SE Computational Systems
June 2004

Model-Carrying Code
A Practical Approach for Safe Execution of Untrusted Applications

R. Sekar, V.N. Venkatakrishnan, et. al.

Survey by
Gregor König
gkoenig@cosy.sbg.ac.at

University of Salzburg
Department of Computer Science.

Contents

Abstract

In this survey the model-carrying code approach for safe execution of untrusted applications is presented. The approach originates from R. Sekar et. al. [1] at Stony Brook University, New York.
The approach consists of three main activities. The first one is to extract a model containing a trace of system calls. This activity is performed by the code producer. Together with the code the model is sent to the user. The user can check the model against self-defined security policies and gets a comprehensive report abaout all security violations. She is empowered to decide either not to execute the code at all or to grant more access right to be able to execute the code.
As the model itself can also be faulty it has to be enforced during runtime. This causes an acceptable amount of overhead.

1 Introduction

Untrusted code has become a part of our daily life. Surfing the internet users are constantly confronted with possibilities to download code. This can happen intentionally, as it is the case with shareware or document viewers, or even unintentionally. Java Applets, which can be embedded in any website are downloaded automatically and executed.

The risk of executing faulty or malicious code is high. To be safe the user has to disable the features, which could download code automatically. But by doing this she wont be able to benefit from the code. In some cases it is not even possible to disable special features, because they are simply needed to work efficiently.

1.1 State of the Art

Currently there are several technologies available which try to deal with the problem of safe execution of untrusted code.

Code Signing Using signed code it is possible to give proof abaout the origin of code. The code producer signs the code and the consumer can verify the identity of the producer e.g. using a public/private key protocol. No proof is given about the programs behavior and its functionality. The code can still be faulty and cause unwanted effects on the user's computer. Code signing needs the cooperation of the producer and the consumer but does not solve the problem of safe execution.

Content inspection This idea deals with the static analysis of code on the user side. Usually code is retrieved in binary form. This makes it especially hard to analyze the programs behavior before actually executing it. A cooperation with the producer, who has access to the source code, would be desireable.

Execution monitoring Monitoring the runtime behavior of an application is the basic idea of this approach. Using policies the user is prompted if she is willing to grant access to some resources. Obviously the constant interaction with the user is not very convenient. In case the user chooses to abort the execution she will have to undo all modifications made by the application until that point of time. In addition all efforts made to acquire the program are in vain and the time spent to interact with the program is wasted.

Proof-Carrying Code In [2] the proof-carrying code technology is described. This approach enables the code producer to prove certain aspects of the program. This proof is sent to the user together with the code. The user can verify the proven aspects of the code but she cannot verify aspects of the application's behavior which are not proven by the producer.

None of these approaches satisfies the goals which can be reached using mode-carrying code. These goals are:

Producer and consumer Monitoring the execution enables the user to execute code safely, but it is completely her own task to secure the code. Using Java Applets the Java security policies can be used. Very fine grained policies can be declared but for amateur users this is an unsolvable task.
Using the proof-carrying code approach the producer has to guess which aspects of the program are security relevant for the code consumer. The burden of securing the code lies completely on the producer's side.

Empowering the code consumer Without losing convenience the user should be able to define policies herself. Using proof-carrying code she is completely passive concerning the proven security aspects.

Convenience for the user Execution monitoring can only be done during runtime. But it should be possible to compare the user's policies with the programs behavior prior to execution.

2 The MMC Approach

The key ideas of the approach can be seen in figure 1.

The model carrying code framework has a producer side and a consumer side. The code producer extracts a model, which covers all security relevant aspects of the code. An automated model extractor can be used to build the model. This model is sent to the code consumer together with the code itself.

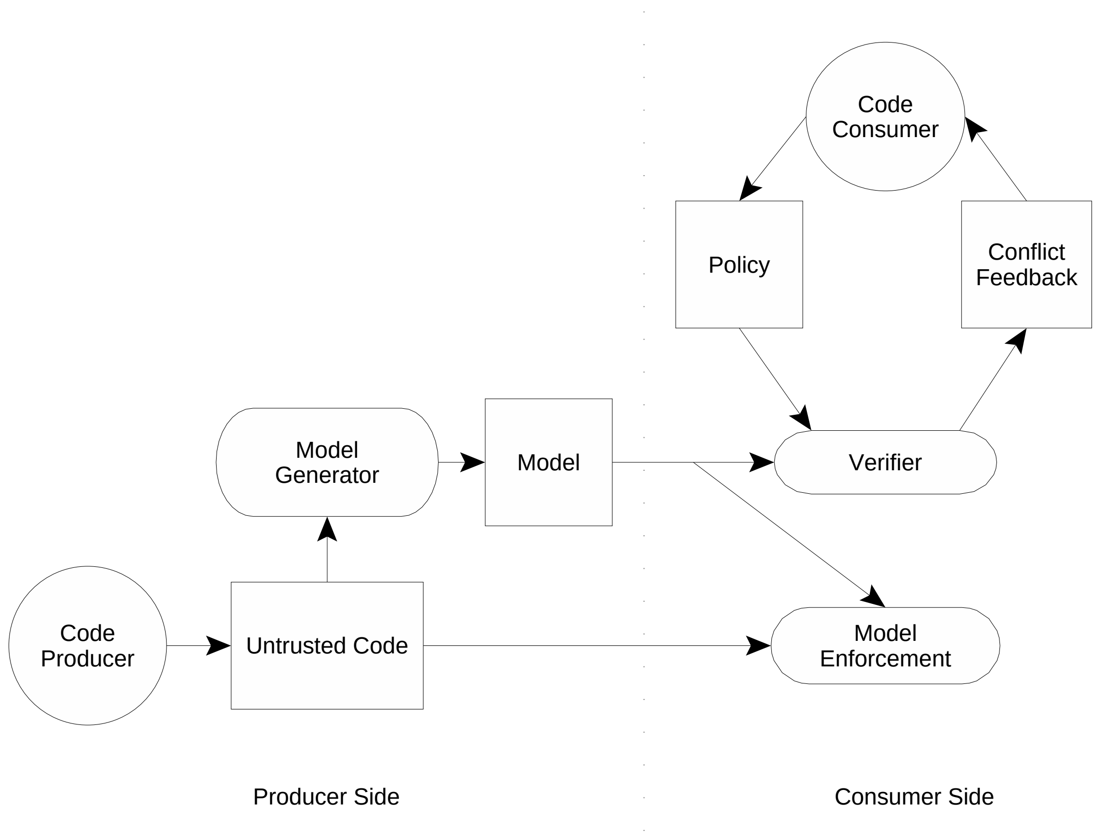

Figure 1: The model-carrying code framework.

Before executing the user can verify the model and check it against the local security policies. A report listing the policy violations is generated and enables the user to adapt her policies accordingly or in case the security vialotaion are severe, not to execute the code.

The model itself does not contain any information abaout the application's functionality. It only covers the security relevant behavior of the code.

A malicious code producer could try to generate a faulty model, which does not cover all aspects of the code. This problem is solved using runtime model enforcement. During execution of the code the behavior of the program is monitored and checked against the model.

Using this approach it is possible to reach the goals mentioned above. The producer and consumer are decoupled. The programmer does not need to know anything abaout the user's restrictions and the consumer is enabled to check the code's security need before execution.

Furthermore the extracted model helps to bridge the semantic gap between the low level binary code and the high level security policies.

In a more formal way the policy satisfaction and model enforcement can be described the following way:
B is a functions which maps the Model/Policy/Application to the actual behavior.

- Policy Satisfaction: $B[M] \subseteq B[P]$
  The behavior described in Model M has to be a subset of the behavior allowed by policy P.
- Model Safety: $B[A] \subseteq B[M]$
  The actual behavior of application A has to be a subset of the behavior described in model M.
- If both requirements are fullfilled, the wanted effect is reached: the application's behavior is allowed by the policy P: $B[A] \subseteq B[P]$

2.1 Security Policy Language

The language used by the MCC approach to describe security policies is able to express various security relevant aspects:

- Access control: An important category of policies is the control of access. There has to be control which resources can be accessed by the code and which not.
- Resource usage policies: In order to regulate usage of resources there has to be a mechanism to control the quantity of resource usage.
- History sensitive policies: An important feature of the language used, is the history sensitivity. History sensitive means that e.g. the program is only allowed to modify those files, which were also created by it.

The models are represented as Extended finite state automata (EFSA). These are Finite State Automata which have the ability to store values in a finite number of variables. A detailed explanation of Finite state automata is beyond the scope of this paper and can be found in default literature dealing with Logics.

The Behavior Monitoring Specification Language (BMSL) is used to describe the EFSA [3]. This language has originally been used for defining intrusion detection policies. A compiler exists which is able to translate BSML to an EFSA.

An example of such a policy can be seen in figure 2. The corresponding BMSL description is $any * .((socket(d, f)|d! = PF\_LOCAL)||FileWriteOp(g)$.
This policy allows any behavior and makes a transition to the final state if the program tries to connect to a non-local network resource or tries to open a file with write access.

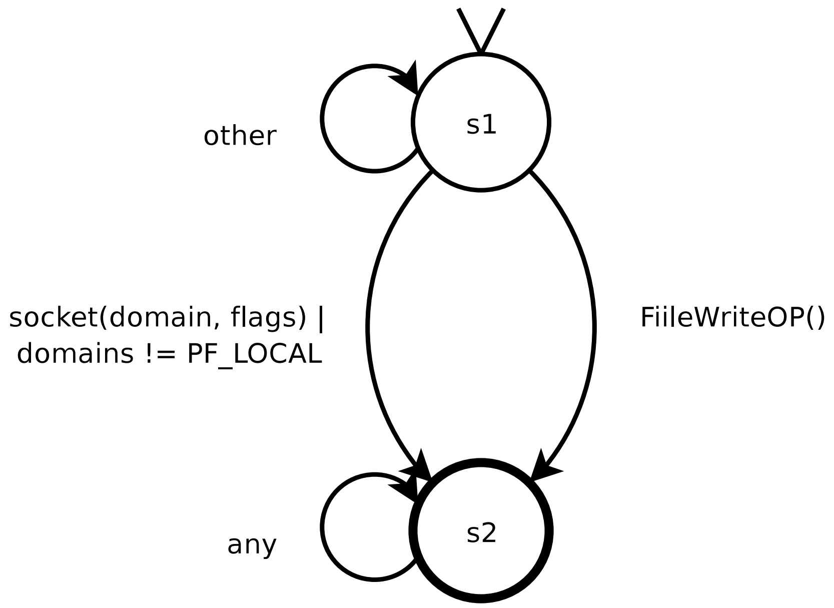

Figure 2: Example of an EFSA defining a policy.

A detailed description of the BMSL and more examples can be found in [**?**].

2.2 Model Extraction

The model extraction is performed by the code producer, which has access to the source code. The model should cover all security relevant aspects of the program. Sekar et. al. have found out in [1], that the sequence of system calls produced by the code covers the relevant behavior of the program.
Basically two approaches exist to extract a model.

2.2.1 Static analysis

The main idea of the static analysis approach is the compile time analysis of the source code. Models generated this way cover all aspects of the program. This has the the

advantage, that faithful code producers will always produce valid models.
The main disadvantage is, that it is very hard to check all values of variables at compile time. This is sometimes not possible at all, think of user interaction, e.g. a "Open File" dialog. It is not possible to predict these values in advance. In the case of the file dialog "All files" has to be assumed at compiletime. The produced models will become too conservative. It can even happen, that the model contains paths through the program which will never occur during real execution.

2.2.2 Execution Monitoring

The alternative to static analysis is to monitor the program's behavior during runtime. Every software engineer should have sufficient test cases for her code. Using these test cases every possible behavior of the program can be monitored. Without sufficient test cases aspects of the program will not be covered by the model and will cause execution aborts during runtime model enforcement on the consumer-side.

In [1] a FSA learning algorithm is described. The basic issues of this algorithm will be covered here:
The main idea is to generate a valid model out of the execution trace logged during runtime.
Spoken general this means building a FSA from a string.
In general it is a computationally hard problem to generate a FSA from strings. Ambiguous situations can occur which make it even impossible to generate an according model.
In our case the "strings" are execution traces. In order to have an unambiguous occurrance of system calls the location from where the system call is made is remembered.

The algorithm works as follows:

Every time a new system call occurs a new state labled with the location of the system call is created.
A new transition is made to the next location or to a final state, if the program terminates. The transition is labled with the occurred system call and its parameters. The following example will help to understand this approach:

```
1  S0;
2  while(..){
3    S1;
4    if (..) S2;
5    else S3;
6    S4;
7  }
8  S5;
9  S2;
```

Using two test cases the following system call traces can be observed executing the piece of pseudo code above:

```
1. S0/1 S5/8 S2/9

2. S0/1 S1/3 S2/4 S4/6 S1/3 S3/5 S4/6 S5/8 S2/9
```

The model in figure 3 is generated from these two traces.

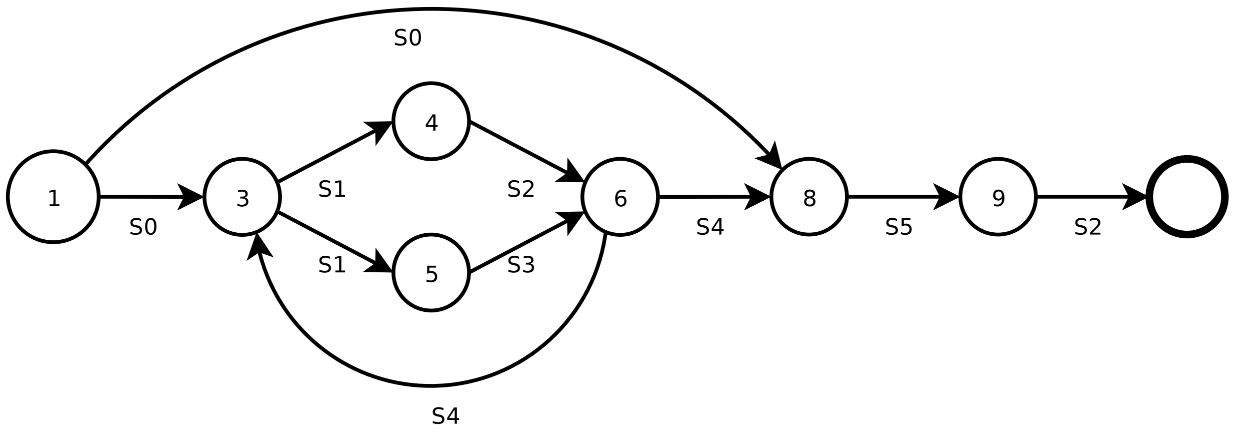

Figure 3: An example model generated during runtime execution.

The model extractor consists of two components. The online component intercepts the system calls. System calls are logged along with useful system call arguments. Values of buffers etc. are not considered as useful.
The offline component builds the FSA using the online component's logfile. In case that multiple arguments for one edge are observed a threshold is introduced. Whenever the number of arguments on one edge reaches this threshold, the arguments are accumulated (e.g. /tmp/* instead of /tmp/file1, /tmp/file2).

An important feature of the model extractor is to find out temporal relationships between system call arguments. For example the open and write system calls are in a temporal relationship. The write system call operates on files using a file descriptor, but for the model the actual file name is important. This filename is used by the according open system call. How these relationships are learned in detail is described in [?].

Finally the extracted model is sent along with the code to the consumer.

2.3 Verification

On the consumer's side the model has to be verified before execution. The user's policy has to be checked against the behavior described in the model.
The actual policy does not describe the behavior allowed, but the forbidden behavior. Formally this can be expressed by the negation of the Policy P:
$B[M] \subseteq B[P] \longleftrightarrow B[M] \bigcap B[\overline{P}] = \oslash$

In order to enable the user to adapt her policies all violating states have to be represented. This is done by building the product of $M \times \overline{P}$. All feasible paths of this product automaton lead to violating states.
This product is again projected onto the policy. Before the report is actually presented to the user all common aspects of multiple violating paths are combined. This is done the same way as the accumulation while extracting the model. Finally the user gets a report of the following form:

```
open operation on file /tmp/logfile in write mode
socket operation involving the domain PF_INET
```

The user now can refine her policies accordingly wether she wants to execute the code or not. All additionally needed rights have to be granted.

2.4 Enforcement

The MCC approach enforces the model during runtime. Basically it would also be possible to enforce the policy itself. This approach is not encouraged (although it is possible using the MCC framework) mainly for the following reason: The code producer could have modeled unwanted behavior in the model. Enforcing the policy, this behavior would not be covered.

During execution the model is enforced. This is done by an in-kernel module which intercepts the system calls and keeps track of the state in which the executing program is in. This causes an acceptable amount of overhead.

In case the executing code violates the model it is aborted. This is very unconvenient for the user and should only happen in rare cases. As most code producers are honest the models should be correct. It can be the case that the model does not cover all possible traces of behavior if the used test cases during model extraction were not sufficient.

3 Implementation

The MCC approach has been implemented by [1]. The model generation is realized using the Linux ptrace utility to intercept system calls. This causes a huge overhead of 40%-200% because of the context switches produced by ptrace. The model extraction can take up to a few minutes. But th model extraction is an offline activity which is only done once by the code producer.

The model verifier is built using the XSB logic programming system. It consists of 300 lines of prolog code.

The model enforcement is done using an in-kernel module as said before. During runtime an overhead of 2%-30% is needed. This overhead has to be low because it concerns the actual execution of the code.

3.1 Performace

In the following table measurements are presented. The data is taken from [1].

Application	Xpdf 1.0	Gaim 0.53	http-analyze
Size kb	906	3173	333
States	125	283	158
Transitions	455	937	391
Enforcement Overhead	30%	21%	2.4%
Verification msec	1	1.8	0.7

The xpdf pdf viewer has been tested with a conventional sandboxing policy which prevents the application from creating new files and disallows any modification of other files. Network access is also restricted.

The instant messaging application Gaim was not allowed to open any files except its own configuration scripts.

The http-analyze application was tested with a refined version of the policy in figure 2.

The most obvious difference between the applications can be observed looking at the enforcement overhead. The overhead is very low for the http analyzer tool. The other two applications have significantly more system calls than the web analyzer tool.

3.2 Integration

The MCC approach has synergy effects with existing systems. For example cryptographic signing can be used to produce signed models which gibes proof abaout the code producers identity.

An application called RPMShield uses the MCC approach. The application is used to manage packages with precompiled software for linux systems (RedHat Package Manager). RPMShild can be used to assure that during the installation of new packages no existing files are corrupted or manipulated.
It also is able to control pre- and postinstallation scripts.

4 Conclusion

According to Sekar et. al MCC is a very practical and useable approach which works with acceptable overhead. Several extensions like a catalog of policies for applications are planned. The verifier then can shoose the policy automatically which would improve user convenience.
Another point mentioned is the understandability of the report of violations. They are sometime not trivial and hard to understand for the average user. This feature could be improved too.
In my opinion MCC is indeed a useable approach for safe execution of untrusted code. Eventhough I want to add that many formalisms are used in the paper in order to describe the model and the corresponding description language. It is questionable if it this is really necessary. The same result could be reached by simpler means e.g. the EFSA, which is in practice an execution flow chart should be also named that way. This would improve the understandability of the paper.

References

[1] R. Sekar, V.N. Venkatakrishnan, et. al.; *Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications*, SOSP 03; 2003.

[2] G. Necula; *Proof-carrying code*, ACM Transactions on Software Engineering Methodology; 1999.

[3] R. Sekar, Prem Uppuluri; *Synthesizing fast intrusion prevention/detection systems for high-level specifications.*, USENIX Security Symposium, 1999.