whether M_s can accept

w_s:

$$(\text{Start-8}, w_s, \varepsilon) \vdash_{M_s} (\text{\_main-0}, w_s, \text{‘0001’})$$
$$\ldots\ldots$$
$$(\text{\_main-4}, \varepsilon, \text{‘0001’}) \vdash_{M_s} (\text{Start-10}, \varepsilon, \varepsilon)$$

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

D. Recursion Program Attestation

We use one instance to illustrate how to do 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 behavior model of the corresponding binary program by following the procedures described in section II, and get 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 Figure 2.

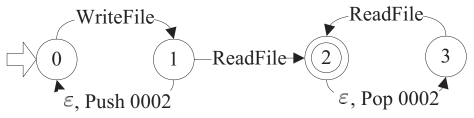

Figure 2. Core function model of recursion program

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
->CloseHandle
```

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 first API `OpenFile` and the last API `CloseHandle`.

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

$$(0, \text{‘WriteFile’ } w'^{1}_r, \varepsilon) \vdash_{M_r} (1, w'^{1}_r, \varepsilon)$$
$$\ldots\ldots$$
$$(3, \text{‘ReadFile’}, \varepsilon) \vdash_{M_r} (2, \varepsilon, \varepsilon)$$

`2` is one final state, so w'_r can be accepted by M_r. It means the program behavior during this run time passed the dynamic attestation. This shows our method can solve the difficulty of dynamic attestation for recursion program.

E. Library Link Program Attestation

We use one link library instance (source code is in Example 4 to illustrate how to do 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 bounded with the static link library (Example 4) 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 behavior model of the corresponding binary program from Example 4 and Example 5 by following the procedures described in section II, and get the core function model, $M_{sl} = (Q_{sl}, \Sigma_{sl}, \Gamma_{sl}, \delta_{sl}, q_{sl0}, Z_{sl0}, F_{sl})$, which is shown as Figure 3.

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
```

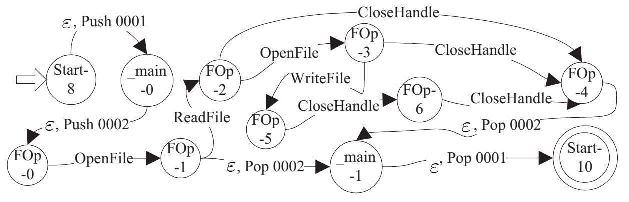

Figure 3. Behavior model of static link library program

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

$$(\text{Start-8}, w_{sl}, \varepsilon) \vdash_{M_{sl}} (\text{\_ main-0}, w_{sl}, \text{‘0001’})$$
$$\ldots\ldots$$
$$(\text{\_main-1}, \varepsilon, \text{‘0001’}) \vdash_{M_{sl}} (\text{Start-10}, \varepsilon, \varepsilon)$$

`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 static link library program.

When one program is bounded with the dynamic link library, due to that the link library is dynamically loaded to program space, the actual address of export functions in library can not be obtained by static analysis. In our future work, we will get the name of export function in link library by carrying analysis of Win32 API arguments, and complete the attestation for dynamic link library program.

F. Multi-thread Program Attestation

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

The current behavioral 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, and form a complete global automaton.

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

We do the dynamic attestation for every sub-thread's behavior independently. The specific method is similar to singe thread program attestation (see III-C). The difference from the single thread program is that we need try to determine the correspondence relation between the actual behavior and sub-function's behavioral model of a certain thread by making multi attempts.

In our future work, we will get the name of sub-thread function by carrying analysis of the arguments in `CreateThread`, and then the behavior model of sub-thread function can be embedded into the model of 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 system directory to launch the attack.

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

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

Because 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 in the same way like original `ws2_32.dll`. `send()` is modified as using the following API sequence to stole sensitive information.

```
OpenFile->WriteFile->CloseHandle->send
```

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:

$$(p_s, \texttt{'OpenFile'}\, w_s', Z_s\beta) \nvdash^*_{M_s} (q_s, w_s', \gamma\beta)$$

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

B. Buffer Overflow

The object being attacked is a 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 behavior model of P_b by following the procedures described in section II, and get its function model, $M_b = (Q_b, \Sigma_b, \Gamma_b, \delta_b, q_{b0}, Z_{b0}, F_b)$. There has the following valid API sequence in M_b:

```
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 into the first file. When P_b uses `strcpy`, the Shell Code will be called. We run the 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:

$$(p_b, \text{`}\mathtt{MessageBoxA}\text{'}\, w_b', Z_b\beta) \nvdash^{*}_{M_b} (q_b, w_b', \gamma\beta)$$

At this time, $p_b \notin F_b$, $\text{`}\mathtt{MessageBoxA}\text{'}\, w_b' \neq \varepsilon$, $Z_b\beta \neq \varepsilon$.This means M_b can not accept w_b, and P_b's running behavior can not pass the dynamic attestation. We can see our method can protect the system against buffer overflow attack.

Our methods are also effective against other unknown attacks.

V. Conclusion and Future Work

Our method for trusted computing dynamic attestation uses the behavior model based on static analysis of binary code. This paper proposed one method of building almost the same core function model for different versions of same code. This paper also overcame the difficulty that some dynamic behaviors can not be obtained by static analysis. The paper also gave out some solutions of dynamic attestation for some complex programs.

Our current method can not protect programs against mimicry attack [9] [10]. Some researchers have proposed some methods to protect program against mimicry attack. Based on these researches, we will build program behavior model by using EFSA (Extended FSA) to describe arguments value. The behavior model including arguments also can help solve the difficulty of dynamic link library programs' behavior attestation.

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

Acknowledgment

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

References

[1] Shen Changxiang, Zhang Huanguo, Wang Huaimin and *et al*. Research and development of trusted computing. *Science China: Information Science*, 2010, 40(2): 139-166. (in Chinese)

[2] R. Sekar, V. N. Venkatakrishnan, Samik Basu, *et al*. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03)*, Bolton Landing, New York, USA, ACM Press, 2003: 15-28.

[3] R. Sekar, C. R. Ramakrishnan, I.V. Ramakrishnan, and *et al*. Model-Carrying Code (MCC): A New Paradigm for Mobile-Code Security. *Proceedings of the 2001 New Security Paradigms Workshop (NSPW'01)*, Cloudcroft, New Mexico, USA, ACM Press, 2001: 23-30.

[4] Wei Da, Jin Ying, Zhang Jing,and *et al*. Enforcing Security Policie s in Open Source JVM. *ACTA ELECTRONICA SINICA*, 2009, 37 (4A): 36-41. (in Chinese)

[5] Jin Ying, Li Zepeng, Zhang Jing, and *et al*. Static Checking of Security Related Behavior Model for Multithreaded Java Programs. *Chinese Journal of Computers*, 2009, 32 (9): 1856-1868. (in Chinese)

[6] George C. Necula. Proof-Carrying Code. *Proceedings of 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, ACM Press, 1997: 106-119.

[7] Vivek Haldar, Deepak Chandra, Michael 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, USENIX Association, 2004: 29-41.

[8] Yu Fajiang, Yu Yue. Static analysis-based behavior model building for trusted computing dynamic verification. *Wuhan University Journal of Natural Sciences*, 2010, 15(3): 195-200.

[9] David Wagner, Paolo Soto. Mimicry attacks on host based intrusion detection systems. *Proceedings of the 9th ACM conference on Computer and Communications Security*, Washington, DC, USA, ACM Press, 2002: 255-264.

[10] Li Wen, Dai Yingxia, Lian Yifeng, and *et al*. Context sensitive host-based IDS using hybrid automaton. *Journal of Software*, 2009, 20(1): 138-151. (in Chinese)