Security Vulnerabilities: From Analysis to Detection and Masking Techniques

SHUO CHEN, JUN XU, MEMBER, IEEE, ZBIGNIEW KALBARCZYK, MEMBER, IEEE, AND RAVISHANKAR K. IYER, FELLOW, IEEE

Invited Paper

This paper presents a study that uses extensive analysis of real security vulnerabilities to drive the development of: 1) runtime techniques for detection/masking of security attacks and 2) formal source code analysis methods to enable identification and removal of potential security vulnerabilities. A finite-state machine (FSM) approach is employed to decompose programs into multiple elementary activities, making it possible to extract simple predicates to be ensured for security. The FSM analysis pinpoints common characteristics among a broad range of security vulnerabilities: predictable memory layout, unprotected control data, and pointer taintedness. We propose memory layout randomization and control data randomization to mask the vulnerabilities at runtime. We also propose a static analysis approach to detect potential security vulnerabilities using the notion of pointer taintedness.

Keywords**—***Protection, randomization, security attack, vulnerability.*

I. Introduction

Security vulnerabilities are constantly discovered and exploited in computer systems. In-depth analysis of these vulnerabilities enables us to identify and extract their fundamental characteristics. This understanding can then guide the development of generic solutions, applicable to a broad range of vulnerabilities, making us less reliant on highly customized techniques aimed at protecting against specific types of vulnerabilities.

Manuscript received October 1, 2004; revised January 2, 2005. This work is supported in part by a grant from Motorola Inc. as part of Motorola Center for Communications, and in part by the MURI under Grant N00014-01-1-0576.

S. Chen, Z. Kalbarczyk, and R. K. Iyer are with Coordinated Science Laboratories, University of Illinois, Urbana-Champaign, IL 61801 USA (e-mail: shuochen@crhc.uiuc.edu; kalbar@crhc.uiuc.edu; iyer@crhc.uiuc.edu).

J. Xu was with the University of Illinois, Urbana-Champaign, IL 61801 USA. He is now with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695 USA (e-mail: junxu@csc.ncsu.edu).

Digital Object Identifier 10.1109/JPROC.2005.862473

This paper combines an analysis of data on security vulnerabilities (published in Bugtraq and CERT databases) with a focused examination of source code to model security vulnerabilities and associated attacks. The insights gained are used to devise two generic solutions: *measurement-driven mechanisms* for runtime detection and/or masking of security attacks that exploit residual vulnerabilities in the application code and *formal-reasoning-driven techniques* for automated (or semiautomated) identification and removal of vulnerabilities.

Our first step was to develop a finite-state machine (FSM) modeling methodology to depict and analyze a significant portion of the security vulnerabilities reported in CERT Advisories and Bugtraq databases. In our approach, each vulnerable program is decomposed into multiple elementary activities, each corresponding to a pair of predicates: the predicate that is *expected to be* implemented to ensure security and the predicate that is *actually* implemented in the program. A security vulnerability is thus represented as a predicate pair mismatch. The FSM methodology is exemplified by analyzing a wide spectrum of vulnerabilities, including stack buffer overflow, integer overflow, heap overflow, file race condition, and format-string vulnerabilities. Decomposing vulnerable programs into multiple elementary activities offers a formalism for analyzing the program implementation. The approach enables pinpointing of common characteristics among different categories of vulnerabilities.

FSM-based analysis of vulnerabilities indicates that most exploits (e.g., format-string, integer overflow, heap overflow and buffer overflow) succeed because of predictable program memory layout or unprotected control data. The vulnerabilities susceptible to these types of exploits are defined as unauthorized control information tampering (UCIT). Our survey of the CERT advisories indicates that UCIT vulnerabilities account for 65% of advisory entries. We use *memory layout randomization* (MLR) and *control data randomization* (CDR) to mask UCIT vulnerabilities.

Both techniques effectively break an attacker's ability to exploit the vulnerabilities, and both incur only small runtime overhead. These techniques do not require modification of application source code; they are implemented either by modifying the dynamic program loader (MLR) or by enhancing the existing C compiler (CDR). Both techniques are proven to be effective against real-world attacks.

Further data analysis provides evidence that a programming flaw, namely, *pointer taintedness*, is a common cause of vulnerability. A pointer is tainted when its value can be derived directly or indirectly from user input. Since pointers are internal to applications, they should be transparent to users. Thus, a taintable pointer is a potential security vulnerability. We have shown that the semantic of pointer taintedness can be formally defined using equational logic, which allows theorem-proving techniques to be applied on program source code to detect potential vulnerabilities. This technique has been applied to examine commonly used C-library functions to formally derive the security preconditions that must be met to ensure their vulnerability-free implementation.

Although the techniques we have implemented currently target programs written in C, they aim at uncovering residual security vulnerabilities and defeating attacks due to low-level memory errors. Therefore, the observations and techniques discussed in this paper are generically applicable to any program exposed to memory-related vulnerabilities.

To the best of our knowledge, this research represents a unique study, which exploits extensive analysis of real security vulnerabilities to drive the development of runtime detection and masking techniques. We believe that an in-depth understanding of the nature of vulnerabilities can significantly improve the effectiveness, efficiency, and general applicability of protection mechanisms. This paper extends our earlier work on analysis of vulnerability reports and MLR [30]. The extensions include: 1) analysis of complex vulnerabilities using the FSM model; 2) implementation and evaluation of mechanisms for control data (e.g., function pointers and return addresses) randomization; and 3) use of pointer taintedness analysis to uncover security vulnerabilities in applications.

II. Related Work

Security vulnerability analysis. Several studies have proposed classifications to abstract observed vulnerabilities into easy-to-understand classes. Representative examples include *protection analysis* [4], *RISOS* [3], Landwehr's taxonomy [2], Aslam's taxonomy [1], and the Bugtraq classification. Similarly, taxonomies for intrusions have been proposed. Examples include Lindqvist's intrusion classification [5] and the Microsoft *STRIDE* model [6]. In addition to providing taxonomies, [1] and [5] perform statistical analysis of actual vulnerability data, based on the proposed taxonomies. Several studies focus on modeling attacks and intrusions in order to evaluate various security metrics. Michael and Ghosh [9] employ an FSM model constructed using system call traces. By training the model using normal traces, the FSM is able to identify abnormal program behaviors and thus detect intrusions. In [8], an FSM-based technique to automatically construct attack graphs is described. The approach is applied in a networked environment consisting of several users, various services, and a number of hosts, and a symbolic model checker is used to formally verify the system's security. Recent studies have proposed stochastic models to evaluate security metrics quantitatively. Ortalo *et al.* [7] developed a Markov model to describe intruder behavior and evaluate system security in terms of mean effort to failure (METF). However, there is little work on modeling of discovered security vulnerabilities to capture how and why an implementation fails to achieve the desired level of security. Our research uses actual vulnerability reports and code inspection to derive simple predicates, which are then used to generate FSM models.

Static vulnerability avoidance techniques and runtime vulnerability masking techniques. Many static detection techniques have been developed based on the recognition of existing categories of security vulnerabilities. Techniques such as [10] and [11] can check security properties if vulnerability analysts are able to specify them as annotations in the code. Domain-specific techniques require less human effort, but each technique detects only a specific type of vulnerability. Static detection techniques, such as [13], have been proposed to detect buffer overflow vulnerabilities. Runtime techniques [14]–[18] either insert special checking code at compile time or instrument the runtime environment to dynamically detect possible security attacks. Most of these techniques aim at providing protection against specific types of attacks, e.g., StackGuard [16] against stack-smashing buffer overflow attacks, or FormatGuard [15] against format-string attacks.

Randomization-based techniques. Forrest [27] proposes protecting computer systems by introducing diversity into a system. The paper suggested several ways to do this, such as extending *gcc* to pad each stack frame by a random amount, thus thwarting stack-smashing attacks. Address Space Layout Randomization (ASLR) by PaX [28] implements an idea similar to that of memory layout memorization (MLR), but there are three important differences between the two.

1) MLR is implemented entirely in the user-space dynamic program loader, while ASLR requires changes to the Linux kernel. User-space implementation does not require reinstallation or even reboot of the operating system and hence is easier to use and deploy.
2) MLR overcomes the challenges of randomizing the global offset table (GOT) entries,[1] a frequent target of many attacks, while ASLR does not.
3) While our study shows that MLR has a small overhead only at process initialization time, the performance impact of ASLR is yet to be evaluated. The idea of encoding control data using a random key is implemented independently in CDR [26] and PointGuard [29].

[1]The GOT entry is a function pointer to a specific function. Usually, in position-independent code, e.g., shared libraries, all absolute symbols must be located in the GOT, leaving the code position-independent. A GOT lookup is performed to decide the callee's entry when a library function is called.

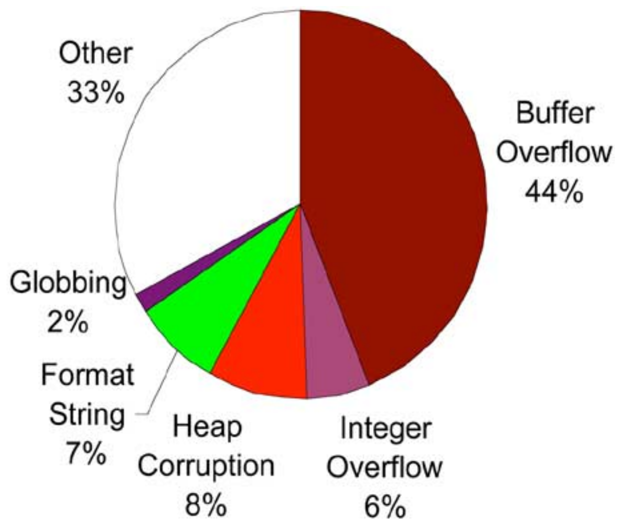


Fig. 1. Security vulnerabilities in CERT advisories (2000–2003).

III. Measurement-Driven Analysis of Security Vulnerabilities Using Modeling

A. *Categorization of Vulnerabilities*

Bugtraq database and CERT advisories are major data sources in our study. As of 30 November 2002, the Bugtraq database included 5925 reports on software-related vulnerabilities [19]. Details on the distribution of vulnerabilities across different categories can be found in [22]. This classification broadly defines the most common causes of security vulnerabilities, but it does not precisely indicate the programming flaws leading to these categories.

Obtaining the information on programming flaws requires an in-depth analysis of the vulnerability reports together with a close examination of the associated application code. CERT advisories are a suitable source on which to perform such a labor-intensive analysis, since CERT includes only the most significant vulnerabilities in the field. We have conducted analysis on the 107 CERT advisories from 2000 to 2003. Fig. 1 gives a breakdown of the leading programming flaws causing the vulnerabilities. Buffer overflow is a result of writing to an unchecked buffer; format-string vulnerabilities are caused by incorrect invocations of $printf$-link functions; integer overflow is due to interpreting extremely large signed integers as negatives; heap corruption is due to the corruption of heap structure or to freeing a buffer twice; globbing vulnerabilities result from incorrect invocation of LibC function *glob()*. These categories collectively account for 65% of the advisories.

Although the categories indicated in Fig. 1 correspond to different vulnerabilities, they share a common characteristic: all of them allow attackers to tamper with control data (e.g., function pointers or return addresses) to take control of the system, usually by forcing the system to execute the attacker's malicious code. We call these program flaws UCIT vulnerabilities. Detecting and masking UCIT vulnerabilities require identification and understanding of their fundamental characteristics. Toward this end, an FSM model is constructed for each vulnerability category. The models enable extracting the underlying system and application deficiencies leading to UCIT vulnerabilities, which are: 1) predictability of memory layout; 2) unprotected control data; and 3) the possibility of pointer taintedness. Sections IV and V introduce techniques to defeat UCIT vulnerabilities based on the extracted properties. In the rest of this section, we present the FSM-based approach for analyzing a security vulnerability.

B. *Security Vulnerability Characteristics*

In-depth analysis of the vulnerability reports in the Bugtraq database and CERT advisories and of the associated application source codes leads to the following observations.

- *Exploitation of a security vulnerability must pass through multiple elementary activities, at any one of which the exploit can be foiled*. We illustrate this observation using data from three signed integer overflow vulnerabilities given in Table 1. Here the analysts have used three different activities as reference points to classify the same type of vulnerability into three categories. Thus, #3163 has been classified as an input validation error, #5493 as a boundary condition error, and so on. The existence of three categories for the signed integer overflow vulnerability suggests that the code executions of the corresponding applications contain at least three elementary activities: 1) get an input integer; 2) use the integer as the index to an array; and 3) execute code referred to by a function pointer or a return address. Analysis of other types of vulnerabilities, including stack buffer overflow, heap corruption, and format-string vulnerabilities leads to the same observation.
- *Exploiting a vulnerability involves multiple vulnerable operations on several objects*. Let us consider again example #3163, a *Sendmail debugging function signed integer overflow*. This vulnerability involves two operations: first, manipulate the input integer (the object of this operation), consisting of elementary activity 1 (get an input integer) and elementary activity 2 (use the integer as the index to an array); second, manipulate the function pointer (the object of this operation), consisting of elementary activity 3 (execute code referred to by a function pointer).
- *For each elementary activity, the vulnerability data analysis and corresponding code inspections allow us to define a predicate, which if violated, results in a security vulnerability*. For example, in vulnerability #3163, an integer index x is assumed to be in the range [0,100], but the implementation only checks to guarantee that $x \leq 100$, hence the problem (vulnerability): allowing x to be a negative index and underflow an array. The correct predicate to eliminate this vulnerability would be $0 \leq x \leq 100$.

C. *State Machine Approach to Vulnerability Analysis*

Based on the observations discussed in the previous section, an FSM characterization of the vulnerable operations is developed. The goal of this FSM is to reason whether the

Table 1
Examples of Ambiguity Among Vulnerability Categories

Vulnerability	Description	Elementary Activity	Assigned Category
#3163 *Sendmail Debugging Function Signed Integer Overflow**	A negative input integer accepted as an array index	Get an input integer	Input validation error
#5493 *FreeBSD System Call Signed Integer Buffer Overflow*	A negative value supplied for the argument allowing exceeding the boundary of an array	Use the integer as the index to an array	Boundary condition error
#3958 *rsync Signed Array Index Remote Code Execution*	A remotely supplied signed value used as an array index, allowing the corruption of a function pointer or a return address	Execute code referred by a function pointer or a return address	Access validation error

* Each vulnerability reported to Bugtraq is assigned a unique ID, e.g., the report of vulnerability #3163 can be accessed from http://www.securityfocus.com/bid/3163.

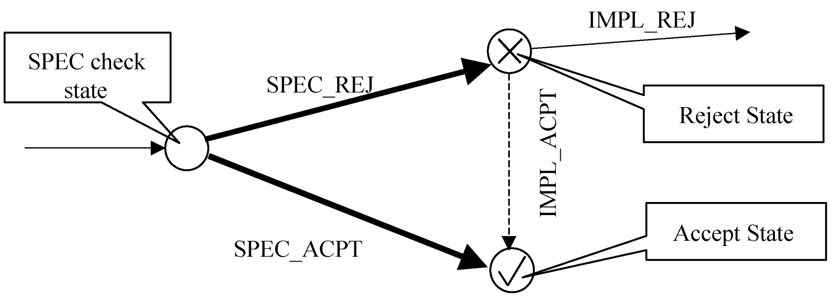


Fig. 2. Primitive FSM (pFSM).

implemented operation, or more precisely each elementary activity within the operation, satisfies the derived predicate. To this end, we take three steps.

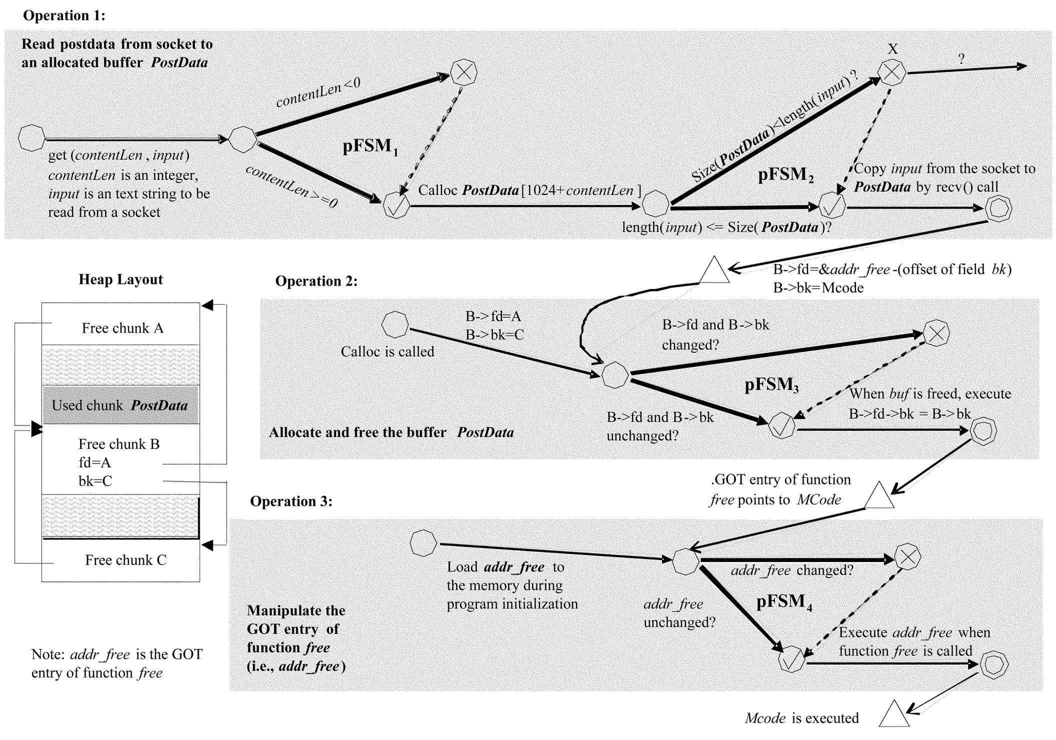
1) Represent each elementary activity as a primitive FSM (pFSM) expressing a predicate for accepting an input object. The predicate is first checked with respect to the specification and then with respect to the implementation.
2) Model an operation on an object as a series of pFSMs.
3) Cascade the operations to model the vulnerable implementation.

While our objective here is to reason that a vulnerability (violation of a derived predicate) is not present in the implementation, we shall see that the process of this reasoning can uncover a previously unknown vulnerability.

1) Primitive FSM (pFSM): The pFSM consists of four transitions and three states, as shown in Fig. 2. The transitions *SPEC_ACPT* and *SPEC_REJ* depict, respectively, the specification predicates of accepting and rejecting an object (e.g., request). The transition *IMPL_REJ* represents the condition under which the implementation rejects what should be rejected according to the specification. This transition depicts the expected or correct behavior, i.e., the implementation conforms to the specification. A dotted transition *IMPL_ACPT* represents the condition under which an object that should be rejected according to the specification is accepted in the implementation. This transition is a hidden path representing a vulnerability. Three states are identified: 1) the *SPEC check state*, where an object is checked against the specification; 2) the *reject state* ⊗—transition to this state indicates that the object is insecure according to the specification; and 3) the accept state ✓—transition to this state indicates that the object is considered secure.

2) FSM Model of a Heap Corruption Vulnerability: Null HTTPD is a multithreaded web server for Linux and Windows platforms. *Null HTTPD* 0.5 heap overflow is modeled as a series of four pFSMs, as shown in Fig. 3(a). pFSM_1 and pFSM_2 depict the buffer manipulation in the function *ReadPOSTData* (the function source code is shown in Fig. 3(b)), which allocates a buffer (*PostData*, source code Line 1) and copies a user-specified string from a socket (source code Line 4), which is marked as *input* in Fig. 3(a). One of the input parameters (*contentLen*) provides the length of *input*, which according to the specification should be a nonnegative integer. However, *Null HTTPD* allocates (by calling *calloc* in source code line 1) a buffer for *PostData* with size $1024 + contentLen$ without checking whether $contentLen$ is nonnegative. A buffer overflow occurs when the attacker provides a negative $contentLen$ (e.g., $contentLen = -800$) to make *PostData* a buffer with only 224 bytes. This results in buffer overflow (denoted by pFSM_1) because *Null HTTPD* always copies at least 1024 bytes arriving from the socket to *PostData* (source code Line 4).

As indicated earlier, each elementary activity offers an independent opportunity for checking. If the checks corresponding to the predicates depicted by pFSM_1 and pFSM_2 [in Fig. 3(a)] are not in place, the impact of this vulnerability is further analyzed using pFSM_3, which describes the operation manipulating the heap layout [as shown in the left of Fig. 3(a)]. The buffer *PostData* is allocated on the heap, followed by a free memory chunk (chunk B). Free chunks are organized as a double-linked-list by GNU-libc. The beginning few bytes of each free chunk are used as the forward link (fd) and the backward link (bk) of the double-linked list. In this case, since free chunks A, B, and C are in the list, $B \rightarrow fd = A$, $B \rightarrow bk = C$. The predicate defined


Operation 1:
Read postdata from socket to an allocated buffer *PostData*
get (*contentLen* , *input*) *contentLen* is an integer, *input* is an text string to be read from a socket
contentLen<0
contentLen>=0
pFSM$_1$
Calloc *PostData*[1024+*contentLen*]
Size(*PostData*)<length(*input*) ?
X
?
pFSM$_2$
Copy *input* from the socket to *PostData* by recv() call
length(*input*) <= Size(*PostData*)?
B->fd=&*addr_free* -(offset of field *bk*)
B->bk=Mcode
Heap Layout
Free chunk A
Used chunk *PostData*
Free chunk B
fd=A
bk=C
Free chunk C
Note: *addr_free* is the GOT entry of function *free*
Operation 2:
Calloc is called
B->fd=A
B->bk=C
B->fd and B->bk changed?
pFSM$_3$
When *buf* is freed, execute B->fd->bk = B->bk
B->fd and B->bk unchanged?
Allocate and free the buffer *PostData*
.GOT entry of function *free* points to *MCode*
Operation 3:
Load *addr_free* to the memory during program initialization
addr_free changed?
pFSM$_4$
addr_free unchanged?
Execute *addr_free* when function *free* is called
Manipulate the GOT entry of function *free* (i.e., *addr_free*)
Mcode is executed
(a)

```
PostData = calloc(contentLen
  +1024,sizeof(char)); x=0; rc=0;
pPostData= PostData;
do {
   rc=recv(sock, pPostData,
          1024, 0);
   if (rc==-1) {
       closeconnect(sid,1);
       return;
   }
   pPostData+=rc;
   x+=rc;
} while  ((rc==1024) &&
          (x<contentLen));
```

(b)

Fig. 3. (a) *NULL HTTPD* heap overflow vulnerabilities. (b) Source code, reading *input*.

in pFSM$_3$ provides a check so that $B \rightarrow fd$ and $B \rightarrow bk$ are not changed (i.e., pFSM$_3$ does not transit to the reject state) due to the overflow of the buffer *PostData* described in the pFSM$_1$ and pFSM$_2$. However, when *PostData* is freed, the actual implementation does not check the pointer $B \rightarrow fd$ and $B \rightarrow bk$, causing the transition from the reject state to the accept state (the hidden or dotted transition in pFSM$_3$), which allows the attacker to write an arbitrary value to an arbitrary memory location. Specifically, in this example, the attacker exploits this vulnerability and overwrites the GOT entry of the function *free()* so that it points to the location of malicious code *MCode*. The attack succeeds because when *PoseData* is freed, the assignment $B \rightarrow fd \rightarrow bk = B \rightarrow bk$ is executed, but the values of $B \rightarrow fd$ and $B \rightarrow bk$ are overwritten by the attacker.

pFSM$_4$ depicts the manipulation of the GOT entry of *free()* (i.e., *addr_free*). When HTTPD is started, *addr_free* is loaded to the memory. Finally, when *free()* is called, the value of *addr_free* is used as the function pointer to *free()*. Following the predicate depicted by pFSM$_4$, the system should check whether the value of *addr_free* is unchanged since it was loaded to the memory. If this is not the case (i.e., the *addr_free* has been tampered with), the program should not call to the location indicated by the corrupted *addr_free*. However, the corresponding implementation of *Sendmail* does not perform the check on the *addr_free* and accepts any value of it. As a result, the program again makes the dotted transition, and control jumps to the malicious code (*Mcode*) when *free()* is called.

In summary, this model consists of three operations. The first operation encompasses two activities, each described by an independent pFSM (pFSM$_1$ and pFSM$_2$). Operation 2 and operation 3 each consist of a single pFSM. Cascading these four pFSMs allows us to reason through all the vulnerable code.

3) Common Types of pFSMs: The FSM approach enables a detailed modeling/analysis of several types of security vulnerabilities: buffer overflow, race condition, signed integer, and format-string vulnerabilities [22]. Vulnerabilities including access validation errors, input validation errors, and failures to handle exceptional conditions can also be modeled.

The operations involved in each vulnerability can be modeled as a series of pFSMs, each corresponding to an elementary activity. Since pFSMs are critical to the analysis, it is meaningful to ask: Are there a few pFSMs that allow us to model the bulk if not all of the studied vulnerabilities? Our analysis shows that the pFSMs of studied vulnerabilities can be categorized into three types.

1) *Object type check*. This is a predicate to verify whether the input object is of the type that the operation is defined on. Examples of object type check errors include vulnerabilities reported in CERT Advisory CA-1994-06 and Bugtraq #3163, where the UNIX *rwall* service fails to check whether the file type is a terminal or a nonterminal file, and *Sendmail* fails to check whether the input represents an integer or a long integer.
2) *Content and attribute check*. This is a predicate to verify whether the content and the attributes of the object meet the security guarantee. For example, an HTTP daemon should verify that the request does not contain substring "../", a program should verify that format directives are not embedded in the strings to be printed, and the length

of a input string should be less than the size of the receiving buffer.

3) *Reference consistency check*. This is a predicate to verify whether the binding between an object and its reference is preserved from the time the object is checked to the time the operation is applied on the object. Examples include the return address referring to the parent function code, the function pointer referring to a function code, and a filename referring to a file. Several conditions may result in violating reference consistency, including stack overflow, signed integer overflow, heap overflow, format-string, and file race conditions.

Building FSM models for vulnerabilities makes it possible to identify simple common predicates that must be met to ensure vulnerability-free implementation. These predicates drive development of the detection and masking techniques discussed in Sections IV–VI.

IV. Runtime Masking of Security Vulnerabilities Using Randomization

Several *ad hoc* protection mechanisms have been proposed for preventing or detecting attacks that exploit subclasses of UCIT vulnerabilities; for example, StackGuard [16] against stack-smashing buffer overflow attacks and FormatGuard [15] against format-string attacks. However, the relatively new class of vulnerabilities, e.g., heap buffer overflow, integer overflow, and double-free, do not as yet have effective protection solutions. Given that there are many different types of vulnerabilities, a generic mechanism is clearly preferable. We propose MLR and CDR, two generic mechanisms for defending against a large class of UCIT vulnerabilities.

A. Common Characteristic of Attacks

Randomization defense mechanisms are based on a common characteristic of attacks that exploit UCIT security vulnerabilities. The *modus operandi* in all these attacks is the same: an intruder launches an attack by sending malicious messages/data to the system running the vulnerable application(s). Messages and data here broadly refer to various forms of external input that an application can receive, e.g., network messages, console input, command line options, or environment variables. Using the malicious messages/data, the attacker attempts to accomplish two things: 1) inject malicious code and 2) change existing control information (e.g., return addresses and function pointers) to point to the malicious code.

Let m be the address at which the malicious code is to be placed, and let p be the address of the target application's control information (return address or function pointer). The goal of the attacker is to overwrite the value at p so that the control information now points to the memory address m, where the malicious code/data is located. Table 2 shows a list of representative attacks, the vulnerabilities they exploit, the location (m) where the malicious code/data is to be placed, the control information at address p that needs to be changed, and the values that need to be determined for a successful attack. In a stack-smashing attack, the intruder only needs to determine one address, either m or p, while in others such as a *malloc*-based heap attack or integer overflow attack, the attacker needs to determine both m and p.

Table 2
Different Types of Attacks

Attack	Vulnerability	*m*	*p*	Value
stack-smashing	unchecked stack buffer	stack buffer	*ra*	*m* | *p*
return-to-library	unchecked stack buffer	stack buffer	*ra*	*m* & *p*[*]
malloc-heap	unchecked heap buffer	heap buffer	*fp* | *ra*	*m* & *p*
format-string	fmt string validation	format string	*fp* | *ra*	*m* & *p*
integer-overflow	unchecked signed integer	env. variables	*fp* | *ra*	*m* & *p*
double-free	released memory	heap buffer	*fp* | *ra*	*m* & *p*

fp - Function pointer; ra - Return address

[*]Here the attacker needs to change p to point to a function in a program's shared library and force that function to use the malicious data at m; therefore, the attacker needs to determine both m and the address of the library function.

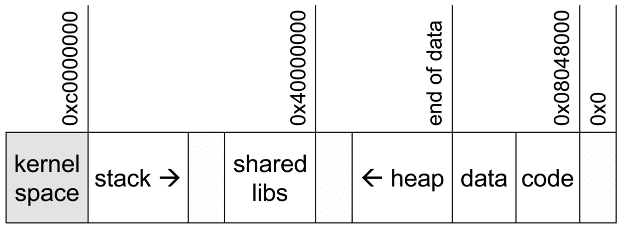


Fig. 4. Memory layout for Linux process (dotted regions are unmapped).

To launch a successful attack, an intruder must correctly determine the runtime values of m or p or both. This is usually achieved in the following way: 1) identify the versions of the application and operating system; 2) configure a pilot system to mimic the target system; and 3) craft and test the attack using the pilot system. A successful test run allows the attacker to obtain the values of p and/or m.

An attacker can determine these addresses correctly at a remote site even before the target application is running because process memory layouts on all modern operating systems are well known. Once an application is compiled, the locations of code, static data, shared libraries, and heap can be statically determined most of the time. Fig. 4 shows the process image on a default compilation of the Linux operating system. The starting addresses of the code, static data, bss, and heap are fixed once the application executable is generated. The locations of the user stack and shared libraries are determined by the kernel at process initialization time, but the kernel uses standard locations, as shown. Exploiting this predictable memory layout property, attackers can easily break into a system.

B. Memory Layer Randomization (MLR)

The idea behind MLR is to randomize the runtime locations of critical data elements in an application so that it is virtually impossible for an attacker to correctly determine the runtime locations of the necessary data elements shown in Table 2. In a stack-smashing attack, the attacker can easily find the value m (the address of vulnerable stack buffer where the malicious code is placed) in a regular system. In a system with MLR, however, the location of the stack is determined at runtime, and the attacker can no longer statically find its value. Hence, the attack is defeated. The remainder of this section discusses the design and implementation of MLR.

1) Randomizable Memory Regions: The goal of MLR is to transparently and randomly relocate memory regions at process initialization time so that existing applications can run without recompilation. We have identified two types of memory regions: *position-independent* and *position-dependent*. A position-independent region can be freely placed in the virtual address space at process-loading time without restrictions. Simple relocation of a position-dependent region could cause a chain of broken references from the program code.

Position-independent regions include *user stack*, *shared libraries*, and *user heap*. The location of the user stack is set up by the operating system kernel before an application begins to execute. The application addresses data on the stack through the stack pointer register (on Linux/IA-32, it is the *esp* register) and an offset value. The actual location of the stack is irrelevant as long as the stack pointer is correctly initialized. Shared libraries, also known as dynamically linked libraries, are compiled as position-independent code (PIC). The library functions are invoked by the program using base register plus offset, and they can be loaded anywhere in a program's address space as long as the base register is correct. User heap is managed by dynamic memory management functions, such as *malloc()* and *free()*. At runtime, *malloc()* determines the beginning of the heap via the *brk()* system call. A program accesses the heap using pointers returned by *malloc()*, hence the program does not make any assumptions about the runtime location of the heap.

Random relocation of a position-dependent region would invariably violate its correctness. Techniques for such relocation must be region-specific. We have implemented an algorithm in MLR to relocate the GOT, a position-dependent region that is a frequent target of attacks. The GOT is a table of function pointers that is used by all dynamically loaded libraries. Once a program is compiled, the GOT is fixed at a location inside the program's static data segment. The program code (i.e., the procedural linkage section) directly references the GOT. Our algorithm uses binary rewriting technique to make the changes consistent. Due to space limit, the GOT relocation algorithm is not presented here. Interested readers can find the technical details in [25]. We present the algorithm for position-independent region below.

2) Relocation of Position Independent Region: In modern, UNIX-based operating systems (Linux, FreeBSD, and Solaris), the memory layout of an application process is determined by the compile-time link editor, the operating system kernel, and the runtime dynamic program loader. The compile-time link editor determines the memory addresses of the program code, static initialized data, and static uninitialized data (BSS). The operating system kernel determines the starting address for the program heap, stack, and dynamic program loader. The dynamic program loader determines the memory location for shared libraries.

One way to randomly relocate the stack, heap, and shared libraries is to modify the *execve()* system call. When the kernel calculates and sets the base addresses for these position-independent regions, a random offset is added/subtracted to the base value. This approach, however, requires modification, recompilation, and reinstallation of the operating system. MLR implements the relocation algorithm in the dynamic program loader. When the operating system kernel hands over control to the dynamic program loader, the base addresses for the stack, heap, and loader have been set up. The modified loader changes these base execution environment settings before any other code is executed.

Before the kernel transfers control to the dynamic program loader, it places information about the execution environment on top of the user stack: the environment variable strings, an auxiliary information vector, the command-line option strings, a vector of pointers to environment variable strings, a vector of pointers to the command-line option strings, and the number of command-line arguments. To randomly relocate the user stack, the modified loader must preserve this information and the integrity of the pointers. The relocation is performed in the following steps: 1) create a new stack segment at a random location below the current stack using the *mmap* system call; 2) copy the content of the old stack to the newly allocated stack, adjusting the pointers in the auxiliary, environment, and command-line pointer vectors; 3) set the stack pointer register to the top of the new stack; and 4) free the old stack using the *munmap* system call. After these steps, the original stack allocated by the operating system kernel no longer exists; the application program will instead use the new stack randomly relocated by MLR.

MLR randomly relocates the user heap by growing the initial heap with a random amount of space using the *brk()* system call. To randomize the location of shared libraries, MLR creates a randomized memory mapping immediately following the dynamic program loader. When the loader maps the shared libraries used by the program into virtual memory, the mapping forces the libraries to be loaded at a location following the randomized region. Although random relocations for both heap and shared libraries require extra virtual memory address space, no extra physical memory space is needed. Operating systems usually allocate physical page frames only when the virtual memory page is accessed. Since the program is not aware of the added heap space and the memory mapping, no access should be made to these

Table 3
Evaluation Results Against Real Attacks

Program	*Attack Type*	*No MLR*	*MLR*
Traceroute	double free	local shell	crash
Sendmail	integer overflow	local shell	crash
Ghttpd	stack smashing	remote shell	crash
rpc.statd	format string	remote shell	crash
null httpd	heap overflow	remote shell	crash

memory regions, and hence no physical page frame should be needed.

3) Effectiveness and Performance Evaluation: The effectiveness of MLR was tested using publicly available vulnerable programs and attacks against them.[2] The selected applications are widely used in the open-source Linux operating system. Table 3 shows the vulnerable programs and attacks used in our experiments. Without MLR, the attacks succeed in obtaining a remote or local shell. With MLR in place, the attacks cause the target vulnerable program to crash, and therefore the intruder is stopped from causing further damage to the target system. The results also demonstrate the general applicability of MLR in defeating different types of attacks, including double-free, integer overflow and *malloc*-based heap overflow, for which no good solutions exist to date.

The performance overhead of MLR is measured using a large set of applications. Since MLR is implemented at process initialization, we measure the time between the application program's entry to the *execve()* system call and the run-time system's handing over of control to the program's entry point. The overhead numbers show that MLR only introduces minor program-startup overhead (2%–9%). The memory overhead of MLR is essentially the size of the GOT for the program. The additional memory space required is quite small, ranging from less than 200 bytes for small applications (e.g., *traceroute*) to around 3.5 KB for very large applications, e.g., the Netscape web browser (details can be found in [25]).

4) Possible Attacks Against MLR: Brute Force Attack. An attacker can try to guess the random offset used by MLR. The probability that an attacker guesses correctly is a function of the range of possible random offsets used by MLR. The larger the range, the lower the chance that it can be guessed. The current MLR implementation uses the range between 0 and 8192K. Each time an attacker makes a wrong guess, the application will be restarted, which results in a new random memory layout. The problem can be further alleviated by monitoring mechanisms: when a critical application keeps crashing, an alarm can be raised.

Information Disclosure Attack. Additional vulnerabilities can cause a program to leak runtime memory layout information. For example, information on a program's stack, such as return addresses and frame pointers, can be used to derive the initial starting address of a stack that has been randomized by MLR. The *FTP SITE EXEC bug* [20] is such a vulnerability for the specific version of wu-ftp 2.6.0. In this case, the remote FTP client can send a malicious input string to force the vulnerable FTP server to output contents of its runtime stack. When the threat exists, it is caused by an additional vulnerability that MLR is not designed to protect.

[2][Online]. Available: http://www.securityfocus.com

C. Control Data Randomization (CDR)

While MLR defeats attacks by making the address space unpredictable, CDR protects otherwise unprotected control data information. We define control data as program data that facilitate runtime program control flow transfer. There are two types of control data: *function pointers* (used for indirect function calls or indirect jumps), and *return addresses* (for resuming program flow after a function invocation). Control data are stored in a program's writable data regions (e.g., stack, heap, and data segment) and are modified dynamically as specified by program semantics. The fact that these data are in writable data regions is exploited by many security attacks.

In CDR, when a control datum is defined, it is encoded using a random key before being stored in its memory location. When the control datum is used, it is decoded (using the same random key) to its original form and stored in a temporary location for use. The in-memory value of the control datum, once defined, remains in encoded form throughout its lifetime. The CDR mechanism does not prevent an attacker from overwriting a control datum, but rather detects an incorrect value when the datum is used for control flow transfer. Without knowledge of the randomization key, the value chosen by the attacker will be decoded into a random address value. Control flow transfer to a random memory address will lead to a memory access violation, which will be detected by the underlying memory protection mechanism.

1) Function Pointer Randomization: The CDR approach requires that all function pointers be encoded at the time of definition and decoded at the time of use. This is not always necessary, since some cases of encoding and decoding can be eliminated without affecting the protection provided by CDR. We distinguish three types of function pointers definitions and uses.

1) *Function pointers defined by function symbol name*. The initial value of the function pointer is assigned from a function symbol name, e.g., $fptr = printf$. The function symbol name $printf$ will be translated to an immediate value by the compile time linker. CDR transforms $fptr = printf$ into $fptr = k \oplus printf$, where k is the randomization key, and $\oplus$ is the bit-wise exclusive-OR currently used for encoding.
2) *Function pointers used for control flow transfer*. When a function pointer is used for direct control flow transfer, as in $(*fptr)(\ldots)$, the compiler translates the C statement into machine-code-like call $*fptr$ or $jmp*fptr$. Since the value stored for $fptr$ in memory is always in its encoded form, CDR performs the control flow

transfer by transforming the machine code into call $(*fptr \oplus k)$, or $jmp\ (*fptr \oplus k)$.

3) *Function pointers defined or used for value propagation.* A function pointer can be defined using the value of another function pointer; similarly, the value of a function pointer can be used to define another function pointer, e.g., $fptr = fptr'$. Since $fptr'$ is already stored in its encoded form in memory, it is sufficient to copy the value of $fptr'$ to $fptr$ without encoding/decoding operations.

Function pointer initialization in declaration requires special handling. There are two basic types of function pointer variables: *automatic* and *global/static*. An automatic function pointer is declared inside the body of a function, and its storage is allocated on the runtime stack. Initialization of automatic pointers is done at runtime through compiler-generated code, alongside which CDR generates the encoding code. Storage for global or static function pointers is allocated in a program's static data segment, and their initial values are stored in the binary executable file at static linking time. Since no code is generated for the initialization and the encoding key is generated at runtime, CDR generates additional assembly code when parsing initialized global/static function pointers inside a variable declaration (which might be a scalar or an aggregate such as a structure, union, or array). The added code tags the variable and places the address of the variable in a special data section. CDR instruments the linker so that these special address sections from different object files are merged into a single read-only section. CDR also inserts a code section and schedules it to be executed at process initialization time. This piece of code goes through the list of addresses recorded in the special section and uses the dynamically generated random key to encode the initialized function pointers.

2) Randomizing Function Return Address: CDR implements the return address randomization in the register transfer level (RTL—the intermediate representation used by the compiler) code generation phase of the compiler by changing the function prologue/epilogue generation section of the compiler. The added RTL code loads the return address (top of the stack) into one of the registers, performs, and writes the encoded return address back on the top of the stack. The return address decoding is similar to the encoding procedure, except that it is implemented in the function epilogue generation section of the compiler.

3) The Randomization Key: The integrity of a CDR algorithm relies on the secrecy of the randomization key. There are two issues in protecting the secrecy of the key value: when to generate and how to store the key. Two approaches are proposed.

1) *Per-compilation key*—The value of the random key is generated at compile time and embedded directly in the application binary code as an immediate value. Embedding the key in the code instead of the data segment can reduce the number of data memory accesses. It also simplifies the encoding algorithms for initialized global/static function pointers, as the key is known at compile time and the encoded values can be directly stored in the executable by the compile time linker. The disadvantage is that the key is known and fixed throughout the life of the software.

2) *Per-process key*—The key regeneration mechanism is embedded in the executable and is run at process initialization time. This method increases the level of security with a slight decrease in runtime performance, as the CDR code needs to access memory each time to retrieve the key. CDR currently uses the per-process, key regeneration approach.

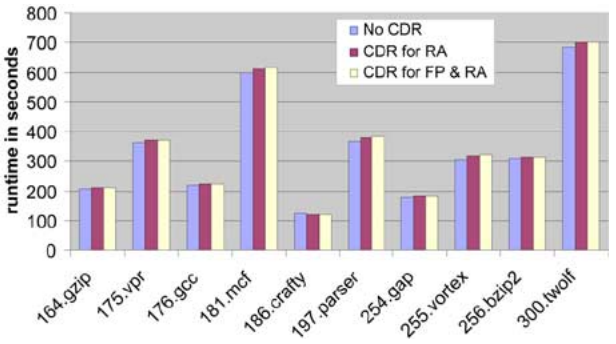


Fig. 5. Performance for SPEC-INT 2000.

For the per-process approach, the random key is regular memory data and needs to be stored in the address space of a program. The attacker can try to overwrite the key by exploiting the same vulnerability used to change the control data. We thwart this attack by allocating the key in a read-only data section. When generating the key, we change the memory protection attribute of the section to writable, write the key, and change the section back to read-only after the key is initialized.

4) Implementation and Evaluation: CDR is implemented by changing the C compiler, *gcc*, and the compile-time linking procedure currently used on Linux/IA-32 systems. The compiler is augmented to analyze the source code to determine the places where control data encoding/decoding is needed. Once the locations are determined, the compiler generates code to be executed at runtime for control data encoding and decoding. The compile-time linking procedure is instrumented with code to: 1) merge CDR data sections from the object files into a single read-only section and 2) generate the runtime key and encode global/static function pointers. The modified compiler has successfully compiled the GNU C library suite glibc-2.2.3.

To assess performance impact, we compiled the SPEC2000 integer benchmark using the modified compiler and linking the C library we compiled above. The results are shown in Fig. 5. Three sets of experiments were run: the base set with no CDR instrumentation, CDR instrumentation only for return address, and CDR instrumentation for both return address and function pointers. The overhead in the CDR for both return address and function pointers case is between 1% and 5%. Comparing this with the CDR for return address only, we find that the major portion of the incurred performance overhead is due to randomization of the return address. This is because in applications there are relatively fewer uses of function pointers than function invocations.

The effectiveness of CDR was tested using a set of attacks similar to the one used to evaluate the MLR approach. Without CDR instrumentation, the attacks succeeded in getting a remote/local shell prompt. CDR instrumentation foiled the attacks by crashing the target application.

V. Uncovering Security Vulnerabilities Using Pointer Taintedness Analysis

MLR and CDR techniques are runtime solutions that mask vulnerabilities by interfering with the attacker's ability to predict the memory layout of programs and overwrite control data. In this section, we examine the vulnerabilities from the program semantic perspective to develop techniques for automated uncovering of security vulnerabilities at the source code level.

A. *Pointer Taintedness*

We introduce the notion of pointer taintedness as a basis for reasoning about security vulnerabilities. This notion is based on the observation that a common cause of UCIT vulnerabilities is the fact that a pointer value (including return address) can be tainted, by which we mean derived directly or indirectly from user input. Since pointers are internal to applications, they should be transparent to users. By analyzing the application source code, the potential for pointers to be tainted can be determined and, hence, possible vulnerabilities can be identified. In [24], we illustrated how pointer taintedness can cause most types of security vulnerabilities. For example, format-string vulnerability is due to the taintedness of argument pointers of $printf$-like functions, heap corruption vulnerability is due to taintedness of the free-chunk, doubly linked list of heap management system, and stack buffer overflow is due to the taintedness of return addresses or frame pointers. Pointer taintedness allows attackers to arbitrarily specify the memory locations to read, write, or transfer control to, and thus usually indicates a pathological program behavior caused by security attacks.

Our analysis of vulnerability internals suggests that reasoning about pointer taintedness requires a thorough examination of program memory layout. Therefore, a memory model aware of the notion of taintedness is necessary in the code-analysis technique. The next section gives the workflow of pointer taintedness analysis; Section V-C defines the formal semantics of pointer taintedness using a memory model, and Section V-D shows a theorem-proving approach to reasoning about security vulnerabilities in library functions based on the defined semantics. Although the analysis tool is implemented for programs written in the C language, the underlying principle of pointer taintedness is applicable to reasoning about any program written in a type-unsafe language.

B. *Workflow of Pointer Taintedness Analysis*

Pointer taintedness analysis encompasses three basic steps, as illustrated in Fig. 6: 1) *parsing*—a C-source code of a given library function is automatically translated into corresponding formal representation expressed using a notation required by a selected theorem prover (the *Maude* theorem prover [21] is used in our study); 2) *theorem generation*—for each code line of the target function, where a pointer p is dereferenced, a theorem of the form " p *is not tainted*" is generated; and 3) *theorem proving*—a prover (the *Maude* in our case) is used to establish proofs of the generated theorems. If all theorems are proved to be valid unconditionally, the function has no possibility of pointer taintedness. Otherwise, the theorem prover identifies a set of preconditions that must hold to avoid pointer taintedness at runtime. These preconditions constitute security specifications of the target function.

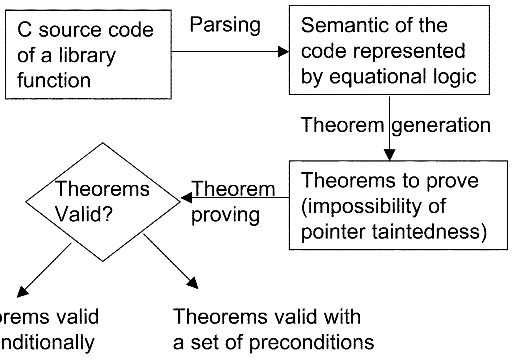

C source code of a library function
Parsing
Semantic of the code represented by equational logic
Theorem generation
Theorems to prove (impossibility of pointer taintedness)
Theorem proving
Theorems Valid?
Theorems valid unconditionally
Theorems valid with a set of preconditions

Fig. 6. Flowchart of pointer taintedness analysis.

C. *Formal Definition of Pointer Taintedness*

The formal semantic used to reason about pointer taintedness is based on the *programming semantic* of Goguen and Malcolm [12], which defines instructions, variables, and expressions. This semantic is extended with *memory model*, which defines memory locations and addresses as well as a set of operations to enable evaluating *content* and *taintedness* attributes for each memory location.

We define a *Store* to be a snapshot of the entire memory state at a point in the program's execution. The execution of a program instruction is defined as a function taking two arguments, a *Store* and an instruction, and producing another *Store*. Two operations, *fetch* and *location-taintedness*, are formally defined on a *Store*. The *fetch* operation $Ftch(S, I)$ gives the content of the address I in store S; the *location-taintedness* operation $LocT(S, I)$ returns a Boolean value indicating whether the content of the specified address is tainted. We also define two operations, *evaluation* and *expression-taintedness*, for expressions based on the *fetch* and *location-taintedness* operations. The *evaluation* operation $Eval(S, E)$ gives the result of evaluating the expression E under store S; the *expression-taintedness* operation $ExpT(S, E)$ indicates whether expression E contains any data from a tainted location. Table 4 gives the semantics of a subset of the supported statements. These are sufficient to analyze a wide variety of program constructs in the C language.

Formal specifications of statements other than the *mov* statement are fairly straightforward and similar to the definitions given in [12]. The semantic of *mov* is defined

Table 4
Semantics of Statements

Statement	Semantics
mov [E1] <- E2	Move the evaluation result of the expression E2 to the memory location addressed by the evaluation result of the expression E1.
if T then P1	If the condition T is true, execute P1.
else P2 fi	Otherwise, execute P2.
while T do P od	If the condition T is true, execute P; repeat until T is false.
goto L1	Go to the code line with the label L1.

by axioms that assert the facts: 1) after executing the *mov* statement, the content in the memory location addressed by E1 is equal to the evaluation result of E2 before executing the *mov* statement; 2) after executing the *mov* statement, the taintedness of the memory location addressed by E1 is identical to the expression taintedness E2 before executing the *mov* statement. A more detailed description of the semantic definitions of program statements is given in [23].

D. Extracting Security Specifications From Library Source Code

Having defined the memory model and the semantics of program statements, we can examine the possibility of pointer taintedness and extract security specifications or preconditions for a given library function.

Pointer taintedness analysis was applied to several functions, including *strcpy* (a string copying), *printf* (a printing function), *free* (a memory deallocation function), and socket reading functions of *Apache* HTTP daemon and NULL HTTP daemon. The results show that pointer taintedness analysis uncovers vulnerabilities such as buffer overflow, format-string vulnerability, and heap corruptions present in those functions. The details of the approach and an example of the analysis can be found in [23].

VI. Conclusion

This paper presents a measurement-driven approach that combines in-depth analysis of security vulnerabilities with development of runtime and static techniques to protect programs from a wide spectrum of vulnerabilities. The analysis starts with FSM modeling to depict vulnerabilities and associated attacks. The results drive development of runtime mechanisms to foil malicious attacks. Predictable memory layout, unprotected control data, and the possibility of pointer taintedness are identified as primary causes of a large class of system and application vulnerabilities.

MLR and CDR are proposed to provide runtime vulnerability masking. These techniques have shown low runtime overhead and high protection coverage. The pointer-taintedness analysis technique is also proposed to enable detection (and subsequent removal) of vulnerabilities using automated source code examinations. This technique has been applied to extract security preconditions for commonly used C library functions.

References

[1] T. Aslam, I. Krsul, and E. Spafford, "Use of a taxonomy of security faults," in *Proc. 19th NIST-NCSC Nat. Information Systems Security Conf.,* 1996, pp. 551–560.

[2] C. Landwehr, A. Bull, J. McDermott, and W. Choi, "A taxonomy of computer program security flaws, with examples," *ACM Comput. Surv.,* vol. 26, no. 3, pp. 211–254, Sep. 1994.

[3] R. P. Abbott, J. S. Chin, and J. E. Donnelley *et al.*, "Security analysis and enhancement of computer operating systems," Institute for Computer Sciences and Technology, National Bureau of Standards, NBSIR 76-1041, Apr. 1976.

[4] B. Bisbey, II and D. Hollingsworth, "Protection analysis project final report," USC/Information Sciences Institute, ISI/RR-78-13, DTIC AD A056816, May 1978.

[5] U. Lindqvist and E. Jonsson, "How to systematically classify computer security intrusions," in *Proc. 1997 IEEE Symp. Security and Privacy,* pp. 154–163.

[6] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, WA: Microsoft Press, 2001.

[7] R. Ortalo, Y. Deswarte, and M. Kaaniche, "Experimenting with quantitative evaluation tools for monitoring operational security," *IEEE Trans. Softw. Eng.,* vol. 25, no. 5, pp. 633–650, Sep. 1999.

[8] O. Sheyner, J. Haines, and S. Jha *et al.*, "Automated generation and analysis of attack graphs," in *Proc. IEEE Symp. Security and Privacy,* 2002, pp. 254–265.

[9] C. Michael and A. Ghosh, "Simple, state-based approaches to program-based anomaly detection," *ACM Trans. Inf. Syst. Secur.,* vol. 5, no. 3, pp. 203–237, Aug. 2002.

[10] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.,* vol. 19, no. 1, pp. 42–51, Jan./Feb. 2002.

[11] B. Chess, "Improving computer security using extended static checking," in *Proc. IEEE Symp. Security and Privacy,* 2002, pp. 160–172.

[12] J. A. Goguen and G. Malcolm, *Algebraic Semantics of Imperative Programs*. Cambridge, MA: MIT Press, 1996.

[13] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A first step toward automated detection of buffer overrun vulnerabilities," in *Proc. Network and Distributed System Security Symp. (NDSS 2000)* pp. 1–15.

[14] A. Baratloo, T. Tsai, and N. Singh, "Transparent run-time defense against stack-smashing attacks," in *Proc. USENIX Annu. Technical Conf.,* 2000, pp. 251–262.

[15] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman, "Formatguard: Automatic protection from printf format-string vulnerabilities," in *Proc. 10th USENIX Security Conf.,* 2001, pp. 191–199.

[16] C. Cowan *et al.*, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th USENIX Security Conf.,* 1998, pp. 63–78.

[17] Linux kernel patch from the Openwall Project. OpenWall Project [Online]. Available: http://www.openwall.com/linux/

[18] Stack Shield. Vendicator, 2000 [Online]. Available: http://www.angelfire.com/sk/stackshield/

[19] Bugtraq Vulnerability Archives. SecurityFocus [Online]. Available: http://www.securityfocus.com/bid/

[20] CERT, "Two input validation problems in FTPD," CERT Advisory ca-2000-13, 2000 [Online]. Available: http://www.cert.org/advisories/CA-2000-13.html

[21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, "The Maude 2.0 system," in *Rewriting Techniques and Applications*. Heidelberg, Germany: Springer-Verlag, 2003, vol. 2706, Lecture Notes in Computer Science, pp. 76–87.

[22] S. Chen, Z. Kalbarczyk, J. Xu, and R. Iyer, "A data-driven finite state machine model for analyzing security vulnerabilities," in *Proc. IEEE Int. Conf. Dependable Systems and Networks,* 2003, pp. 605–614.

[23] S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Formal reasoning of various categories of widely exploited security vulnerabilities using pointer taintedness semantics," in *Proc. 19th IFIP Int. Information Security Conf.,* 2004, pp. 83–100.

[24] S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Formal reasoning of various categories of widely exploited security vulnerabilities by pointer taintedness semantics (full version)," [Online]. Available: http://www.crhc.uiuc.edu/~shuochen/pointer_taintedness.pdf

[25] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *Proc. 22nd Symp. Reliable and Distributed Systems (SRDS),* 2003, pp. 260–269.

[26] J. Xu, "Intrusion prevention using control data randomization," in *Proc. IEEE Int. Conf. Dependable Systems and Networks (DSN), Suppl.,* 2003, pp. A25–A27.

[27] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proc. Workshop Hot Topics in Operating Systems* 1997, pp. 67–72.

[28] Documentation for the PaX Project. PaX Team. [Online]. Available: http://pax.grsecurity.net/docs/index.html

[29] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: protecting pointers from buffer overflow vulnerabilities," in *Proc. 12th USENIX Security Symp.,* 2003, pp. 91–104.

[30] R. K. Iyer, S. Chen, J. Xu, and Z. Kalbarczyk, "Security vulnerabilities—from data analysis to protection mechanisms," in *Proc. IEEE Workshop Object-Oriented Real-Time Dependable Systems,* 2003, pp. 331–338.

Shuo Chen received the B.S. degree from Peking University, China, in 1997 and the M.S. degree from Tsinghua University, China, in 2000. He is currently working toward his Ph.D. degree in computer science at the University of Illinois at Urbana-Champaign.

His research interests include security and fault tolerance, with an emphasis on systems research related to the analysis of real-world security vulnerabilities, security attacks, and the impacts of software/hardware faults on security.

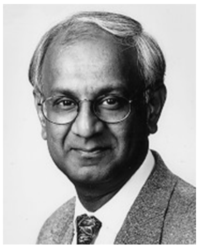
Jun Xu (Member, IEEE) received the B.S. degree in computer science from Peking University, China, in 1996, the M.S. degree in computer science from University of Pittsburgh, Pittsburgh, PA, in 1998, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 2003.

He is currently an Assistant Professor of Computer Science at North Carolina State University, Raleigh. His major research interests are computer system security and reliability. In particular, he is interested in operating system, compiler and architectural support for security and reliability, fault injection (aka fault insertion) based system security and reliability evaluation, distributed system for security and reliability, and measurement-based system security and reliability analysis.

Zbigniew T. Kalbarczyk (Member, IEEE) received the Ph.D. degree in computer science from the Technical University of Sofia, Bulgaria.

After receiving his doctorate, he worked as an assistant professor in the Laboratory for Dependable Computing, Chalmers University of Technology, Gothenburg, Sweden. He is currently a Principal Research Scientist at the Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign. His research interests are in the area of reliable and secure networked systems. Currently, he is a lead researcher on the project to explore and develop high availability and security infrastructure capable of managing redundant resources across interconnected nodes, to foil security threats, detect errors in both the user applications and the infrastructure components, and recover quickly from failures when they occur. His research also involves developing automated techniques for validation and benchmarking of dependable computing systems.

Dr. Kalbarczyk served as a program cochair of International Performance and Dependability Symposium (IPDS), a track of the Conference on Dependable Systems and Networks (DSN 2002), and is regularly invited to work on the program committees of major conferences on design of fault-tolerant systems.



Ravishankar K. Iyer (Fellow, IEEE) received the Ph.D. degree in electrical engineering from the University of Queensland, Australia.

He is the director of the Coordinated Science Laboratory (CSL), University of Illinois at Urbana-Champaign, where he is the George and Ann Fisher Distinguished Professor of Engineering. He holds appointments in the Department of Electrical and Computer Engineering and the Department of Computer Science, and he is codirector of the Center for Reliable and High-Performance Computing at CSL. He currently leads the Chameleon-ARMOR's project at Illinois, which is developing adaptive architectures for supporting a wide range of dependability and security requirements in heterogeneous networked environments. His research interests are in the area of reliable networked systems.

Prof. Iyer is a fellow of the Association for Computing Machinery and an Associate Fellow of the American Institute for Aeronautics and Astronautics (AIAA). He has received several awards, including the Humboldt Foundation Senior Distinguished Scientist Award for excellence in research and teaching, the AIAA Information Systems Award and Medal for "fundamental and pioneering contributions toward the design, evaluation and validation of dependable aerospace computing systems," and the IEEE Emanuel R. Piore Award "for fundamental contributions to measurement, evaluation and design of reliable computing systems."