

# Safe Execution of Mobile and Untrusted Code: The Model-Carrying Code Project

R. Sekar      C.R. Ramakrishnan      I.V. Ramakrishnan      Scott Smolka  
Samik Basu      Sandeep Bhatkar      Abhishek Chaturvedi      Daniel DuVarney  
Zhenkai Liang      Yow-Jian Lin      Dipti Saha      Karthik Srinivasa Murthy      Weiqing Sun  
Alok Tongaonkar      Prem Uppuluri      V.N. Venkatakrishnan      Wei Xu      Mohan Channa  
Yogesh Chauhan      Kumar Krishna      Shruthi Krishna      Vishwas Nagaraja      Divya Padmanabhan  
Department of Computer Science,  
Stony Brook University, Stony Brook, NY 11794.

## Abstract

Starting from the Melissa email virus of 1999, threats posed by software from untrusted sources have grown enormously. Untrusted code can install spyware or steal confidential information, including identities and financial information. Even worse, it can turn an unsuspecting user’s computer into a so-called “zombie” that can be commandeered by an attacker to carry out criminal activities, including the launching of attacks on other systems on the Internet. These threats can all be eliminated by simply preventing users from accessing any untrusted code or data. However, such an approach isn’t practical: users have become accustomed to a wealth of information as well as software on the Internet that have significantly simplified their day-to-day activities and enhanced their productivity. Thus, the goal of our model-carrying code (MCC) project was to develop an infrastructure and software tools that enable users to access benign mobile code, while bounding their risks due to malicious mobile code. Our approach enables code producers and consumers to collaborate in order to achieve security, yet it doesn’t impose a significant burden on either one of them. This paper provides an overview of the MCC approach, surveys the scientific contributions of the project, and summarizes its practical outcomes.

## Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection, *Invasive software*

K.6.5 [Computing Milieux]: Management of computing and Information Systems, *Security and Protection, Unauthorized access*

## General Terms

Security, Verification

## Keywords

mobile code security, malicious code, sand-boxing, security policies

## 1 Introduction

The Model-Carrying Code (MCC) project began in 2001, just as the threat of cyber attacks due to untrusted content on the Internet began to rise. In the preceding decade, the primary threat to enterprise networks was perceived to be coming from the Internet. To counter this threat, most research (as well as products) focused on building a “hard exterior shell” around these networks, using technologies such as firewalls and intrusion detection systems. Unfortunately, such technologies aren’t effective against cyber attacks launched from inside an enterprise. While it may be justifiable to assume that insiders weren’t likely to intentionally attack their own networks, there is a high risk that they may do so *unwittingly*. The Melissa email virus of 1999, which took the form of an email attachment, was perhaps the first large-scale cyber attack that relied on this approach. When the email recipient attempted to view this attachment, malicious code embedded within the attachment was executed, causing copies of the virus to be sent to many other users. Several copycat attacks followed, and it soon became clear that mobile code (and content from untrustworthy sources) posed a real and serious threat to Internet security.

The rising trend of threats posed by untrustworthy mobile code has accelerated sharply in the last couple of years. This increase began with attacks embedded in maliciously crafted web pages that compromised popular browsers such as the Internet Explorer, and made it possible for attackers to execute arbitrary code on the computers running these browsers. Similar high-profile attacks included malicious code embedded in different types of image files, multimedia content, word-processing documents, and so on. The rising popularity of peer-to-peer networks such as KaZaA and instant-messaging software has only compounded the situation by providing other conduits for spreading cyber-attacks via malicious code.

Some previous efforts in mobile code security have focused on the dangers posed by mobile code, and developed techniques to prevent damage due to them. Often, these techniques posed undue restrictions on mobile code and prevented it from providing much useful functionality. In contrast, the MCC project is based on the observation that the vast majority of code and content downloaded over the Internet is *benign*. Thus, MCC's goal is to ensure that users can benefit from *benign mobile code* while minimizing the risks posed by malicious code. Its focus is on *practical* and *usable* solutions that are deployable in the near-term to secure the large base of existing software, while posing minimal burden on mobile code producers and consumers.

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

Current approaches to mobile-code security fall into two categories: *content inspection* and *behavior confinement*. Content-inspection techniques analyze potentially malicious content (whether it be data or code) in order to determine if it is indeed malicious. Their key benefit is *convenience*: little effort is required by mobile-code producers or consumers to make use of these tools. Their main drawback, which significantly limits their use, is the difficulty of detecting malicious code. Antivirus technologies [66] rely on detecting unique bit patterns ("signatures") that have previously been found only in malicious code. This approach, however, is ineffective for screening out malicious code that has not been seen before. Even worse, automated code-morphing tools are now available that can transform malicious code in ways that preserve their function, but alter the bit-patterns in them. These tools are being improved at a rapid rate, and will likely make signature-based detection ineffective in the future.

A second approach for content inspection is based on mathematical reasoning about the runtime behavior of code. Such reasoning can potentially infer all possible actions of mobile code, and can be used to discard code that may exhibit unsafe behavior. To be usable, such mathematical reasoning should be automated into a code-scanning software tool. Although major advances have been made in the area of automated reasoning and formal verification [13], the problem of verifying nontrivial properties of modern COTS (commercial, off-the-shelf) software continues to be intractable. The problem is further compounded by the fact that in the case of mobile code, such reasoning has to be done on binary code, which is much harder to analyze than source code.

Behavior confinement, otherwise known as *sandboxing*, is employed in the Java programming language [26] and several research tools. It is based on limiting the actions of mobile code so that it cannot cause any harm. Its benefits and drawbacks are complementary to that of content inspection. In particular, behavior confinement avoids the hard problem of reasoning about *all possible* behaviors of software. Instead, it inspects the actions that are actually performed during *a particular* execution of mobile code, and blocks any actions deemed risky. Its drawback is that by the time the risky behavior is observed, some damage may already be done. For instance, consider a file-compression program that replaces a file with a compressed version of its contents. By the time malicious behavior is detected, the program may have already erased the original file. Less worrisome, but still problematic, is the fact that the program may have created a number of intermediate files that need to be cleaned up manually after the risky behavior is detected and the program has terminated.

In summary, content-inspection approaches are *convenient* but do not provide a general solution for malicious software, whereas behavior-confinement approaches are *broadly applicable* but *difficult-to-use*. In contrast, the MCC approach aims to provide a convenient yet broadly applicable solution. A second drawback of existing approaches, shared by content inspection as well as behavior confinement, is the effort

needed to characterize “risky behavior,” which differs depending on the functionality of mobile code. For example, an instant-messaging program needs to communicate with remote hosts on the Internet. On the other hand, one would not expect an image viewer, operating on a local file, to access the network. Network communication represents risky behavior for such a program, as it may be used maliciously to send sensitive documents. The MCC project is the first one to provide practically useful tools to tackle this *security-policy specification* problem for untrusted code.

## 1.2 The MCC Approach

The MCC approach is aimed at combining the benefits of content inspection and behavior confinement approaches, while mitigating their drawbacks. In the vast majority of cases, MCC blocks the execution of unsafe mobile code even before it begins, thereby retaining the convenience of content-inspection approaches. Even when safety cannot be accurately determined before execution, but is observed subsequently at runtime, MCC eliminates the need for manual cleanup actions by incorporating automated recovery procedures. MCC avoids the computational intractability associated with content-inspection approaches by employing a judicious combination of runtime-monitoring techniques with automated analysis and reasoning.

One of the primary innovations in the MCC approach is the introduction of an intermediate level of abstraction, called a *model*, to bridge the semantic gap between low-level binary code and high-level security concerns of code consumers. Such a model captures the security-relevant aspects of code behavior, while abstracting away most other details that relate to the function of the code. Code consumers download mobile code together with its behavior model, and hence the term “model-carrying code.” These models are used by an automated verification procedure to determine if the code satisfies the consumer’s notion of safety. Since MCC models are hundreds (or thousands) of times smaller than programs, and considerably simpler, fully automated verification is feasible. Moreover, since models get generated by mobile code producers, the model generation process can benefit from the availability of source code, as well as the test suites used by the code producer.

In addition to tackling computational difficulties, the introduction of models in MCC provides another important benefit: it provides a way for code consumers and producers to collaborate to achieve security, while at the same time decoupling their concerns. This contrasts with previous approaches such as the proof-carrying code [41], which placed the responsibility entirely with the code producer, or approaches such as those used in Java [26], which placed the responsibility entirely on code consumers. In particular, our approach doesn’t burden producers with issues of “safety.” Indeed, the definition of safety can vary from one consumer to another, and is best left to the consumer. Similarly, consumers don’t need to predict the access needs of an application, which is best left to the code producer that wrote the code and understands its functionality as well as implementation. Producers simply encode this information about the access requirements of mobile code using a model. Armed with this model, a consumer can use automated verification tools to check if a piece of mobile code satisfies the specific safety concerns of interest to him/her.

The definition of safety in MCC is still based on *security policies*, but unlike previous approaches, MCC brings a considerable degree of automation to the policy-selection process. Indeed, suitable families of security policies can be preselected by security administrators so that naive users do not have to make complex security-related decisions. More sophisticated users are assisted in policy selection by the automated verification process, which can automatically suggest policy refinements that are consistent with the behavior of a given piece of mobile code.

The techniques developed in the MCC project are being incorporated into software tools that are placed at the entry points for mobile code and/or content downloaded over the Internet. Specifically, they are being incorporated into software installers, through which explicitly downloaded and installed software enters the system, and email/web browsers, through which implicitly downloaded code and content enter the system. These tools are being released as open-source software over the Internet. The list of currently available tools can be found on the project home page at <http://seclab.cs.sunysb.edu/mcc/>.

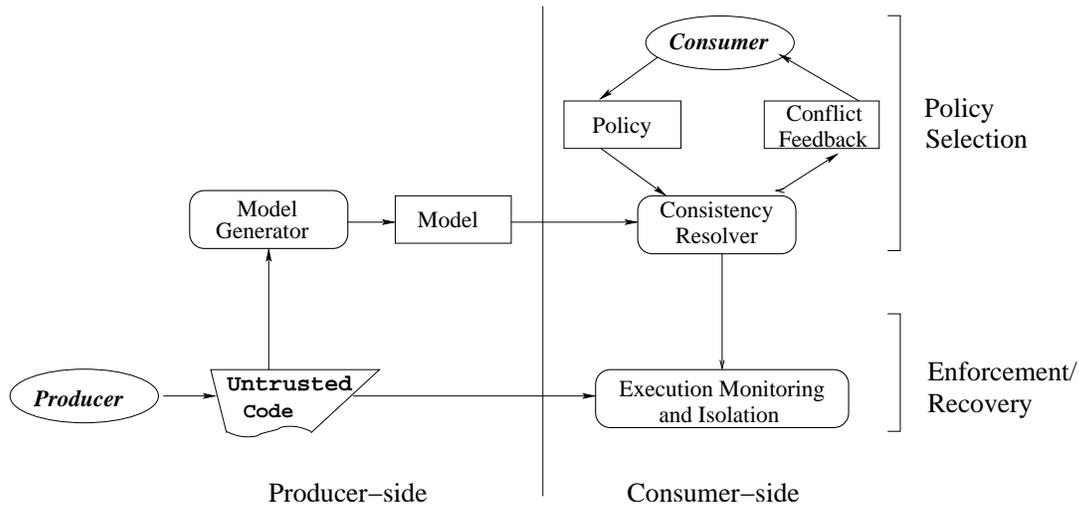


Figure 1: The Model-Carrying Code Framework

### 1.3 Paper Organization

The rest of this paper is organized as follows. Section 2 provides a high-level technical overview of the MCC approach. The research carried out in the MCC project falls into four main areas: *security policies*, *model extraction*, *consistency resolution*, and *enforcement*. These research efforts are summarized in Sections 3, 4, 5 and 6, with pointers to other papers that contain in-depth treatments of these efforts.

Another important outcome of MCC research has been the development of a new generation of techniques to counter *buffer overflow* attacks, which account for about 75% of security vulnerability advisories issued by leading organizations such as the US-CERT. A related development is that of automatic generation of signatures to block fast-spreading, Internet-wide attacks such as Code Red and Slammer. These research efforts are described in Section 7.

## 2 Overview of the MCC Approach

Execution of mobile/untrusted code has become an integral part of the everyday Internet experience. It appears in many forms, such as “active web pages” (e.g. pages with Java, Javascript, VBScript, or ActiveX content), content viewers and players (e.g., RealPlayer, FlashPlayer, Acrobat, and image viewers), games, P2P applications, and freeware/shareware or commercial applications that provide utility functions (e.g., photo album organizers, file compression and format conversion utilities, instant messengers, document search tools and related browser plug-ins). Moreover, complex content, such as images and documents, share some of the properties of code — in particular, maliciously crafted content can exploit vulnerabilities in the software that operates on this content to execute arbitrary code.

The MCC approach is designed so that users can enjoy all of the functions and benefits provided by benign mobile code and content, while adequately protecting themselves from the risks posed by malicious mobile code. The key idea in MCC (see Figure 1) is the introduction of program behavioral *models* that help bridge the semantic gap between (very low-level) binary code and high-level security concerns of consumers. These models successfully capture security-related properties of the code, but do not capture aspects of the code that pertain only to its functional correctness. The model is stated in terms of security-relevant operations made by the code and the operands of these operations.

While models can be created manually, doing so would be a time-consuming process. Code producers are unlikely to spend the additional effort needed to generate models, and requiring them to do so would hinder the widespread adoption of the MCC approach. To address this problem, we have developed techniques that can *automatically generate* the required models during software testing.

A code consumer receives both the model and the program from the producer. The consumer wants assurance that the code will satisfy a certain security policy. The use of a security behavior model enables us to decompose this assurance argument into two parts:

- *policy satisfaction*: check whether the model satisfies the policy, i.e., the behaviors captured by the

model are a subset of the behaviors allowed by the policy.

- *model safety*: check whether the model is a safe approximation of program behavior, i.e., the behaviors of the program are a subset of the behaviors of the model.

Together, policy satisfaction and model safety imply that the behavior exhibited by the program is a subset of the behavior permitted by the policy.

It should be noted that model safety is a necessary step whenever the code consumer does not trust the model provided by the code producer. In particular, the model provided by a producer may be incorrect either due to malice, or unintentional errors.

In principle, policy satisfaction as well as model safety can be established using automated verification techniques. In practice, however, we resort to runtime enforcement for ensuring model safety due to the difficulties in verifying properties of low-level (binary) code.

The policy selection component in Figure 1 is concerned with policy satisfaction, whereas the enforcement component is concerned with model safety. Policy selection uses automated verification; since models are much simpler than programs, complete automation of this verification step is possible. If the model is *not* consistent with the policy, the consistency resolver generates a compact and user-friendly summary of all consistency violations. The consumer can either discard the code at this point, or refine the policy in such a way that would permit execution of the code without posing undue security risks. Alternatively, a system administrator may preconfigure acceptable policies (and their refinements) for a given computer system, so that naive users don't have to make these decisions.

If the refined policy is consistent with the model, then the model and the code are forwarded to the enforcement module. Our current implementation of enforcement is based on runtime interception of security-sensitive operations made by untrusted code, specifically, system calls made by the program to access resources administered by the underlying operating system. If the enforcement component detects a deviation from the model, then the execution of the untrusted code is terminated. An alternative to model enforcement is to directly enforce the consumer's security policy.

As mentioned earlier, runtime monitoring and behavior enforcement suffers from the drawback that if a violation occurs at runtime, then the offending code will need to be terminated. Not only has the user wasted time with the malicious code, but in addition, she may have to manually clean up the "mess" left behind by the aborted application. This may entail recovering files that may have been deleted by the application, deleting files created by the application, and so on. To mitigate this problem, we have developed a new approach in the MCC project called *isolated execution* (see Section 6.1) that automates the recovery process, so that runtime aborts don't inconvenience users.

Although the primary focus of the MCC implementation has been on untrusted programs executing on the UNIX operating system, techniques from our approach could be easily adapted for different execution environments such as Java or Microsoft's Common Language Runtime (CLR). As a first step in this direction, we have performed some preliminary work in defining security policies in terms of security-relevant method calls in Java, and in implementing policy enforcement via bytecode rewriting [61].

### 3 Security Policies

We have developed a language for security policy specification called BMSL. As compared to previous research in security policy languages [19, 51, 26, 56], BMSL is more expressive and hence allows specification of a larger class of policies. Specifically, BMSL policies can restrict not only individual security-relevant operations, but also constrain the sequence in which they can be issued. Moreover, these policies can express complex constraints regarding the operands to these operations.

We have developed a compilation algorithm for BMSL policies that generates efficient policy-enforcement engines from high-level policy specifications. The performance of these enforcement engines is largely insensitive to the size or complexity of policies, thereby allowing users to focus on correctness of their specifications rather than their computational efficiency. Another key feature of BMSL is its mathematical foundation [59], including a precise semantics, and a formal proof that enforcement engines are faithful to this semantics. This factor, together with a new type system designed for the language, decreases the like-

likelihood of specification errors in security policies. These results are explained at depth in [59]. Application of this language to the related problem of intrusion detection (via policy enforcement) can be found in [60]. Finally, [61] describes the application of BMSL to the definition and enforcement of policies on Java programs.

We have developed a methodology for classifying applications into categories such that all applications in a given category can share the same policies. This method is well-integrated into the tools used for software installation and execution so that policy development and management can be largely automated. These results are promising in that they make it possible to apply MCC on a large scale, where it can manage the policies for a large number of diverse applications. This result also lays the foundation for applying MCC policy development and refinement framework more broadly, for instance, in the context of security-hardened operating systems such as SELinux [56] that require security policies to be developed for every application.

## 4 Model Extraction

We have developed efficient algorithms to generate security behavior models. These algorithms generate models that are hundreds of times smaller than the programs from which they are derived, a fact that has been instrumental in making the MCC approach practical for large programs. A high-level overview of our approach to model extraction can be found in [54], while an in-depth presentation can be found in [11]. The latter paper also discusses the application of these models to the problem of intrusion detection. It shows that MCC models are richer and more powerful as compared to models of program behavior used in previous intrusion detection techniques [21, 53, 20, 23, 22]. Whereas the previous approaches were focused on capturing control-flow behaviors, our approach can capture dataflows as well. This factor enables our models to detect a range of attacks that were outside the scope of previous approaches.

## 5 Consistency Resolution

We have developed a number of techniques for verifying that a security behavior model is consistent with the code consumer’s security policies. We use *model checking* [12, 47], a technique that explores different configurations (or states) the model may find itself in during execution. For example, the model may specify an execution where the contents of an arbitrary file  $F$  are read, followed by sending and receiving data over the network, followed by the writing of  $F$ . In this execution, we can identify a number of configurations: before  $F$  is read, after the read but before the network activity, the configurations due to the individual actions over the network, and the ones before and after  $F$  is written. Verification of models involves inspection of all the configurations (and sequences of configurations) that are possible in the model, and checking whether the policies are satisfied in these configurations (and configuration sequences).

Although there are many advanced model-checking techniques for the verification of systems of varying size and complexity [29, 17, 4, 28], MCC models have several characteristics that existing techniques cannot handle automatically. First, our models (as well as policies) may specify data values over unbounded domains (e.g., the above model treats arbitrary files). Model-checking techniques typically require that the set of possible states is finite, and verification techniques that can handle infinite state spaces typically require assistance from the user to complete the verification. Such restrictions limit the applicability of existing verification techniques to MCC.

In our research, we made a key observation that made these problems amenable to an elegant solution: that the behaviors of models do not change drastically when the data values change. In many cases, the models are in fact *data independent*, meaning that the control behavior of the model (i.e., its actions) are independent of the data values themselves. In [50] we described an automatic technique to verify properties of such models and corresponding policies. Our approach considers a set of configurations at a time, represented by a single *constraint*: every solution to the constraint represents a configuration in the set. For a large class of models, including all data-independent ones, the number of distinct constraints describing its set of reachable configurations is finite, thereby making complete verification possible. For a larger class of models, we developed simple approximation techniques where the set of configurations

represented by a constraint may under- or over-approximate the set of possible configurations, thereby making it possible to conservatively analyze the models for policy violations in finite time.

Another feature of mobile code models that is not treated by traditional model-checking techniques is the presence of function calls and recursion. These features also lead to a potentially infinite set of configurations. In [5] we describe an efficient technique for verifying such models. A key benefit of our approach is that it provides improved accuracy as compared to previous approaches.

Finally, a policy may be satisfied when only one instance of a mobile code is executed at any time, but may be violated when two instances are run simultaneously. Model-checking techniques can be used to verify properties only when the number of simultaneous executions is bounded and very small. In [6] we describe a technique that overcomes this drawback for certain classes of systems. In fact, our technique can verify properties of an *infinite* family of systems, i.e., an unbounded number of simultaneous executions. The key idea is to compute the condition each instance imposes on the remainder of the system in order to satisfy a policy. By observing the sequence of such conditions and evaluating how this sequence converges, we can estimate whether an arbitrary number of instances satisfies the policy.

*Policy refinement* comes into play when consistency resolution fails, i.e., the model violates some security policy. In this case, the policy may be more strict than is necessary and it becomes useful to provide feedback to the user regarding the failure so that the policy can be appropriately refined. Such feedback is given in terms of *counter-examples*: execution sequences of the model that demonstrate the policy violation. Previous verification techniques were geared towards providing one counter-example at a time. While this is a reasonable approach when verifying correctness properties, it is too cumbersome in the context of MCC. For instance, users may be told that the program violates a policy because it opens a file  $F_1$ , and if they accept this, then they are told that the policy is still violated because a file  $F_2$  is opened, and so on. In contrast, we have developed new approaches [7, 8] that can efficiently compute *all* counter examples, and thus present all violations in one shot to the user. This enables users to make more informed decisions on how to relax their policies, while also reducing the time spent in the refinement process.

## 6 Runtime Enforcement

We have developed techniques to enforce compatibility of mobile code with its model, or the consumer’s security policies. Our original enforcement approach was based entirely on intercepting operating system requests made by an application, and validating them against the model (or policy). This approach, however, has its drawbacks, one of which is related to recovery after attacks, and another of which is related to privacy policies. We subsequently addressed this shortcoming using the concept of *isolated execution* described below.

### 6.1 Isolated Execution

A purely enforcement-based approach causes significant inconvenience to a user *if* untrusted code is aborted at runtime. Specifically, we need to restore the system state so that it is *as if the aborted execution never took place*. It is cumbersome to manually identify the set of restoration actions to be performed. It is further complicated in realistic systems where a number of applications are running concurrently with the untrusted code, since we don’t want to undo the effects of these applications. To address this problem, we have developed a new approach, called *isolated execution*, that enables automatic recovery from runtime aborts. The main idea is to isolate the effects of mobile code, such as the files created or deleted, from the rest of the system. In effect, untrusted code operates on a “private copy” of the entire file system, allowing any and all operations on this copy. If this execution is to be aborted, we simply discard the private copy. If it is successful, then we provide techniques to merge the results of the execution into the host file system.

In addition to permitting automated and painless recovery from program aborts, the approach also expands the classes of security policies that can be supported within the MCC framework. In particular, policies are no longer required to be stated in terms of operations made by the mobile code, but can be given in terms of the system state resulting from these operations. Indeed, a policy can be based entirely

on the system state at the end of mobile-code execution.

Based on the concept of isolated execution, we have developed a stand-alone tool (independent of MCC) called *Alcatraz* for safe execution of potentially malicious applications [38]. An important advantage of *Alcatraz* is that most programs can run successfully within *Alcatraz*. This is in stark contrast with other behavior confinement approaches [26, 25, 1, 52], which cause most programs with substantive functionality to fail. Moreover, it permits users to manually inspect the changes made by mobile code before accepting the changes. This is helpful since our security policies can control only the operations performed on different files, but not the resulting changes to underlying file data.

Recently, we have shown that isolation can provide the basis for *safe execution environments* (SEE) within which users can “try out” operations that can potentially break their computer system. For instance, they can try out new software patches, new software packages, system configuration changes, vulnerability testing tools, and so on. If the result is satisfactory, they can continue on. Otherwise, they can recover back to the original system state at the touch of a button. Additional details on SEE can be found in [58].

## 6.2 Enforcing Privacy Policies Via Source-Code Rewriting.

It is well known that policies involving dissemination of private data cannot be enforced simply by examining the operations made by an application [51]. It is necessary to examine the flow of sensitive data within the application. Work to date in this area has focused primarily on compile-time techniques, called static analysis, to analyze all possible behaviors of a programs, and reject it if there is a potential for a leak [16, 2, 64, 40, 49]. Unfortunately, this approach hasn’t proved very practical for all but the simplest programs due to the fact that the analysis must simultaneously consider all possible execution paths and data values, and be conservative, reporting an information leak if there is even a single path along which a leak could occur. At runtime, the leaking path could well be infeasible due to conditions that cannot be determined at compile-time. As an example, consider a program that incorporates a bug report feedback to a vendor. If this crash report contains sensitive information (or data derived from sensitive information), a static analysis based approach would simply reject this program. In contrast, a runtime based approach can abort execution of the program when it is about to send a bug report. Based on this intuition, we have developed a source-code transformation to enforce information-flow policies [63]. We have formally established that the generality offered by our technique can be achieved *without* having to relax the privacy guarantees [24] provided by previous approaches based on compile-time analysis techniques.

Recently, we have refined and scaled this technique so that it can be applied to stop “injection attacks” that have become the biggest source of software vulnerabilities. In these attacks, an attacker “smuggles” illegal requests to a back-end server (e.g., a database server) past the validation checks performed by a front-end application by exploiting the software vulnerabilities in the latter. We have shown that this class of attacks can be accurately and automatically detected using our approach, which tracks information flow through a web application at very fine granularity. By combining fine-grained information flow with powerful security policies, we have shown that a wide range of attacks, including *buffer overflows*, *SQL injection*, *cross-site scripting*, *directory traversal*, *format-string attacks*, and *command injection* can be stopped. We have shown that such fine-grained information flow tracking can be achieved with very low overheads for security-critical servers, while experiencing moderate overheads for CPU-intensive programs. The technique has been applied successfully to stop attacks on large programs written in several languages, including C, PHP and Bash. Additional details about this technique can be found in [68].

## 7 Techniques for Preventing Memory Errors

About 75% of all security vulnerabilities reported in recent years by organizations such as US-CERT have been due to a specific software bug called *buffer overflow*, or more generally, a memory error. Memory errors cause the memory space of a program to be corrupted, often in ways that can be controlled by an attacker. By appropriately manipulating the input to a vulnerable program, attackers can corrupt its memory in such a manner that malicious code (or data) embedded in the input is copied into the program’s memory space and executed (or used). In spite of the publicity received by buffer overflows

and extensive efforts taken by software vendors to fix them, they continue to be discovered at an alarming rate. Moreover, even when defensive techniques are developed to address specific types of attacks that exploit buffer overflows [14, 18], newer attack types (that are sometimes even more versatile than the older attacks) have continued to emerge.

From the perspective of attacker, buffer overflows are very attractive since memory errors are pervasive in large-scale software, and are very hard to track down and eliminate. Thus, there is an endless supply of vulnerabilities that can be discovered and exploited. Moreover, when they are exploited, attackers are able to execute arbitrary code of their choice, thus giving them a great deal of flexibility and power. For these reasons, memory errors will likely remain the principal source of cyber attacks in the foreseeable future. Indeed, all of the Internet worms that have been reported in the past few years, including Code Red and Slammer, have exploited buffer overflows. Moreover, just within the last year, buffer-overflow vulnerabilities have been revealed in web browsers, image and document viewers, and multimedia players. These vulnerabilities allow attackers to infiltrate into computers that simply download a web page, image, document or play a song. These developments highlight the need for comprehensive solutions that can defeat all types of buffer-overflow attacks, whether they use previously known or novel strategies.

### 7.1 Program Transformation Techniques to Detect All Memory Errors

Although several techniques have been developed for detecting memory errors [27, 3, 45, 31, 48, 42, 30], they suffer from one or more of the following problems: inability to detect all memory errors, requiring extensive modifications to existing C programs, changing the memory management model of C to use garbage collection, and excessive performance overheads. As a result, these techniques aren't suitable to be applied to the vast base of existing software programs. We have therefore developed a new approach [69] for memory error detection that addresses these drawbacks. Our approach can detect *all* memory errors. It makes very conservative assumptions, thereby preserving compatibility with most legacy C code. It still imposes significant performance overheads, although it represents at least a four-fold improvement over previous techniques that preserve C's explicit memory management model [31, 48, 45]. Our ongoing research is focused on program optimization techniques to further reduce these overheads.

### 7.2 Randomization Based Techniques for Preventing Buffer Overflow Attacks

Although dynamic detection techniques such as those described above provide the most comprehensive protection from memory errors, they do impose significant performance overheads. Moreover, they can impact compatibility with precompiled libraries for which source code is unavailable. These factors have fueled the development of alternative solutions specialized for security, i.e., techniques for detecting memory errors that lead to attacks, rather than trying to capture all memory errors. Early work in this direction was focused on a specific type of attack called stack-smashing [14]. Subsequently, other types of attacks were discovered that necessitated the development of techniques specialized for those types of attacks [15]. Clearly, playing catch-up with attackers is not the best way to solve the problem, especially in the context of attacks with very serious consequences such as memory errors. Therefore, we developed the first solution, called *address obfuscation* [9], that offered broad protection against all common types of memory error attacks<sup>1</sup>.

Address obfuscation, alternatively called address-space randomization (ASR), operates by randomizing the locations of objects (such as program variables) within the memory space of an application. As a result, attackers can no longer predict the memory locations that need to be corrupted for a successful attack. We provided a comprehensive analysis of the strengths and vulnerabilities of randomization, and identified new randomization-targeted attacks that can be successful in some cases. Based on this analysis, we have recently developed a more comprehensive randomization approach [10] that provides probabilistic protection against all memory error exploits, whether they be known or novel. Our approach is implemented

---

<sup>1</sup>The same basic technique was also independently discovered by the PaX project [46], and aspects of this approach have since become an integral part of some recent distributions of the Linux operating system. Microsoft has also adopted the technique, incorporating address-space randomization into Windows Vista.

as a fully automatic source-to-source transformation which is compatible with legacy C code. The address-space randomizations take place at load-time or runtime, so the same copy of the binaries can be distributed to everyone; this ensures compatibility with today’s software distribution model. Experimental results demonstrate that our randomization techniques incur low performance overheads.

Recently, we worked with Global Infotek, a research firm based in Reston, Virginia, in a DARPA-sponsored project to implement address-space randomization in the context of Microsoft Windows.

### 7.3 Automated Signature Generation to Counter Automated Attacks

It is widely recognized that large-scale attacks, such as those launched by worms and zombie farms, pose a grave threat to our network-centric society. Existing approaches such as software patches are simply unable to cope with the volume and speed with which new vulnerabilities are being discovered. We have recently developed a new approach that can provide effective protection against a vast majority of these attacks. Our approach uses a forensic analysis of a victim server memory to correlate attacks to inputs received over the network, and *automatically* develop a signature that characterizes inputs that carry attacks. The signatures tend to capture characteristics of the underlying vulnerability (e.g., a message field being too long) rather than the characteristics of an attack, thereby making it effective against variants of attacks. Our approach introduces low overheads (under 10%), does not require access to source code of the protected server, and has successfully generated signatures for all of the attacks that we studied in our experiments, without producing false alarms. Signature generation is fast, taking a few milliseconds at most. This enables such filters to be deployed very quickly, thereby providing protection against fast-spreading worms. The techniques we have developed in this regard are described in detail in [35] and [36, 37]. As compared with most recent research in automated generation of attack signatures [55, 44, 33, 70, 65, 67], our approach can generate generalized signatures from a single attack instance. In experiments, we have shown that it produces no false positives, and is able to defeat variants of an attack that exploit the same underlying vulnerability. We have shown that these signatures can be deployed in the Snort intrusion detection system [57] to filter out attacks in the network.

## 8 Summary

In this project, we have developed a new approach called model-carrying code for safe use of untrusted code. Unlike previous approaches that were focused mainly on malicious code containment, MCC makes no judgment regarding the inherent risks associated with untrusted code. It simply provides the infrastructure and tools needed by code consumers to make risk versus reward decisions regarding untrusted code.

We have established the practicality of MCC by developing a software prototype. This prototype has been used on many moderate to large COTS applications in use today, and provides good runtime performance. More importantly, we have shown that MCC can be incorporated in a seamless fashion into tools that serve as conduits for untrusted code, including software installers, email handlers, and web browsers. We have developed an effective and convenient user interface for management and control of MCC. This management infrastructure can provide a highly simplified user-interface for naive users that can be tailored by security administrators. At the same time, these tools provide a much richer set of functionalities to security-savvy users.

In addition to mobile/untrusted code security, we developed solutions to several other important problems in computer security, including: security policy languages, policy verification and refinement, efficient policy enforcement, automatic extraction of security behavior models and their use in host-based intrusion detection, debugging memory errors, buffer-overflow defense, accurate detection of a wide range of attacks, automated generation of attack signatures, and so on. We have developed and publicly released several software tools that implement these solutions, including:

- *RPMSHield* for secure software installation,
- *Alcatraz* for safe execution of untrusted code,
- *Tracer*, which provides the infrastructure for system-call interception and runtime enforcement in MCC.
- *Address-space randomization tools* that produce address-space randomized programs from source code

- *Memory-Safe C* tool for transforming C programs to detect all memory errors at runtime

In addition, several other tools, including the complete MCC infrastructure and management tools, are in the final stages of development and are slated for release in the near future. The latest information related to this project, including publications and software releases, can be found at the project web page at <http://seclab.cs.sunysb.edu/mcc/>.

## References

- [1] Acharya, A. and Raje, M. 2000. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security Symposium*.
- [2] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):56–75, Jan. 1980.
- [3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Orlando, Florida, 20–24 June 1994. *SIGPLAN Notices* 29(6), June 1994.
- [4] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Computer Aided Verification CAV*, New York-Berlin-Heidelberg, July 2001.
- [5] S. Basu, K.N. Kumar, R.L. Pokorny, and C.R. Ramakrishnan. Resource constrained model checking for recursive programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.
- [6] Samik Basu and C. R. Ramakrishnan. Compositional analysis for verification of parameterized systems. In *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 315–330, Warsaw, Poland, April 2003. Springer.
- [7] Samik Basu, Diptikalyan Saha, Yow-Jian Lin, and Scott A. Smolka. Generation of all counter-examples for push-down systems. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, 2003.
- [8] Samik Basu, Diptikalyan Saha, and Scott A. Smolka. Localizing program errors for Cimple debugging. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, 2004.
- [9] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceeding of 12th USENIX Security Symposium*, 2003.
- [10] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceeding of 14th USENIX Security Symposium*, 2005.
- [11] Sandeep Bhatkar, Abhishek Chaturvedi and R. Sekar. Dataflow Anomaly Detection, *IEEE Security and Privacy*, 2006.
- [12] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, Yorktown Heights, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
- [13] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.
- [14] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.
- [15] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.
- [16] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

- [17] D. L. Dill. The Mur $\phi$  verification system. In *Computer Aided Verification*, 1996, pages 390–393.
- [18] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL <http://www.trl.ibm.com/projects/security/ssp/main.html>, June 2000.
- [19] David Evans and Andrew Tywman. Flexible policy directed code safety. In *IEEE Symposium on Security and Privacy*, 1999.
- [20] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.
- [21] S. Forrest, S. A. Hofmeyr, A. Somayaji, Intrusion Detection using Sequences of System Calls, *Journal of Computer Security* Vol. 6 (1998) pg 151-180.
- [22] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, August 2004.
- [23] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2004.
- [24] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.
- [25] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.
- [26] L Gong, M Mueller, H Prafullchandra, and R Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [27] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkeley, CA, USA, January 1992. USENIX.
- [28] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification CAV*, 2002.
- [29] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [30] Trevor Jim, Greg Morrisett, Dan Grossman, Micheal Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.
- [31] Robert W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In M. Kamkar and D. Byers, editors, *Third International Workshop on Automated Debugging*. Linkoping University Electronic Press, 1997.
- [32] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of 13th USENIX Security Symposium*, 2004.
- [33] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of 2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.
- [34] K. Krishna, W. Sun, P. Rana, T. Li, and R. Sekar. V-NetLab: A cost-effective platform to support course projects in computer security. *Colloquium for Information Systems Security Education*, 2005.
- [35] Z. Liang and R. Sekar. Automated, sub-second attack signature generation: A basis for building self-protecting servers. In *ACM conference on Computer and Communications Security (CCS)*, 2005.
- [36] Zhenkai Liang, R. Sekar, and Daniel C. DuVarney. Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self-healing systems. In *USENIX Annual Technical Conference*, 2005.
- [37] Zhenkai Liang and R. Sekar. Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models, In *Annual Computer Security Applications Conference (ACSAC)*, 2005.
- [38] Zhenkai Liang, VN Venkatakrishnan, and R. Sekar. Isolated program execution: An application

- transparent approach for executing untrusted programs. *Annual Computer Security Applications Conference*, 2003.
- [39] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.
- [40] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.
- [41] G. Necula and P. Lee, Proof-carrying code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1997.
- [42] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages (POPL '02)*, pages 128–139, Portland, OR, January 2002.
- [43] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of IEEE Symposium on Security and Privacy*, 2005.
- [44] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.
- [45] Harish G. Patil and Charles N. Fischer. Efficient run-time monitoring using shadow processing. In Mireille Ducasse, editor, *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, St. Malo,
- [46] PaX ASLR. Published on World-Wide Web at URL <http://pageexec.virtualave.net>.
- [47] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.
- [48] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium (NDSS)*, February 2004.
- [49] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), January 2003.
- [50] Beata Sarna-Starosta and C. R. Ramakrishnan. Constraint-based model checking of data-independent systems. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 2885 of *Lecture Notes in Computer Science*, pages 579–598, Singapore, November 2003. Springer.
- [51] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1), 2001.
- [52] Scott, K. and Davidson, J. 2002. Safe virtual execution using software dynamic translation. In *Proceedings of Annual Computer Security Applications Conference*.
- [53] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based approach for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.
- [54] R. Sekar, V.N. Venkatakrisnan, Samik Basu, Sandeep Bhatkar and Daniel C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, New York, October 2003. ACM Press.
- [55] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of 6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.
- [56] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *USENIX Annual Technical Conference (FREENIX Track)*, 2001.
- [57] Snort. Open source network intrusion detection system. <http://www.snort.org>.
- [58] Weiqing Sun, Zhenkai Liang, R. Sekar, and VN Venkatakrisnan. One-way isolation: An effective approach for realizing safe execution environments. *ISOC Network and Distributed Systems Symposium*, 2005.

- [59] Prem Uppuluri. *Intrusion Detection/Prevention Using Behavior Specifications*. PhD thesis, Stony Brook University, 2003.
- [60] Prem Uppuluri and R Sekar. Experiences with specification based intrusion detection. In *proceedings of the Recent Advances in Intrusion Detection conference*, October 2001.
- [61] V.N. Venkatakrishnan, Ram Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *New Security Paradigms Workshop (NSPW)*, 2002.
- [62] V. N. Venkatakrishnan, R. Sekar, T. Kamat, S.Tsipa and Z.Liang. An approach for secure software installation. In *Proceedings of USENIX System Administration Conference*, 2002.
- [63] V.N. Venkatakrishnan, Daniel C. DuVarney, Wei Xu and R. Sekar. A program transformation technique for enforcement of information flow properties. Technical report SECLAB-04-01, Secure Systems Lab, Department of Computer Science, Stony Brook University, Available at <http://seclab.cs.sunysb.edu/pubs.htm>, 2004.
- [64] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
- [65] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Proceeding of 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.
- [66] Anti-virus software, In Wikipedia, available at <http://en.wikipedia.org/wiki/Antivirus>
- [67] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *ACM CCS*, 2005.
- [68] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks, *USENIX Security Symposium*, 2006.
- [69] Wei Xu, Daniel C. Duvarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, California, November 2004.
- [70] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *USENIX Security*, 2005.