Model Carrying Code

An approach for safe execution of untrusted applications

Presented by

Madhukar Anand

Background

- There has been a significant growth in the use of software from sources not fully trusted.
  - Document handlers and viewers
    - Real audio, ghostview.
  - Games, P2P applications
    - File-sharing, Instant messaging.
  - Freeware, shareware, trialware, mobile code.
- "How can we trust the code?"

State of the Art

- Very little OS support for coping with such untrusted applications.
- Code Signing in recent OS's
  - Useful only in verifying code from trusted producers.
- Approaches towards handling untrusted code
  - Execution monitoring
  - Static analysis

State of the Art

- Execution Monitoring
  - Policy violations are detected at runtime
  - User prompted for additional access
    - Unclear whether this solves the problem
  - Terminate the program.
    - Causes Inconvenience, Initiate clean-up.

- Static Analysis
  - No runtime aborts, but…
  - Only effective when operating on source code. Applications are typically binaries.

State of the Art

- **Proof Carrying Codes (PCC)**
  - code producer must prove code is "secure"
    - how does the producer know what is secure?
  - proofs are difficult to develop
    - In practice, used for simple properties, e.g., type safety

Need to combine convenience with enforcing consumer specified security policies.

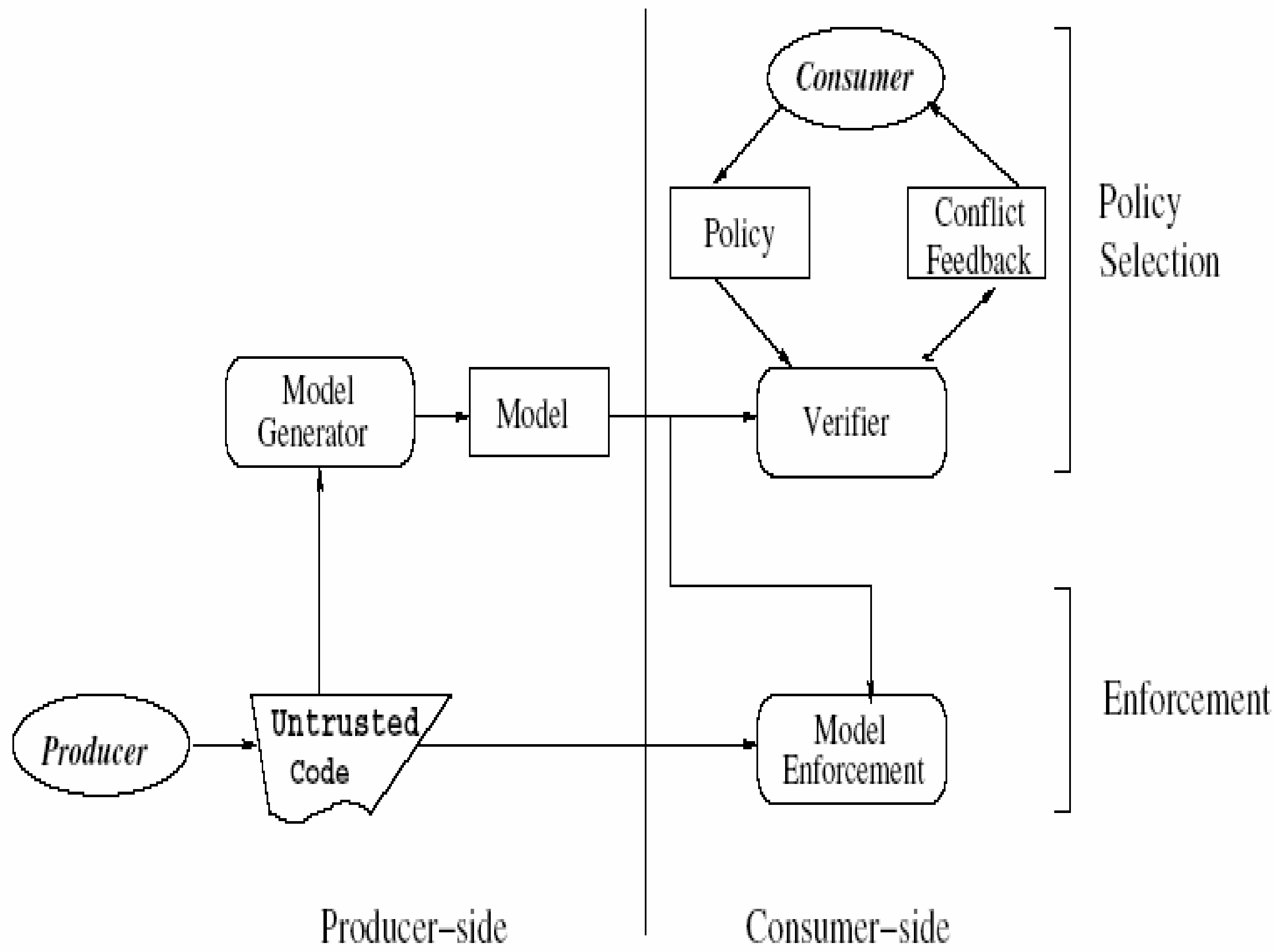
Need for new approach

- **Neither code producer nor code consumer can unilaterally determine security needs**
  - producer does not know consumer security policies
  - consumer does not know access needs of a program
- **Need an approach that enables the two parties to collaborate/coordinate for security**

Model-Carrying Code

- **Key idea: code producer provides code, plus a high-level model of its behavior**
  - model bridges semantic gap between low-level binary code and high-level security policies (of consumer)
  - producer need not guess consumer security policies
  - models being much simpler than programs, automation of consistency checking is feasible (between consumer policy and the model)

MCC Framework



Security Assurance in MCC

- **Security assurance broken down into:**
  - **Policy Conformance**
  - **Model Soundness**

- **Policy conformance: model satisfies policy**
  - $B[M] \subseteq B[P]$
  - since models are much simpler than programs, automated verification is feasible

Security Assurance in MCC

- **Model soundness: program behavior is consistent with the model**
  - $B[A] \subseteq B[M]$
  - Can use a variety of techniques
    - Runtime monitoring of system calls or resource-access ops
    - model-signing: producer vouches for accuracy of model
    - PCC: proof of model soundness

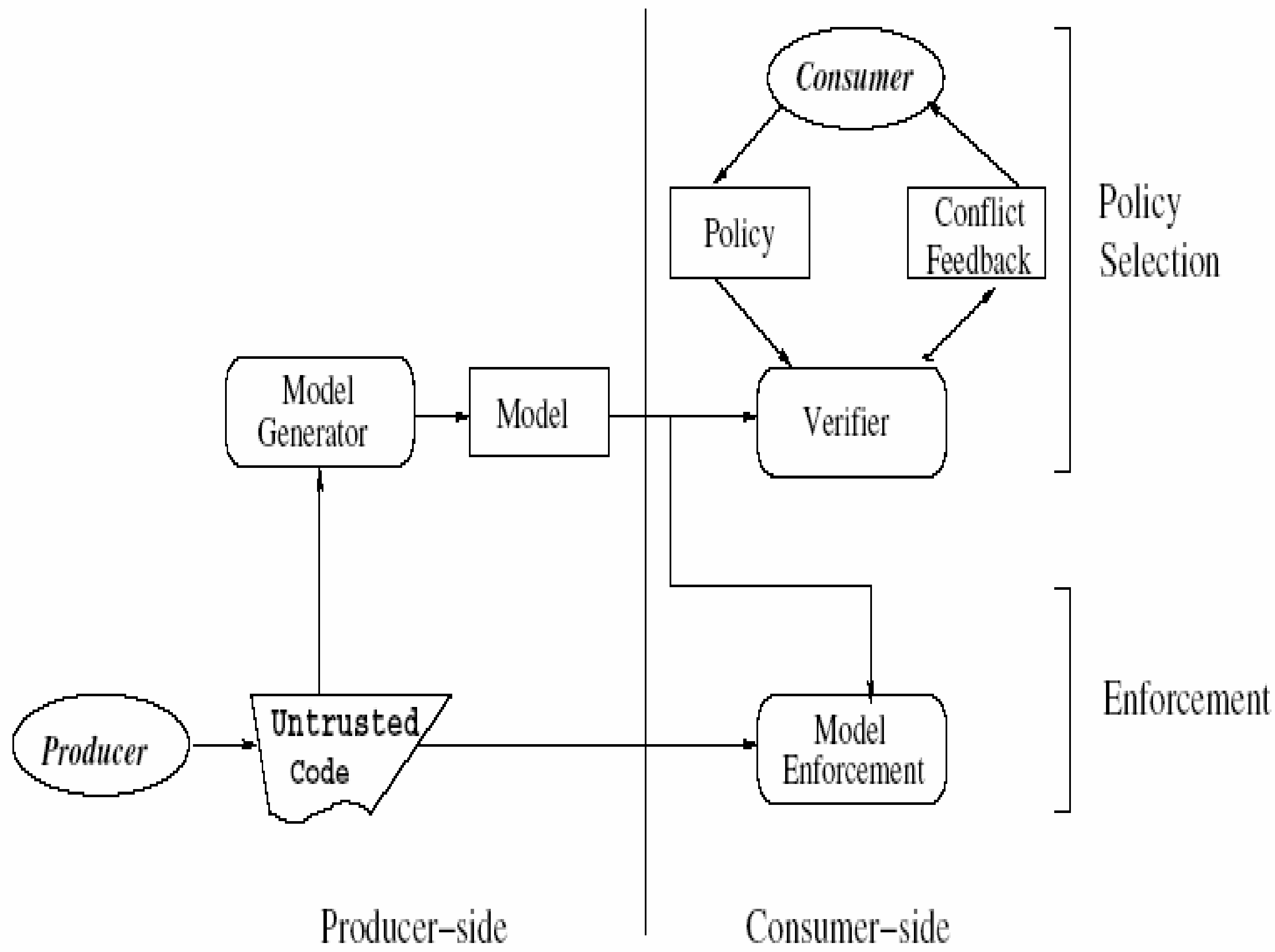
Outline

- Security Policies
- Model Generation
- Verification
- Enforcement
- Implementation and Conclusion

Outline

- **Security Policies**
- Model Generation
- Verification
- Enforcement
- Implementation and Conclusion

MCC Framework



Security Policies

- What are the policies of interest?
- How can they be specified ?

- Since enforcement relies on execution monitoring only enforceable properties are of interest (Safety Properties)
  - E.g. access control, resource usage

Security Policy Language

- Behaviors are modeled in terms of externally observable events.
  - E.g., System calls, function calls etc.
- Enforcement of policies will require secure interception of arbitrary system / function calls.
  - Not possible for function calls in binaries
- EFSA express negation of policies i.e. they accept traces that violate the intended policy.

Security Policies

- **The formalism used for specifying policy language is that of the *EFSA (or also using regular expressions)***
  - **The ability to remember arguments enhances the expressive power of the policy language.**
- **EFSA based policies are expressed in Behavior Monitoring Specification Language (BMSL)**
  - **Equivalently in Regular expressions over events**

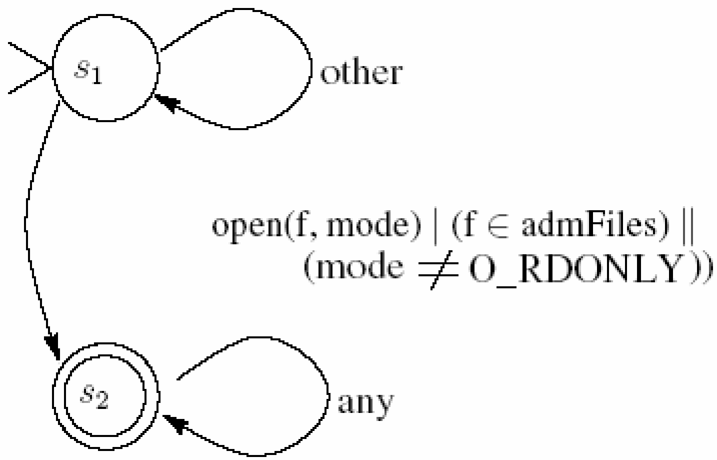
Security Policy Language

- Events are classified into
  - Primitive events
    - For system calls there are two associated primitive events: One corresponding to the invocation and the other to the exit
  - Abstract Events
    - Classes of primitive events
    - In general may be *patterns* of events
      - Different kinds of Patterns that are of interest are defined in the paper : Event occurrence, alternation, repetition, etc.

Examples

List admFiles = {"/etc/f1", "/etc/f2"};
any* · open(f, mode) | ((f in admFiles)
|| (mode != O_RDONLY))

admFiles := "/etc/*", "/var/*"

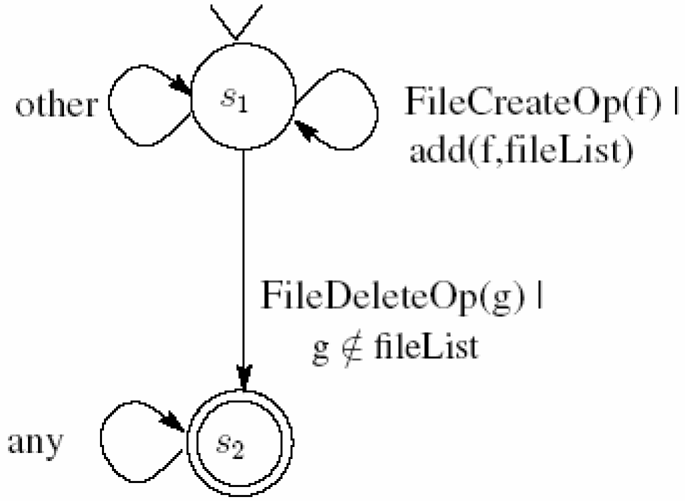


(a) Access control policy

Prevent writes to all files and reads from admFiles.

Examples

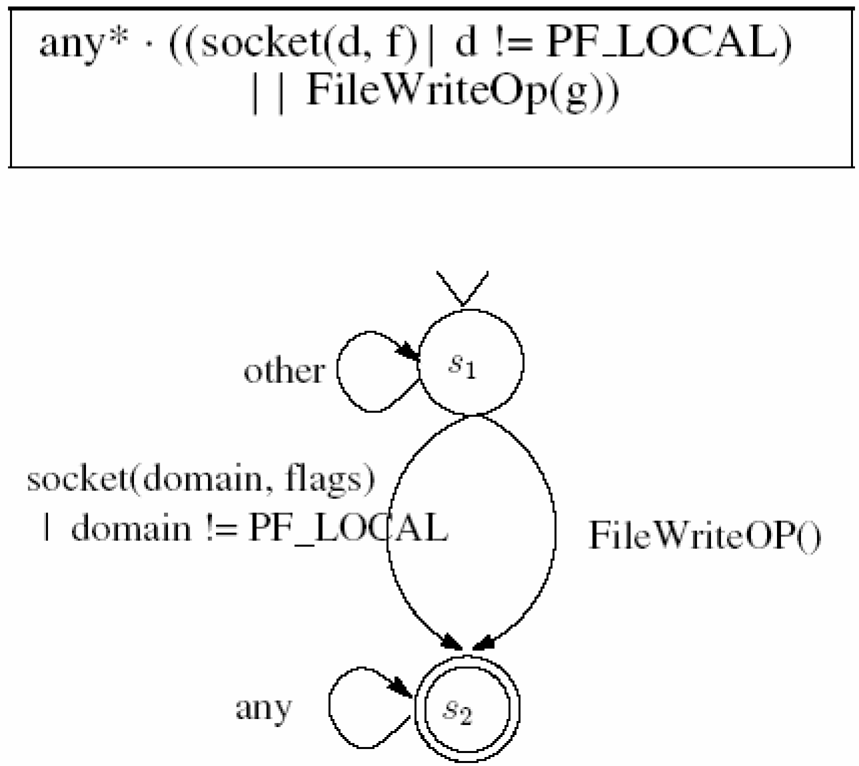
```
List fileList = {};
(FileCreateOp(f) | add(f, fileList) | | other)*
    · (FileDeleteOp(g) | !(g in fileList))
```



(b) History-sensitive policy

Delete only files that the application created.

Examples



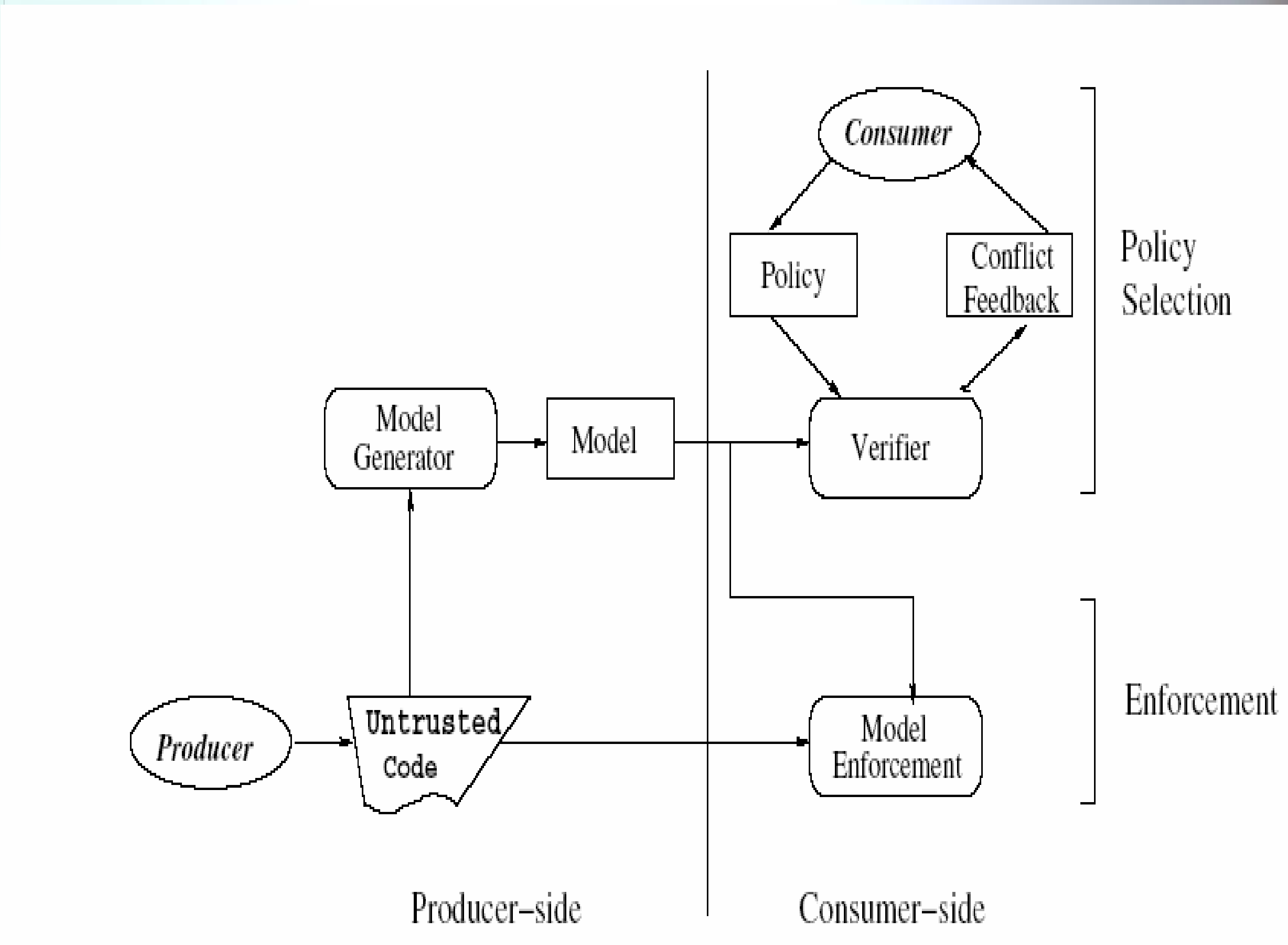
(c) Sensitive file read policy

No network access and no file writes

Outline

- Security Policies
- **Model Generation**
- Verification
- Enforcement
- Implementation and Conclusion

MCC Framework



Model Generation

- In MCC, the code producer generating the model is unaware of the consumer security policies.
  - A single model usable by all consumers must be generated by an automated process.
  - This bears more similarity with behavioral models for intrusion detection.

MCC uses model extraction via machine learning from execution traces.

Overview of the FSA Algorithm

- **Learning FSA from strings(traces) is computationally hard.**
  - **Strings do not give any clue to the state of the automata.**
    - **E.g. Looking at *abcda*, we cannot tell that the 2 a's correspond to the same state.**

- **Key Idea: State-related information can be obtained if the location from where the system call was made is known.**

Example

Example program

- S0;
- While(…){
- S1;
- If (…) S2;
- else S3;
- If (S4) … ;
- else S2;
- S5;
- }
- S3;
- S4;

Traces be

S0/1 **S3**/10 **S4**/11,

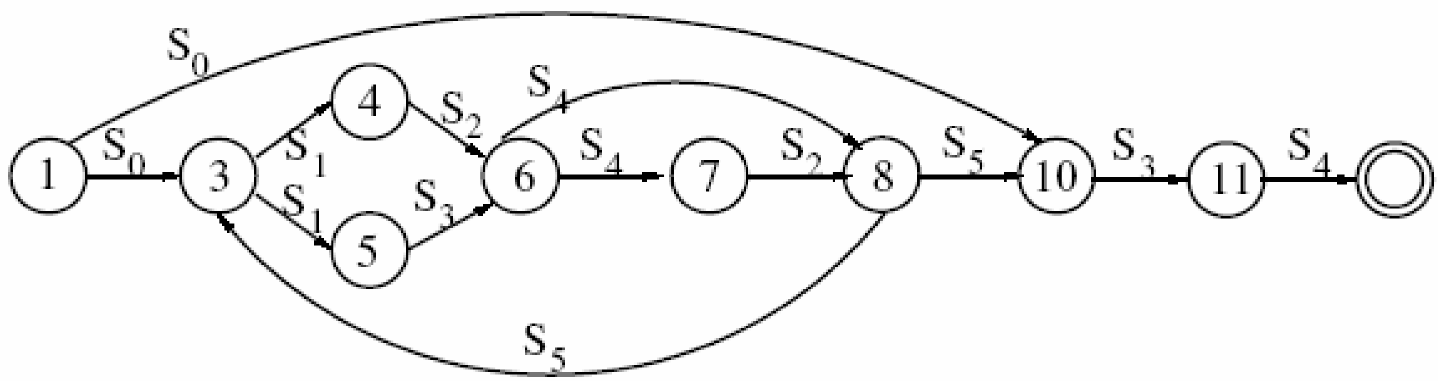
S0/1 **S1**/3 **S2**/4 **S4**/6 **S5**/8 **S1**/3 **S3**/5 **S4**/6 **S2**/7 **S5**/8 **S3**/10 **S4**/11.

Example

Traces be

S0/1 **S3/**10 **S4/**11,

S0/1 **S1/**3 **S2/**4 **S4/**6 **S5/**8 **S1/**3 **S3/**5 **S4/**6 **S2/**7 **S5/**8 **S3/**10 **S4/**11.



Model learnt from the above traces

Overview of the Algorithm

- The above notion of location has to be extended when dealing with libraries.
  - This is remedied by using the location within the executable from where the call was invoked.
    - Obtained by a “walk” up the program stack.

Overview of the Algorithm

- **The model extractor consists of an online and an offline component.**
  - **The Online component consists of a runtime environment to intercept system calls and a logger that records system calls and arguments into a file**
  - **The offline component has two parts : The EFSA learning algorithm and the log-file parser.**
  - **The learning algorithm is comprised of learning argument values and learning argument relationships.**

Learning Argument Values

- **There may be a need to learn absolute values (e.g., filenames)**

- **This is accomplished by recording values along with each system call. A threshold can be used beyond which the values are aggregated.**
  - **In principle, the algorithm should support a variety of aggregation algorithms but they claim that in practice there are only two: Longest common prefix and Union on sets.**

Learning Argument Relationships

- Important aspect here is learning temporal relationships.
  - Identify which pair of system calls needs to be considered.
- The algorithm relies on the fact that relationships of interest are those that have arguments of the same kind
  - E.g, we might be interested in equality of file descriptors but not in inequalities.
  - In their implementation only equality over integers and strings and prefixes and suffixes over strings are considered

Learning Argument Relationships

- **First, a distinct state variable is associated with the triple**

  (system call, invocation location, argument number)

- **Each variable that is a candidate for an equality relationship is stored in a hash table, indexed by its most recent value.**
  - **The hash table for different arguments will be different.**

Example

- **Separate hash tables for process ids and file descriptors**
- ***fd* will be associated with a list of variables whose most recent value is *fd*.**
- **When another system call with variable *v* with value *fd'* is made,**
  - *V* = lookup (*fd'*)
  - **If this is the first time, associate *v* with *V***
  - **If not, then, there is already a set *V'* associated with v. Hence associate $V \cap V'$ with *v*.**
  - **Delete previous value fd_{old} of *v* and add *v* to *V*.**

Note that relationships may weaken but never strengthened.

Learning Argument Relationships

- **For prefix and suffix relationships, a trie data structure is used. (It can be viewed as a tree-structured FSA for matching strings).**

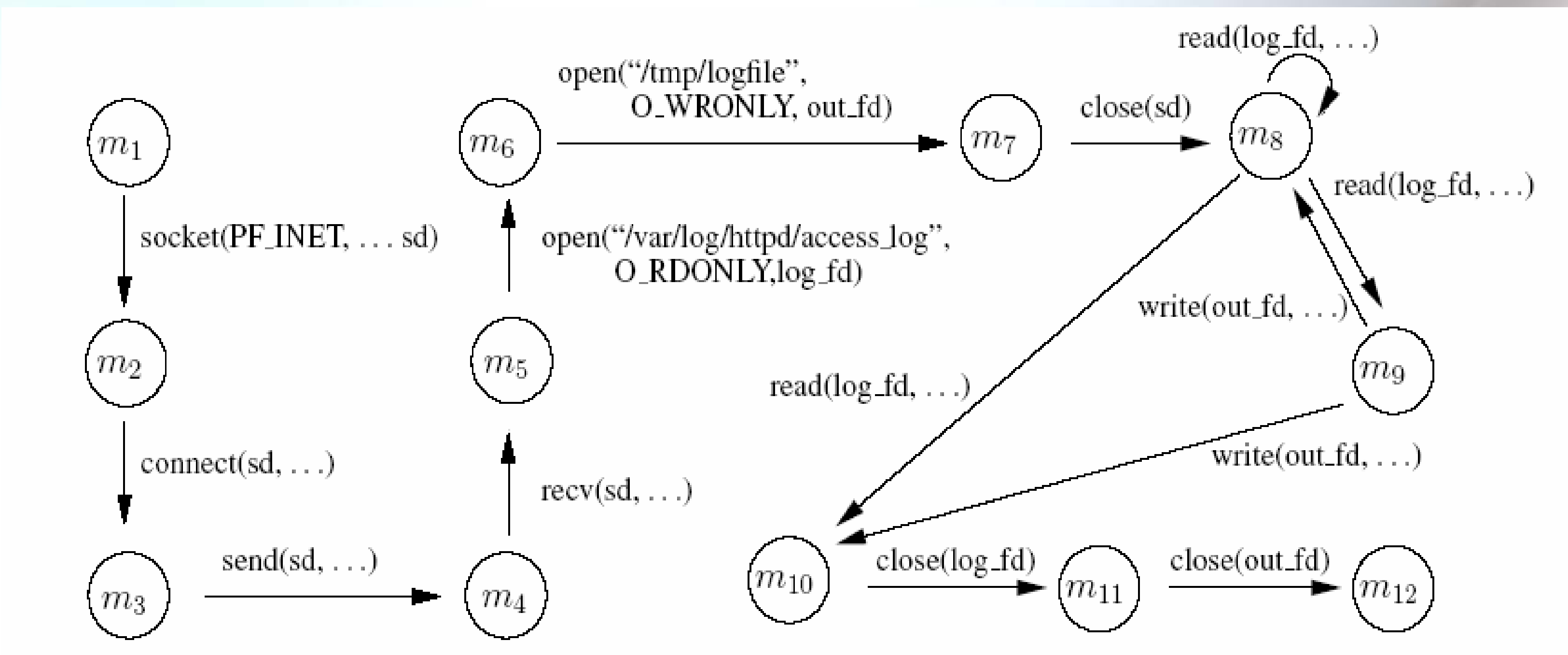
- **Finally a pruning mechanism is used to remove redundant relationships.**

Example

```
int main(int argc, char *argv[]) {
    int sd, rc, i, log_fd,out_fd,flag = 1;
    struct sockaddr_in remoteServAddr;
    char recvline[SIG_SIZE+1], sendline[SIG_SIZE+1];
    char buf[READ_SIZE];

    init_remote_server_addr(&remoteServAddr,...);
    init_sendmsg(sendline,...);
    sd = socket(PF_INET,SOCK_STREAM,0); ◀
    connect(sd, (struct sockaddr*)&remoteServAddr,sizeof(...)); ◀
    send(sd, sendline, strlen(sendline)+1,0); ◀
    recv(sd, recvline, SIG_SIZE,0); ◀
    recvline[SIG_SIZE] ='\0';
    log_fd = open("/var/log/httpd/access_log",O_RDONLY); ◀
    out_fd = open("/tmp/logfile",O_CREAT|O_WRONLY); ◀
    close(sd); ◀
    while (flag!=0) {
        i = 0;
        do {
            rc=read(log_fd,buf+i,1); ◀
            if (rc == 0) flag =0;
        } while (buf[i++] != '\n' && flag != 0);
        buf[i]='\0';
        if (strstr(buf,recvline) !=0)
            write(out_fd,recvline,SIG_SIZE); ◀
    }
    close(log_fd); ◀
    close(out_fd); ◀
    return 0;
}
```

Example

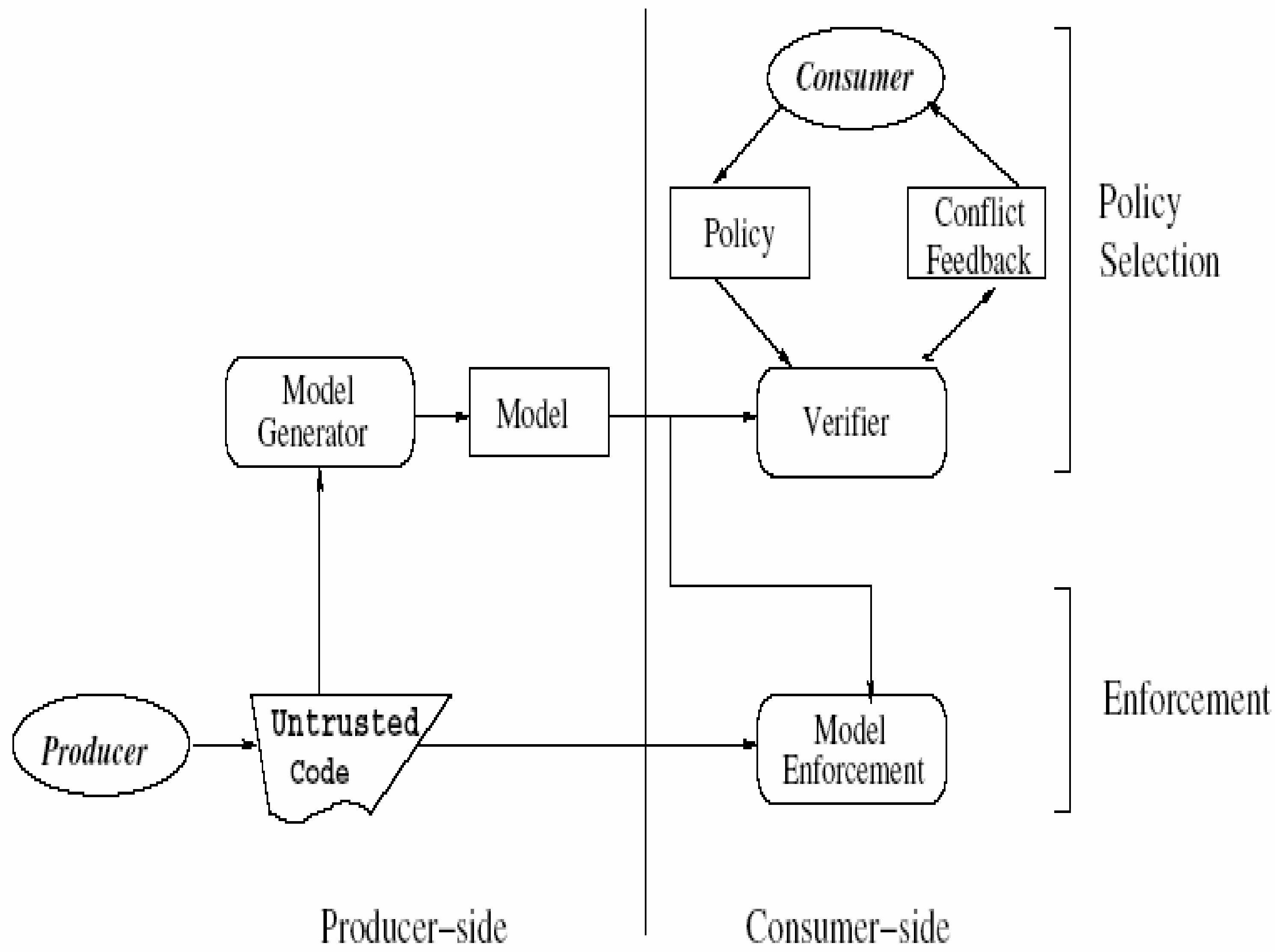


Model for the above program

Outline

- Security Policies
- Model Generation
- **Verification**
- Enforcement
- Implementation and Conclusion

MCC Framework



Verification

- It is concerned with determining whether or not a model M satisfies a security policy P
  - We need to check whether $B[M] \cap B[P^c]=\Phi$
- Thus the verification procedure has to build a product automaton $M \times P^c$.
- Then, if there are feasible paths to the final states, then the policy is violated and $M \times P^c$ is the representation of all such violations.

Verification

- EFSA's have infinite domain variables
- The computation of the product automata has to be extended from that of the FSA's.
  - The state variables of MP is the union of state variables in M and P.
  - The start state of MP is a tuple (m_0, p_0), $F_{MP} \subseteq F_M \times F_P$ is the final state set.

Verification

- $M : \delta(s, (e,C_1,A_1)) = s'$ and
  $P^c : \delta(p, (e,C_2,A_2)) = p'$ then (and only then)
  $M \times P^c : \delta((s,p), (e, C_1 \wedge C_2, A_1 \cup A_2)) = (s',p')$

- The general problem of satisfiability of arbitrary arithmetic constraints is undecidable for EFSA over infinite domain.
  - Restrict them to subsets containing = and ≠ relationships.

Conflict Presentation

- **Important to give the verifier a comprehensive view of violation.**
- **Due to the size of the product, “as is” view is not clear and precise.**
- **Present the violation by projecting it onto the policy automaton.**
- **Due to merges of transitions, a refinement of violation is presented.**

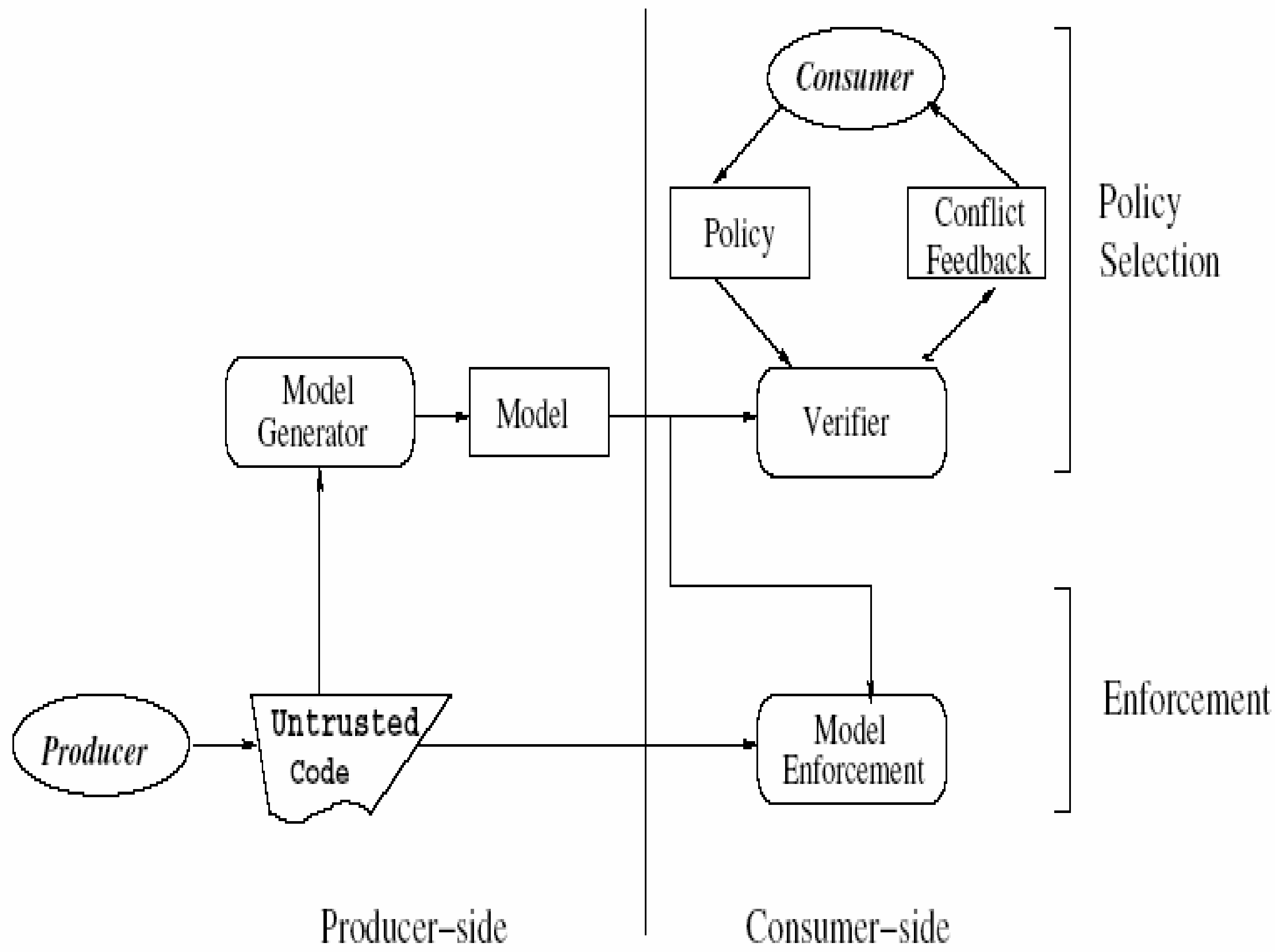
Example

- **Open events corresponding to different files may need to be combined above a threshold.**
  - **File names `/tmp/a1, … /tmp/a3, /etc/xyz, /var/f1, /var/f2` may be combined to `/tmp/a* , /etc/xyz, /var/f1, /var/f2`**
- **Even better : Use a catalog of polices, and present the ones that are compatible with given code.**

Outline

- Security Policies
- Model Generation
- Verification
- **Enforcement**
- Implementation and Conclusion

MCC Framework



Enforcement

- The runtime monitoring consists of intercepting system calls, obtaining argument values and matching them against models.
- If the application violates the behavior captured by the model, the enforcement module aborts the program. Then either,
  - Producer intentionally misrepresented the application behavior
    - Termination is the right choice here.

Enforcement

- Model does not capture all behaviors.
  - Termination may or may not be the correct choice but the only solution.
- In either case, safety is maintained.

- Policy Enforcement Vs Model Enforcement
  - Model EFSA captures a subset of behaviors of policy EFSA. Hence it is a conservative strategy.
  - Model EFSA are larger but deterministic.

Outline

- Security Policies
- Model Generation
- Verification
- Enforcement
- **Implementation and Conclusion**

Implementation tidbits

- Security Policies
  - Policy specified in BMSL.
  - BMSL specification compiled into EFSA.
- Model generation
  - Implemented using execution monitoring
  - Offline process and hence not optimized
- Verifier
  - XSB Prolog implementation (supports memoization).
- Model Enforcement
  - Uses a in-kernel module to perform system call interposition.

Results

Application	Program Size (KB)	Model Size			Enforcement Overhead		Verification	
		States	Transitions	Relationships	Interception only	Total	Time (msec.)	Space (MB)
xpdf 1.0	906	125	455	305	2%	30%	1.00	0.5
gaim 0.53	3173	283	937	432	2%	21%	1.80	0.7
http-analyze 2.4.1.3	333	158	391	247	0%	2.4%	0.70	0.4

MCC Conclusion

- **Supports code from untrusted producers**
- **Synergy with existing approaches**
  - Cryptographic signing
    - Signed models (certifying model soundness)
  - Proof-carrying code:
    - for verifying model soundness
- **Enables producers and consumers to jointly determine security needs.**
  - Mitigate security risks, while enjoying the functionality provided by mobile code

References

- R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar and Dan DuVarney**, Model -Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications,** ACM Symposium on Operating Systems Principles. (SOSP'03; Bolton Landing, New York; October 2003).

- Z. Liang, V.N. Venkatakrishnan and R. Sekar, **Isolated program execution: An application transparent approach for executing untrusted programs,** Annual Computer Security Applications Conference. Las Vegas, December 2003.