Model-Carrying Code (MCC): A New Paradigm for Mobile-Code Security*

[Extended Abstract]

R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan and S.A. Smolka
Department of Computer Science
SUNY at Stony Brook, NY 11794
{sekar,cram,ram,sas}@cs.sunysb.edu

ABSTRACT

A new approach for ensuring the security of mobile code is proposed. Our approach enables a mobile-code consumer to understand and formally reason about what a piece of mobile code can do; check if the actions of the code are compatible with his/her security policies; and, if so, execute the code. The compatibility-checking process is automated, but if there are conflicts, consumers have the opportunity to refine their policies, taking into account the functionality provided by the mobile code. Finally, when the code is executed, our framework uses runtime-monitoring techniques to ensure that the code does not violate the consumer's (refined) policies.

At the heart of our method, which we call *model-carrying code* (MCC), is the idea that a piece of mobile code comes equipped with an expressive yet concise model of the code's (security-relevant) behavior. The generation of such models can be automated. MCC enjoys several advantages over current approaches to mobile-code security. Succinctly put, it protects consumers of mobile code from malicious or faulty code without unduly restricting the code's functionality. Moreover, the MCC approach is applicable to the vast majority of code that exists today, which is written in C or C++. This contrasts with previous approaches such as Java 2 security and proof-carrying code, which are either language-specific or are limited to type-safe languages. Finally, MCC can be combined with existing techniques such as cryptographic signing and proof-carrying code to yield additional benefits.

General Terms

Security, Information assurance

Keywords

Mobile code security

1. INTRODUCTION

Mobile code has become an integral part of the Internet. It appears in many forms, such as "active pages" (e.g. pages with Java, Javascript, VBScript, or ActiveX content), content that invokes plug-ins or helper applications (e.g. Word, Excel, Postscript and Powerpoint documents or email attachments), or software that is explicitly downloaded from a freeware or commercial site. Since mobile code gets executed with the privileges of the user who downloaded the code (henceforth referred to as a *consumer* of the mobile code), the risk of damage due to malicious or faulty mobile code is very high. In this paper, we are concerned only with the risk to the code consumer, and do not address the issue of risks to the producer due to a (malicious) consumer.

1.1 State-of-the-Art in Mobile Code Security

Many of the techniques currently deployed in computer security are not effective when it comes to mobile code. Approaches such as *sand-boxing* can provide security, but only at the cost of unduly restricting the functionality of mobile code (e.g., the code is not permitted to access any files). *Cryptographic code-signing* can certify the origin (i.e., *the producer*) of mobile code and its integrity, but does not address the fundamental risk inherent to mobile code, which relates to *mobile code behavior*. This leaves the consumer vulnerable to damage due to faulty code (if the producer can be trusted), or malicious code (if the producer cannot be trusted).

To address these inadequacies, several new approaches have recently been developed to tackle mobile-code security. The *Proof-carrying code (PCC)* approach [12] enables safe execution of code from untrusted sources by requiring a producer to furnish a proof regarding the safety of mobile code. A consumer can mechanically check the correctness of this proof, and execute the code only if the proof is correct. The main practical impediment in using this approach is the difficulty of developing proofs, especially when they have to be machine-checkable, and moreover, operate on a binary representation of code. Therefore, they propose that such proofs be automatically generated by a compiler from the source code representation of the code [13]. While automatic generation of proofs is possible for simple properties

*This research is supported in part by a ONR University Research Initiative grant N000140110967, and NSF grants EIS-9705998, CCR-9876242, IIS-0072927 and CCR-0098154.

such as memory safety, automatic proof generation for more complex properties is a daunting problem. Apart from this practical difficulty, there is a more fundamental difficulty with PCC: since the producer needs to send the safety proof together with the mobile code, the PCC approach assumes that *the code producer knows all the security policies that are of interest to consumers.* We believe that this is an unrealistic assumption, since security needs vary considerably across different consumers and their operating environments.

Whereas PCC places the burden on the producer to identify and prove safety properties of interest to consumers, the *Java security model* [5] shifts the burden entirely to the consumer side. Specifically, Java 2 provides an access control mechanism that can limit resource access based on the identity of the code producer, and possibly the identity of code consumer [10]. However, the policies themselves are decided solely by the code consumer without any involvement of the producer. Thus, this model assumes that *the consumer can determine the access requirements of a mobile application based on its origin, even without any knowledge about the application.* This assumption either leads to an undue restriction in functionality of the mobile code, or leads to a situation where some applications are given more access than what they need. For example, a consumer would clearly be willing to allow a data-visualization program to read the (possibly sensitive) files containing the data to be visualized. On the other hand, the consumer would be unwilling to let a different program, such as one that collects customer feedback using a form and sends it back to the code producer, to read such files.

1.2 Need for New Approach

The main difficulty with existing approaches is that neither the producer nor the consumer can unilaterally determine the security needs of a mobile program. A producer of mobile code cannot anticipate the security requirements of the consumer, since each consumer may have his/her own security requirements and policies. Similarly, the consumer cannot anticipate the access needs of a piece of mobile code as these will depend on the functionality of the code and on how it is implemented.

An ideal mobile-code security framework would enable a consumer to formally reason about the security-relevant actions of a piece of mobile code; check if these actions are compatible with his/her security policies; and, if so, execute the code. The compatibility-checking process would be automated, but if there are conflicts, consumers would have the opportunity to refine their policies, taking into account the functionality provided by the mobile code. Finally, when the code is executed, the framework would assure that the code does not violate the consumer's (refined) policies. We propose a new approach, called *model-carrying code* (MCC), that seeks this ideal.

MCC is not proposed as an alternative to techniques such as PCC or Java security. Rather, MCC fills a void that is not addressed by previous approaches. It enables both the consumer and producer to coordinate in determining the security needs of mobile code. Techniques such as PCC are currently limited to low-level security properties such as memory safety, and the MCC framework can continue to exploit PCC for establishing such properties.

2. OVERVIEW OF APPROACH

The key idea in our approach is the introduction of program behavioral *models* to bridge the semantic gap between (very low-level) binary code and high-level security policies. These models capture security-related properties of the code, but do not capture aspects of the code that pertain to its functional correctness. These models are then sent by the code producer to the code consumer, together with the program (mobile code). Since these models are much less complex than programs[1], it is feasible for a consumer to mechanically determine whether a model conforms to security policies of interest. Based on the outcome of this check and the intended functionality of the code, the consumer can then refine his/her security policies and retry. Moreover, the producers no longer have to know or guess the security policies of interest to consumers. Instead, they provide models of security-relevant program behaviors that can be used to reason about most security properties of interest to any consumer. The models themselves may be developed either manually, or by using automated techniques that operate on programs.

The use of models enables us to decompose the security-assurance argument into two parts:

- *policy conformance:* check whether the model conforms to the policy
- *model soundness:* check if the model represents a safe approximation of program behavior. Our notion of soundness will be based on the particular execution of the program that takes place at a consumer site, rather than being based on all possible executions.

Note that the second part is necessary because the consumer does not necessarily trust a producer. In particular, the producer may provide an incorrect model (i.e., a model that does not correspond to the security-relevant behavior of mobile code) either due to malice, or due to errors in the model generation process.[2]

The above decomposition of security assurance argument broadens the choice of techniques that can be used to assure security. For instance, a consumer may rely on formal verification to assure policy conformance. Models being much simpler than programs, such automated verification is feasible. For establishing model soundness, a consumer may rely on one of the following techniques:

- *runtime-checking:* the consumer can monitor execution of the mobile code, and affirm that its behavior is

[1]For instance, our model for a large program such as the Washington University FTP server, contains about 200 states, as compared to the source code size of several thousands of lines.

[2]Such errors may arise due to human error or bugs in an automated procedure for model extraction. They may also occur because the environment in which the code is run may differ between the producer and consumer, thereby manifesting behaviors at the consumer that are different from behaviors observed/expected by the producer.

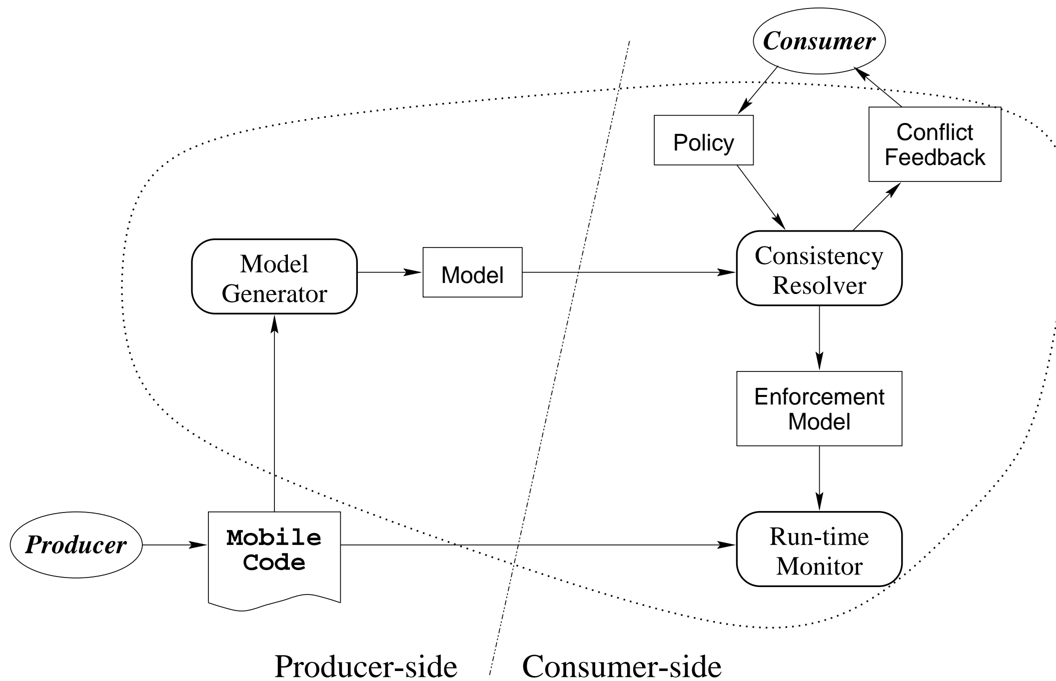

Figure 1: The Model-Carrying Code Framework

consistent with the model. Efficient runtime checking is feasible when policies are specified in terms of externally observable events, such as system calls made by a program to access OS resources [19].

- *model-signing:* the code and the model may be cryptographically signed by the producer to ensure their authenticity and integrity. The consumer may then trust the producer's representation that the model is sound. Although such *model-signing* bears some similarity to code-signing, there is an important difference. The notion of trust is much more clearly defined and narrower in the case of signed models: that the consumer trusts the producer to provide a model that faithfully captures the security-relevant actions of the code.

- *proof-carrying code:* a producer may provide a formal, machine-checkable proof that the model is sound. This proof can be checked by a consumer before the model is accepted as being accurate.

Of these techniques, the first and third allow a consumer to accept and execute code from untrusted producers, while the second technique works only with producers that are trusted by the consumer. A combination of these techniques may also be used.[3]

Figure 1 illustrates our approach. In the figure, the *model generator* is responsible for generating a model of security-relevant behavior of the program. Such a model would capture all of the security-relevant operations made by the program, as well as the temporal relationships between these operations. The model may also capture information flows in the program, although this aspect is not explored further in this paper. Both the code and the model are then sent to the consumer side, where a *consistency resolver* checks whether the model conforms to security policies selected by the consumer. When a model does not conform to a policy, the consistency resolver generates a "difference" between the model and security policy, which will then be presented to the consumer for further resolution, as shown in the "conflict feedback" loop in the figure. Alternatively, this difference may be combined with the model to produce an *enforcement model* that is given to the *runtime monitor.* The runtime monitor is responsible for confining the execution of mobile code so that it conforms to the enforcement model. At the first instance when the program deviates from the model, it may be terminated. Alternatively, the consumer may be prompted about the deviation, and queried whether the deviation is to be permitted. The runtime monitor may provide recovery capabilities to undo the effects of partial execution of the mobile code.

We expect the runtime enforcement to be a deterrent mechanism against attacks where a producer supplies an invalid model. Knowing that such attacks would be thwarted during the execution of mobile code, attackers would look towards other ways to attack a consumer. This means that in practice, models would be sound, and hence the primary decision point for acceptability of mobile code is the consistency resolver.

3. AN EXAMPLE SCENARIO

Consider the mobile application `webstat`, a freeware program that is obtained from an untrusted source. `webstat` gathers and presents usage statistics from Web-server log

[3]We note that some classes of properties are more easily supported using one of these techniques as compared to another. For instance, runtime-checking can easily support properties involving resource usage (e.g., CPU time used), whereas model-signing and possibly PCC approaches can provide better support for information flow properties. We also note that resource usage policies cannot be verified at the consistency resolution stage, but can be easily enforced at runtime.

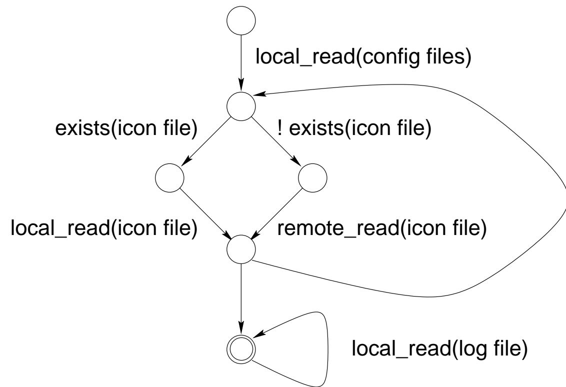

Figure 2: Model of `webstat`

files. For displaying the results, it downloads platform-dependent icons and/or plugins over the network. In the rest of this example, we assume that the security policies of the consumer are defined on a site-wide basis, and hence refer to "site policies" as opposed to "consumer's policies."

The consumer site considers the contents of Web-server log files to be private and wants to protect them from being exported. In our example, this security requirement is initially stated as *policies* that classify mobile applications as *file-only* or *communication-only*. File-only applications can read all files but have no network access, and are very limited in terms of write operations on files. Communication-only applications have network access but cannot access any files.

In the MCC approach, the code for `webstat` comes equipped with a behavior model. In our example, the model is the automaton shown in Figure 2. The model is expressed as an extended finite-state automaton (EFSA), i.e. a finite-state automaton whose states are annotated with data variables and values, and whose transitions are annotated with events and conditions on event arguments. The model in the figure is an abstract version of the producer-supplied model. The full model is given in terms of lower-level events such as system calls, and also has transitions on other events such as writes to temporary files. We have chosen to present an abstract, high-level version of the model to simplify our presentation.

Clearly, `webstat` is neither a file-only nor a communication-only application, and hence violates the security policies. The consistency resolver detects this violation and informs the consumer that a violation of the policy arises due to the fact that `webstat` makes a network access. The consumer, at this point, has the option of getting further information from the consistency resolver regarding the conflict, such as a complete scenario that illustrates the conflict. This information can be used to revise the policy. A less sophisticated consumer may choose to rely on a hierarchy of security policies that have been pre-defined by a local security administrator to aid in policy refinement. Suppose that this hierarchy provides several refinements to the "file-only" policy, one of which is *no access to security-critical files, and no external network access <u>after</u> read from sensitive files.* Note that the revised policy reduces access to certain operations (e.g. reads on security-critical files), while increasing access to certain other operations (e.g. send operations over the network).

Also, in the revised policy, the notions of which files are considered sensitive (or security-critical), and which hosts are considered external, is site-specific. In this case, the Web-server log files are considered sensitive, while a file that contains access permissions for remote access (e.g. `/etc/hosts.deny`) may be considered security-critical. In addition, the revised policy illustrates the ability of our approach to capture temporal behavior. Our language for representing security policies will also be based on EFSA, but this automaton will typically operate over higher level events (e.g., "read from sensitive files") than those used in the model EFSA. Each high-level event will itself be defined in terms of an EFSA on low-level events such as system calls, and hence it is possible to translate the policy EFSA into one that operates on low-level events used in the model EFSA.

The model of `webstat` satisfies the refined policy and hence `webstat` can be run. In general, however, the consistency resolver may be able to prove the property only with additional constraints on the producer-supplied model. For instance, the producer supplied model may suggest that the program may read arbitrary files from the `/var/log/` directory, while the security policy may allow only reads from the `/var/log/httpd` directory. In this case, the consistency resolver would indicate that the model satisfies the policy, provided the file accesses are restricted to `/var/log/httpd` directory. In the worst case, the consistency resolver may not be able to verify the policy at all. In either case, the consumer may wish to run the code. In order to make sure that the code cannot violate the security policy, the consistency resolver generates an *enforcement model*, which captures behaviors that are permitted by the producer-supplied model as well as by the consumer-selected security policy. By monitoring runtime behavior using the enforcement model, we can ensure that a run of the code cannot violate the consumer's security policy.

3.1 Features of the MCC Framework

As illustrated in Figure 3, the model-based approach enforces security in three steps: (1) by verifying that the model of the mobile code satisfies the security policies, (2) by generating an enforcement model as a result of the verification run, and (3) by ensuring that a run of the code conforms to the enforcement model. The satisfaction relation, represented in the figure as "$\models$", means that *every* run of the model is consistent with the policy. The conformance relation which talks only about particular runs of the code is represented in the figure as "$\Rightarrow$". A more direct approach is to ensure, by runtime monitoring, that a run of the mobile code conforms to security policies. Several key advantages of MCC over existing technology as well as a direct-monitoring approach, are apparent from the above scenario.

- A mobile application such as `webstat` cannot be securely run using current technology. For instance, proof-carrying code is not applicable since the property to be proved is site specific (e.g. what are sensitive files?); hence the proof cannot be provided by a producer oblivious to the consumer's requirements. The Java security architecture, as well as a number of other proposals on mobile-code and mobile-agent security,

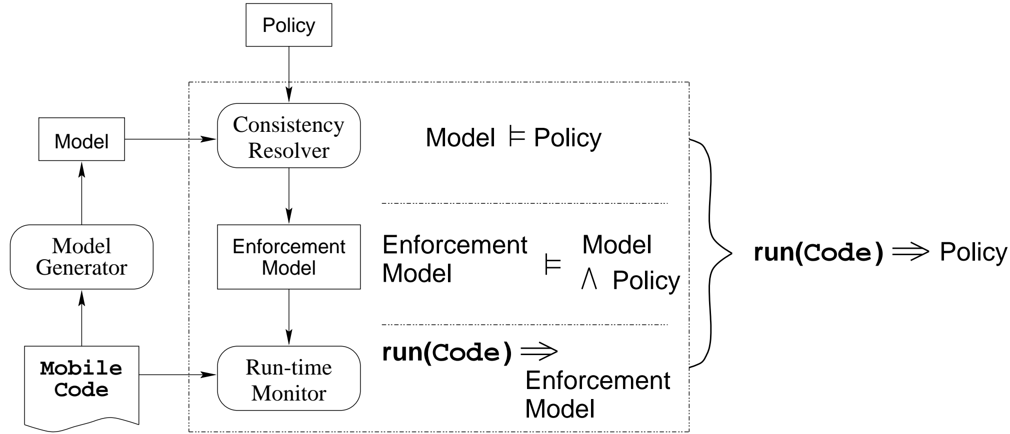

Figure 3: Logical view of the Model-Carrying Code Framework

are based on a refinement of traditional access-control mechanisms. They cannot express the temporal aspects of permissions (e.g. no network access after read from...). Moreover, the access-control decisions are made based on the wishes of the code producer and consumer, with no regard for the functionality provided by the mobile code.

- If runtime monitoring is used as the sole means of ensuring security, an application must be run "in isolation" so that its effects are observable to the outside only when its execution satisfies the security policies. Isolation, rollback, and commitment are difficult to achieve when applications communicate with the external world.

- The feedback offered by the model-based approach is crucial for refining security policies. It should be noted that security policies may be refined in different ways, depending on the application at hand. For instance, the same site in the above scenario may want to run a SATAN-like application to look for system vulnerabilities. For such applications, it is conceivable that the policy to be enforced would allow read access to the entire file system, but disallow writes of any kind except to the screen and/or to a specific output log file.

4. REALIZING MCC

In this section, we outline our technical approach for realizing each of the components of the MCC framework. A comprehensive treatment of each of these areas is outside the scope of this paper. What we attempt here is to try to convince the reader that each of the components *can be realized.*

The starting point for model-carrying code is our work on specification-based intrusion detection [1, 19]. This approach is based on specifying security-relevant behavior of programs in a high-level language called Behavior Monitoring Specification Language (BMSL). We model behaviors of programs in terms of systems calls made during execution. At runtime, the execution of these programs is monitored, and any deviations from specified behavior are flagged as intrusion efforts. Since system calls can be observed externally from a program, the approach can be used for COTS software without modification. Our research to date has shown that (a) BMSL enables convenient and concise specification of security-relevant program behaviors, and (b) runtime monitoring can be performed with very low overheads (5% or less) [1, 19]. Many of the techniques described for realizing the different components of MCC are based on this research.

4.1 Modeling Language

As described in the example, we use extended finite-state automata (EFSA) to represent program models [19]. EFSA are simply standard finite state automaton (FSA) that are augmented with the ability to store values in a fixed number of *state variables,* each capable of storing values over a finite or infinite domain. The state of the EFSA is thus characterized by its *control state* (which has the same meaning as the notion of "state" in the case of FSA), plus the values of these state variables. (Henceforth, the term *state* will be used to refer to the control state of an EFSA.) Transitions in the EFSA are each associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. For a transition to be taken, the associated event must occur and the enabling condition must hold. When the transition is taken, the assignments associated with the transition are performed.

The event alphabet of the EFSA will consist of system-call names. Since all access to resources is mediated by the operating system, and all applications obtain resource access through the operating system's system-call interface, expressing security-relevant behaviors in terms of system call sequences is a good choice. This hypothesis has been validated by many research efforts in intrusion detection, including our own.

While system calls are a natural choice for the event alphabet, this choice does not preclude other possibilities. For instance, in the context of Java, we may choose to model security-relevant behaviors in terms of higher-level operations, such as those that operate on I/O streams. Even within the context of programs written in C, one may choose to represent security properties in terms of operations on a higher-level API, such as the functions defined in `libc`.

Note that regular expressions, FSA, or ω-automata based approaches [15] can also express behaviors in terms of system-

call sequences. However, they lack the power to refer to system call arguments, e.g., they cannot capture the difference between the opening of a file in the `/tmp` directory or the opening of the password file. In contrast, EFSA can represent such distinctions. They can also represent properties that require system-call arguments used in the past, e.g., a program opens a file whose name was provided as a command-line argument (i.e., as an argument to an `exec` system call executed in the past).

4.2 Security Policies

Security policies will also be represented using EFSA. The primary difference between security policies and models is the alphabet over which they operate. Security policies will refer to much higher-level events than models, which would enable consumers to describe their policies at a higher level of abstraction than system calls. Moreover, the policies will be parameterized, so as to accommodate site-specific customization via instantiation of these parameters. For instance, we intend to capture a concept such as "read from a sensitive file" as a high-level event. This event is parameterized with respect to the set `SF` of sensitive files.

4.3 Runtime Monitoring

Runtime monitoring consists of intercepting security-relevant events, and matching them against models of expected behavior of mobile code. We have previously developed a system for runtime monitoring that operates on EFSA models and takes system calls as input [19, 1]. Our experiments show that runtime monitoring can be performed very efficiently, adding less than a 5% overhead to the execution time of most programs. We expect to be able to use this system for runtime monitoring for MCC.

Note that even if a program does not deviate from its model, it may still not have performed the computation expected by the user. For instance, a malicious program purporting to do file compression may remove its input file without producing a useful compressed file output. To deal with this problem, we can isolate the operations of mobile code in an environment where no other program can view the results of its computation. (If the mobile code executes as multiple processes, the unit of isolation includes all such processes.) After the mobile code completes execution, the user may check that the program performed as expected, and then commit the changes made by the code so that they are visible to the rest of the system. Clearly, such isolation may not always possible, e.g., the mobile code may communicate with remote sites. But for the more common case of removing or updating files, such isolation is achievable by intercepting system calls that open a file for writing and transparently redirecting that operation to a different file.

Although our existing runtime monitoring system operates on system calls, our approach is by no means restricted by this. It is relatively easy, for instance, to develop runtime monitoring techniques for Java programs by adding hooks into the JVM to intercept arbitrary function calls made by Java programs and feeding them into a monitor. Alternatively, the monitor could be used to replace the security-management related classes within the JVM.

4.4 Model Generation

Observe that model generation process has to balance the conflicting requirements of ease of consistency resolution (which argues in favor of "throwing away" as much information from the programs as possible) and the danger of leaving out information of interest to a consumer (which argues in favor of retaining as much information in the model as possible). We propose a trade-off that captures most of the security-relevant information of interest to consumers, while still being amenable to automated verification. We propose to express models of program behavior using (non-deterministic) EFSA. One way to generate such models is to *abstract* the source code of a program so as to retain only those portions that relate to system calls made by the program. An approach for deriving finite-state models using program analysis is described in [20], where these models are used for intrusion detection.

A drawback of approaches based on static analysis is that they are language-specific, thus necessitating redevelopment for each programming language. Moreover, for conventional languages such as C and C++, this approach suffers from the fact that we may not have source code access to libraries, especially those that are loaded dynamically. Finally, extending the approach to produce EFSA models (rather than FSA models) that can capture relationships between system call arguments is very difficult, due to limitations of program analysis. Therefore, we consider an approach based on machine-learning to be a more promising alternative. This approach has the additional benefit that it is obtained by observing the execution of a program under typical conditions, and as such, can be more accurate than compile-time techniques.[4]

We have already developed an approach for learning program behaviors as finite-state machines in the context of our previous work on anomaly intrusion detection [18]. Our approach generates compact models (containing a few to several hundred states, even for complex programs such as FTP and Apache web server). A limitation of our current approach is that it does not capture system-call argument values. An extension of our technique to address this limitation is currently under way.

4.5 Consistency Resolution

As described in the example, the consistency resolver is concerned with (a) verifying whether a model satisfies a policy, and (b) presenting the "difference" between them to the user, and help him/her refine the policy as appropriate. In this section, we concern ourselves only with (a). A possible technique to simplify user choices in (b) using a policy hierarchy was outlined in the example, but we do not discuss this any further in this section.

We rely on formal verification to determine whether a model satisfies a policy. Our techniques will be based on model-

[4]It must be noted, however, that the models learnt by runtime monitoring are not conservative. Thus, even if the model of a program satisfies a security policy, the program may in fact violate the policy. However, this factor does not negate the safety guarantees provided by the MCC approach. Through runtime monitoring, we would discover that the program is exhibiting behaviors inconsistent with the model, and abort it.

checking [2], a popular technique, originally proposed for verifying temporal properties of finite-state systems. Since the policies as well as the models are captured in the form of state machines, our techniques will draw on the automata-theoretic formulation of model-checking [9].

If M denotes the model of a mobile program, and P denotes a security policy, then verification amounts to checking if $M \Rightarrow P$. Noting that M and P are represented as state machines, we can think of the languages $L(M)$ and $L(P)$ accepted by these machines. Now, implication checking amounts to determining whether $L(M) \cap L(P)'$ is empty. (Here, $L(P)'$ denotes the complement of the language $L(P)$.) Note, however, that we are interested in the "difference" between P and M, as we wish to present this information to a user as part of conflict resolution. This difference is given by $L(M) \cap L(P)'$, so we will simply present this to the user. We discuss the computation of this difference below.

If M and P are represented using FSA (rather than EFSA), then operations such as intersection and complementation are straightforward. In the case of EFSA, we face the problem that such complementation and intersection problems may be undecidable in general. We tackle this problem in two steps. For complementation, we note that the security properties of interest are usually safety properties, which are of the form that "certain bad things do not happen." (In the example, we considered the property "a network write operation *does not occur* after a read of a sensitive file.") It is thus easier for users to specify an EFSA corresponding to the occurrence of the "bad thing" and state that this should not happen. Such an EFSA directly captures the negation of the property we require, and hence complementation is no longer an issue.

To tackle the problem posed by intersection of EFSA, we make use of the following approach. We simply use the standard FSA intersection algorithm on EFSA. Let M and P' be the two EFSA corresponding to the model and the complement of the security policy respectively. The EFSA D corresponding to their intersection is constructed as follows. The state variables of D consist of the union of state variables for M and P'. The initial state of D is the state (m_i, p'_i), where m_i and p'_i are the initial states of M and P' respectively. Now, we add new states and transitions to D as follows. For each state (s_1, s_2) in D such that there exists a transition on an event e from a state s_1 to s_3 of M and s_2 to s_4 of P', we add the state (s_3, s_4) to D (if this state is not already there). We also create a transition from (s_1, s_2) to (s_3, s_4) on e whose enabling condition is the conjunction of the corresponding enabling conditions in M and P'. The assignment operations associated with this transition are simply the union of the assignment operations on the corresponding transitions in M and P'.

The catch with this simple algorithm is that it may generate an EFSA that contains unrealizable paths. Thus, we may not be able to tell whether D accepts a nonempty language or not. At this point, we do not know whether this is a problem that is likely to be encountered frequently. For instance, this problem does not occur in several examples we have studied to date, including the one presented in this paper. When it does occur, the downside will be that the user is given the impression that the mobile code may violate a security policy when it does not. Clearly, this is much less serious than the case when a user is told that a model does not violate his/her policy when it does. Even so, we are currently investigating techniques to minimize such instances, by pruning away paths in D that are unrealizable. This research is based on our current work in infinite-state model checking.

5. IMPLEMENTATION STATUS

Of the components mentioned in the previous section, we already have prototype implementations of (a) the languages for expressing security policies and program models, (b) runtime monitoring, and (c) model generation. These implementations were taken from our previous research in intrusion detection [19, 7, 1, 18].

We have only recently begun the implementation of the consistency resolver, using our XMC model-checker [3, 16] based on the XSB system [21, 14]. So far, we have succeeded in verifying security properties for simple examples, such as the one described in this paper. We do not envision any problems scaling these results to larger examples, as the runtimes are adequate (in the range of tens to hundreds of milliseconds in our initial prototype), and because the algorithms in use are designed to provide good performance, possibly at the cost of being approximate.

We have also prototyped an implementation of the conflict resolver, where the technical problem is one of presenting the conflicts identified by the verifier in a user-friendly form. Our prototype is based on our earlier research in proof justification [17].

6. SUMMARY

In this paper, we presented a new approach that promises to lead to a comprehensive solution to the problem of mobile-code security, providing the following features:

- *Support for mobile code from untrusted sources.* The ability to enforce behaviors at runtime enables safe execution of code from untrusted sources. The runtime monitor can provide isolation capability so that changes made by a mobile application can be undone in the event of a policy violation, provided the application does not communicate with other applications or sites.
- *Secure mobile code "here and now."* PCC technology appears to be still far away from universal deployment, mainly due to source-language restrictions and the classes of properties that can be verified automatically. Java security is not applicable to the vast majority of mobile code that is written in other languages. In contrast, our approach is directly applicable to existing mobile code. Even in the absence of models from the producer, we can ensure security by enforcing the policies on the code directly at runtime.
- *Expressive language for specifying consumer security policies.* Our approach provides a high-level language in which security policies can be expressed concisely and conveniently. The language is expressive enough

to specify not only invariant properties, but also temporal properties such as "mobile code can overwrite or delete only those files it created previously," and "no operations to send data over a network are permitted after read operations on certain sensitive files." Such policies, which rely on sequencing relationships between different operations, cannot be expressed in existing frameworks for mobile code security such as Java.

- *Synergy with existing approaches.* As mentioned before, our approach can be combined with existing approaches such as cryptographic signing (for authenticity and integrity), and proof carrying code. With such combination, the role of runtime monitoring may be superceded by these mechanisms. The elimination of runtime checks can improve performance, but perhaps more importantly, will allow our approach to deal with properties that cannot be efficiently checked by monitoring security-relevant operations, e.g., properties relating to information flow. (Such properties would require us to reason about every assignment in the program.)

These capabilities are achieved in our approach without placing an undue burden on either the code producer or the consumer.

7. REFERENCES

[1] R Bowen, D Chee, M Segal, R Sekar, P Uppuluri, and T Shanbag. Building survivable systems: An integrated approach based on intrusion detection and confinement. In *DARPA Information Security Symposium*, 2000.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[3] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *Static Analysis Symposium*. Springer Verlag, 1998.

[4] S Forrest, S Hofmeyr, and A Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998.

[5] L Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley Pub Co, 1998.

[6] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[7] K Jain and R Sekar. User-level infrastructure for system call int erposition: A platform for intrusion detection and confinement. In *ISOC Network and Distributed System Security*, 2000.

[8] C Ko, G Fink, and K Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Computer Security Application Conference*, 1994.

[9] R Kurshan. *Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[10] C. Lai, L. Gong, L. Koved, A. Nadalin, R. Schemers. User Authentication And Authorization In The Java Platform. Annual Computer Security Applications Conference, 1999.

[11] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[12] G Necula. Proof carrying code. In *ACM Principles of Programming Languages*, 1997.

[13] G Necula and P Lee. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*, 1998.

[14] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.

[15] F. Schneider, Enforceable Security Policies, ACM Transactions on Information Systems Security, 3(1), 2000.

[16] C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification (CAV)*, 2000.

[17] A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, 2000.

[18] R. Sekar, M. Bendre, P. Bollineni and D. Dhurjati, A Fast Automaton-Based Approach for Learning Program Behaviors, IEEE Symposium on Security and Privacy, 2001.

[19] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 1999.

[20] D. Wagner and D. Dean, Intrusion Detection via Static Analysis, IEEE Symposium on Security and Privacy, 2001.

[21] XSB. The XSB tabled logic programming system. Available from `http://xsb.sourceforge.net`.