7 Software Tamperproofing

To tamperproof a program is to ensure that it “executes as intended,” even in the presence of an adversary who tries to disrupt, monitor, or change the execution. Note that this is different from obfuscation, where the intent is to make it difficult for the attacker to *understand* the program. In practice, the boundary between tamperproofing and obfuscation is a blurry one: A program that is harder to understand because it’s been obfuscated ought also be more difficult to modify! For example, an attacker who can’t find the `decrypt()` function in a DRM media player because it’s been thoroughly obfuscated also won’t be able to modify it or even monitor it by setting a breakpoint on it.

The dynamic obfuscation algorithms in Chapter 6 (Dynamic Obfuscation), in particular, have often been used to prevent tampering. In this book, we take the view that a pure tamperproofing algorithm not only makes tampering difficult but is also able to *detect* when tampering has occurred and to *respond* to the attack by in some way punishing the user. In practice, tamperproofing is always combined with obfuscation:

1. If you both obfuscate and tamperproof your code, an attacker who, in spite of the tamperproofing, is able to extract the code still has to de-obfuscate it in order to understand it;
2. Code that you insert to test for tampering or effect a response to tampering must be obfuscated in order to prevent the attacker from easily discovering it.

Watermarking and tamperproofing are also related. In fact, if perfect tamperproofing were available, watermarking would be easy: Just watermark with any trivial algorithm, tamperproof, and by definition the attacker will not be able to destroy the mark! It's precisely because we don't have perfect tamperproofing that we need to worry about watermarking stealth: We have to assume that an attacker who can find a watermark will also be able to modify the program to destroy the mark.

The prototypical tamperproofing scenario is preventing an adversary from removing license-checking code from your program. So ideally, if the adversary is able to change your code on the left to the code on the right, the program would stop working for him:

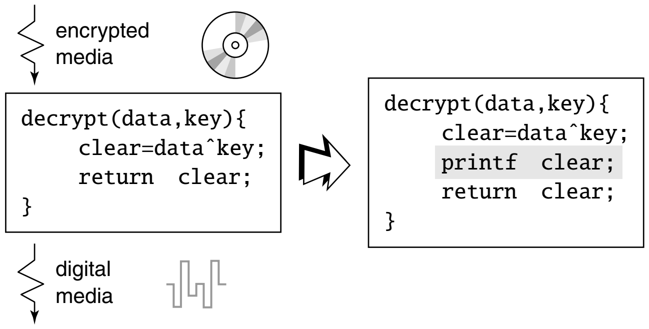
```
if (license_expired()) {
   printf("pay me!");
   abort();
}
```

```
if (false) {
   printf("pay me!");
   abort();
}
```

One way of thinking about this is to note that the program consists of two pieces of semantics: the piece that both you and the adversary want to maintain (because it constitutes the core functionality of the program) and the piece that *you* want to maintain but that the adversary wants to remove or alter (the license check). Clearly, maintaining the core semantics and removing the checking semantics are both important to the adversary. If not, it would be easy for him to destroy the tamperproofing code: Simply destroy the program in its entirety! The adversary may also want to *add* code to the program in order to make it perform a function that you have left out, such as the print function in an evaluation copy of a program.

There are many kinds of invariants that *you* may want to maintain but that your *user* may want to violate. For example, a free PDF reader might let you fill in a form and print it but not save it for later use. This is supposed to act as an incentive for buying the premium product with more functionality. However, from the user's point of view it's also an incentive for hacking the code to add this "missing feature." Other products don't allow you to print, games don't provide you with an infinite supply of ammunition, evaluation copies stop working after a certain period of time, VoIP clients charge you money to make phone calls, DRM media players and TV set-top boxes charge you to watch movies or listen to music, and so on. In all these scenarios, someone's revenue stream is depending on their program executing exactly as *they* intended it to, and an adversary's revenue stream (or debit stream) depends on modifying the semantics of the program to execute the way *they* want it to.

While not technically a case of modifying the program, observing its execution to steal algorithms, cryptographic keys, or any other proprietary code or data from the program secrets is sometimes also considered a form of tampering. For example, a common application of tamperproofing is to protect cleartext data or the cryptographic keys themselves in a digital rights management system. If an attacker can insert new code (shown here shaded) in the media player, he will be able to catch and save the decrypted media:



```
decrypt(data,key){
     clear=data^key;
     return  clear;
}
```

```
decrypt(data,key){
     clear=data^key;
     printf  clear;
     return  clear;
}
```

Observing the player under a debugger can achieve the same result without actually modifying the code:

```
> gdb player
(gdb) break decrypt.c:3
commands
   printf "%x\n",clear
   continue
end
```

Here, we've set a breakpoint on the `decrypt` function such that, whenever we enter it, the cleartext media gets printed and execution continues.

Technically, an attacker typically modifies the program with the intent to force it to choose a different execution path than the programmer intended. He can achieve this by:

1. Removing code from and/or inserting new code into the *executable file* prior to execution;
2. Removing code from and/or inserting new code into the *running program*;
3. Affecting the runtime behavior of the program through external agents such as emulators, debuggers, or a hostile operating system.

A protected program can try to detect that it's running under emulation, on a modified operating system, or inside a debugger, but this turns out to be hard to do reliably. For example, how can you detect that a user has turned back the system clock so as not to trigger your "license-expired" check? We will show a few popular techniques, but in practice, tamperproofing algorithms tend to focus on making sure that the program's static code and data haven't been changed, even if this certainly isn't enough to ensure it's running properly.

Conceptually, a tamperproofing system performs two tasks. First, it monitors the program (to see if its code or data have been modified), and the execution environment (to see if this is hostile in any way). Second, once it has determined that tampering has occurred or is likely to occur, a response mechanism takes over and retaliates in a suitable manner. This can range from making the program exit gracefully to punishing the user by destroying his home directory. In simple systems, detection and response are tightly integrated, such as `if (emulated()) abort()`, but this makes it easy for an attacker to work backwards from the point of response to the point of detection and modify the program to bypass the tamperproofing code. It's therefore important to separate the checking and the response functions as widely as possible, both spatially (they should be far away from each other in the executable) and temporally (they should be far away from each other in the execution trace).

Typical software tamperproofing approaches rely on self-checking code, self-modifying code, or adding a layer of interpretation. In general, these techniques are not suitable for type-safe distribution formats like Java bytecode—in Java it's simply not possible to stealthily examine your own code or generate and execute code on the fly. In this chapter, therefore, you will mostly see algorithms that protect binary executables.

Some algorithms are based on the idea of *splitting* the program into two pieces, allowing most of the program to run without performance penalty in the clear (open to user tampering) and the remainder to run slower, but highly protected. The protected part could be run on a remote trusted machine, in a tamper-resistant hardware module such as a smartcard, or in a separate thread whose code has been heavily obfuscated.

This chapter is organized as follows. In Section 7.1▸405, we give essential definitions. In Section 7.2▸412, we present algorithms based on the idea of *introspection*, i.e., tamperproofed programs that monitor their own code to detect modifications. In Section 7.3▸440, we discuss various kinds of response mechanisms. In Section 7.4▸444, we cover so-called *oblivious hashing* algorithms that examine the *state* of the program for signs of tampering. In Section 7.5▸453, we discuss *remote* *tamperproofing*, i.e., how we can determine that a program running on a remote machine has not been tampered with. In Section 7.6▶464, we summarize the chapter.

7.1 Definitions

An adversary's goal is to force your program P to perform some action it wasn't intended to, such as playing a media file without the proper key or executing even though a license has expired. The most obvious way to reach this goal is to modify P's executable file prior to execution. But this is not the *only* way. The adversary could corrupt any of the stages needed to load and execute P, and this could potentially force P to execute in an unanticipated way. For example, he could force a modified operating system to be loaded; he could modify any file on the file system, including the dynamic linker; he could replace the real dynamic libraries with his own; he could run P under emulation; or he could attach a debugger and modify P's code or data on the fly.

Your goal, on the other hand, is to thwart such attacks. In other words, you want to make sure that P's executable file itself is healthy (hasn't been modified) and that the environment in which it runs (hardware, operating system, and so on) isn't hostile in any way. More specifically, you want to ensure that P is running on unadulterated hardware and operating systems; that it is not running under emulation; that the right dynamic libraries have been loaded; that P's code itself hasn't been modified; and that no external entity such as a debugger is modifying P's registers, stack, heap, environment variables, or input data.

In the following definition, we make use of two predicates, $I_d(P, E)$ and $I_a(P, E)$, which respectively describe the integrity of the application (what the defender would like to maintain) and what constitutes a successful attack (what the attacker would like to accomplish):

Definition 7.1 (Tampering and Tamperproofing). Let $I_d(P, E)$ and $I_a(P, E)$ be predicates over a program P and the environment E in which it executes. P is successfully tamperproofed if, throughout the execution of P, $I_d(P, E)$ holds. It is successfully attacked if, at some point during the execution of P, $I_a(P, E) \wedge \text{not } I_d(P, E)$, holds and this is not detectable by P.

For example, in a cracking scenario, I_a could be, "P executes like a legally purchased version of Microsoft Word," and I_d could be, "The attacker has entered a legal license code, and neither the OS nor the code of P have been modified." In a DRM scenario, I_a could be, "P is able to print out the private key," and I_d could be, "The protected media cannot be played unless a valid user key has been entered $\wedge$ private keys remain private."

Conceptually, two functions, `CHECK` and `RESPOND`, are responsible for the tamperproofing. `CHECK` monitors the health of the system by testing a set of invariants and returning `true` if nothing suspicious is found. `RESPOND` queries `CHECK` to see if P is running as expected, and if it's not, issues a *tamper response*, such as terminating the program.

7.1.1 Checking for Tampering

`CHECK` can test any number of invariants, but these are the most common ones:

code checking: Check that `P`'s code hashes to a known value:

```
if (hash(P's code) != 0xca7ca115)
   return false;
```

result checking: Instead of checking that the code is correct, `CHECK` can test that the *result* of a computation is correct. For example, it is easy to check that a sorting routine hasn't been modified by testing that its output is correct:

```
quickSort(A,n);
for (i=0;i<(n-1);i++)
   if (A[i]>A[i+1])
      return false;
```

Checking the validity of a computed result is often computationally cheaper than performing the computation itself. For example, while sorting takes $O(n \log n)$ time, checking that the output of a sort routine is in sorted order can be done in almost linear time. Result checking was pioneered by Manuel Blum [43] and has been used in commercial packages such as LEDA [145].

environment checking: The hardest thing for a program to check is the validity of its execution environment. Typical checks include, "Am I being run under emulation?", "Is there a debugger attached to my process?", and, "Is the operating system at the proper patch level?" While it might be possible to ask the operating system these questions, it's hard to know whether the answers can be trusted or if we're being lied to! The actual methods used for environment checking are highly system-specific [166].

As an example, let's consider how a Linux process would detect that it's attached to a debugger. As it turns out, a process on Linux can be traced only once. This means that a simple way to check if you're being traced is to try to trace yourself:

```
#include <stdio.h>
#include <sys/ptrace.h>
int main() {
   if (ptrace(PTRACE_TRACEME))
      printf("I'm being traced!\n");
}
```

If the test fails, you can assume you've been attached to a debugger:

```
> gcc -g -o traced traced.c
> traced
> gdb traced
(gdb) run
I'm being traced!
```

Another popular way of detecting a debugging attack is to measure the time, absolute or wall clock, of a piece of code that should take much longer to execute in a debugger than when executed normally. In the light gray code in the example in Listing 7.1▸408, a divide-by-zero exception is thrown that the `gdb` debugger takes twice as long to handle as non-debugged code.

Here's the output when first run normally and then under a debugger:

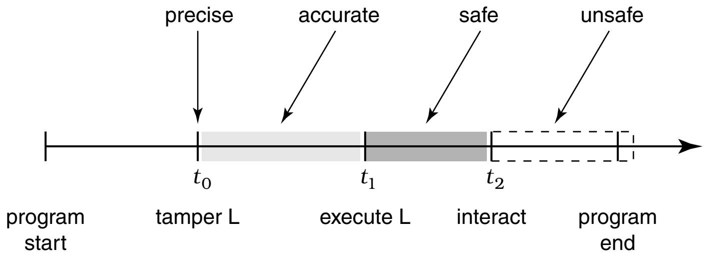
```
> gcc -o cycles cycles.c
> cycles
elapsed 31528: Not debugged!
> gdb cycles
(gdb) handle SIGFPE noprint nostop
(gdb) run
elapsed 79272: Debugged!
```

Listing 7.1 Code to detect if a Linux process is running under a debugger or not. The code in light gray throws a divide-by-zero exception. The x86 `rdtsc` instruction returns the current instruction count, and the `cpuid` instruction flushes the instruction pipeline to ensure proper timing results.

```
#include <stdio.h>
#include <stdint.h>
#include <signal.h>
#include <unistd.h>
#include <setjmp.h>

jmp_buf env;

void handler(int signal) {
   longjmp(env,1);
}

int main() {
   signal(SIGFPE, handler);
   uint32_t start,stop;
   int x = 0;
   if (setjmp(env) == 0) {
      asm volatile (
         "cpuid\n"
         "rdtsc\n" : "=a" (start)
      );
      x = x/x;
   } else {
      asm volatile (
         "cpuid\n"
         "rdtsc\n" : "=a" (stop)
      );
      uint32_t elapsed = stop - start;
      if (elapsed > 40000)
         printf("elapsed %i: Debugged!\n",elapsed);
      else
         printf("elapsed %i: Not debugged!\n",elapsed);
   }
}
```

The bound on the instruction count will have to be adjusted depending on the architecture and the debugger on which the program is run.

Depending on our needs, CHECK may be more or less *precise*, i.e., it could detect tampering close in time to when the tampering occurred, or not. A really imprecise detector is *unsafe*, in that it may fail to detect tampering until after the tampered program is able to cause permanent damage. Consider this timeline of the execution of a program P:



At some time t_0, a location L in P is modified by the attacker. The timeline doesn't show it, but this could either be prior to execution (if P's executable file is modified), during program start-up (if the loader is doing the modification), or during execution (if an attached debugger is doing the modification). At some later time t_1, the program reaches location L and the modified code gets executed. At time t_2, the first "interaction event" after t_0 occurs, i.e., this is the first time that the modified P has the chance to cause permanent damage by writing to a file, play protected content, send secret data back over the Internet, and so on.

A *precise* checker detects the attack "immediately" after the modification has taken place, perhaps within a window of a few instructions. With a precise detector, you're able to immediately punish a user who is experimenting by making random changes just to see "what happens." An *accurate* checker detects at some later time, but before t_1, which allows it to prevent the modified code from ever being executed, for example, by patching L. A *safe* detector waits until after the modified code has been executed, but before the first interaction event. At that point, the program is in an unintended and indeterminate state, but the detector knows this and can prevent any permanent damage by terminating the program. Finally, an *unsafe* detector waits until after the interaction event at t_2 or even until after program termination (postmortem detection) to detect tampering. At this point, permanent damage may have been inflicted, and the only remaining course of action may be to report the violation.

Definition 7.2 (Detector precision). Let t_0 be the time tampering occurs, t_1 the time the tampered code gets first executed, and t_2 the time of the first interaction event following the tampering. A *precise* detector detects tampering at time t_0, an *accurate* during (t_0, t_1), a *safe* detector during $[t_1, t_2)$, and an *unsafe* detector during $[t_2 \ldots)$.

Some tamperproofing systems will run `CHECK` only once on start-up. Such *static* systems catch modifications to the program's executable file, but not tampering that happens at runtime, for which a *dynamic* detector is necessary:

Definition 7.3 (Detector execution). A tamper-detector is *static* if detection happens only once at program load time and *dynamic* if detection is continuous throughout program execution.

7.1.2 Responding to Tampering

`RESPOND` executes a predetermined response to a detected attempt at tampering. Here are some possible responses:

terminate: Terminate the program. Some time should pass between detection and termination to prevent the attacker from easily finding the location of the detection code.

restore: Restore the program to its correct state by patching the tampered code and resetting any corrupted data structures.

degrade results: Deliberately return incorrect results. The results could deteriorate slowly over time to avoid alerting the attacker that he's been found out.

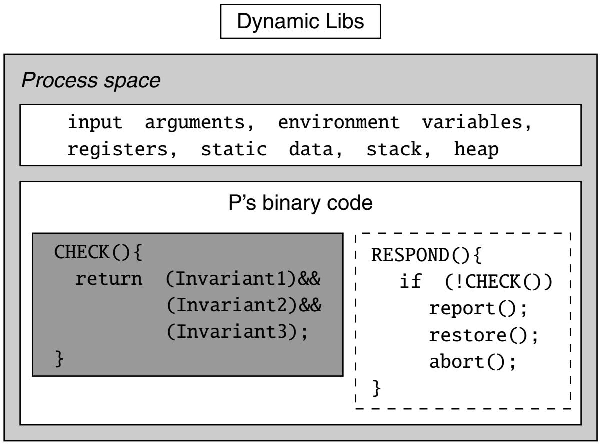
degrade performance: Degrade the performance of the program, for example, by running slower or consuming more memory.

report attack: Report the attack, for example, by "phoning home" if an Internet connection is available.

punish: Punish the attacker by destroying the program or objects in its environment. For example, if the *DisplayEater* program detects that you're trying to use a pirated serial number, it will delete your home directory [126]. More spectacularly, the computer itself could be destroyed by repeatedly flashing the bootloader flash memory.

7.1.3 System Design

There are many possible locations within a computing system for `CHECK` and `RESPOND`. Most obviously, you could integrate them directly in the binary code of your program:


Dynamic Libs
Process space
input arguments, environment variables, registers, static data, stack, heap
P's binary code
CHECK(){ return (Invariant1)&& (Invariant2)&& (Invariant3); }
RESPOND(){ if (!CHECK()) report(); restore(); abort(); }

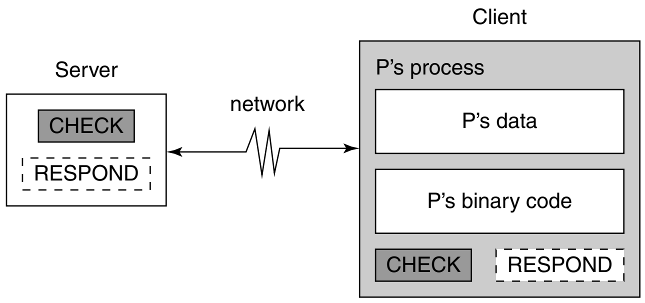
`CHECK` can check invariants over the binary code itself, over the static and dynamic data of the process, or over the environment (hardware and operating system) in which the program is running. We call this kind of organization a *self-checking* system. Note that in this design it's possible (and desirable!) for one of `CHECK`'s invariants to be a check on the integrity of its own code and `RESPOND`'s code. Without such a check, the adversary will start his attack by disabling `CHECK` or `RESPOND`, leaving himself free to further tamper with the program itself.

It's also possible for `CHECK` and `RESPOND` to run from within the hardware, the operating system, the dynamic loader, or a separate process.

Definition 7.4 (Integration). A tamperproofing system is *self-checking/self-responding* if the checker/responder is integrated within P itself. It is *external-checking/external-responding* if it runs in a different process from P but on the same computing system as P, and it is *remote-checking/remote-responding* if it is running on a different computing system than P.

A tamperproofing system can be self-checking but remote-responding, and so on.

Remotely checking the health of a program is an important subproblem known as *remote tamperproofing*. Here, you've put self-checkers in your program on the user's site:



In addition, checkers running on your own server site check for tampering by monitoring the communication pattern for anomalies. You'll read more about remote tamperproofing in Section 7.5▶453.

7.2 Introspection

We will devote the remainder of the chapter to describing tamperproofing algorithms from the literature. In this section, we will talk about two algorithms that use *introspection*. This essentially means the program is augmented to compute a hash over a code region to compare to an expected value. While this is a straightforward idea, as always, the devil is in the details.

Introspection algorithms insert code like this into the instruction stream:[1]

```
.........
start = start_address;
end   = end_address;
h = 0;
while (start < end) {
   h = h ⊕ *start;
   start++;
}
if (h != expected_value)
   abort();
 goto *h;
.........
```

1. We will call these functions *hash functions*. Some authors prefer the term *checksumming*, and others prefer *testing*. The term *guard* is sometimes used to collectively refer to the hash function and the response code.

The light gray part (the initialization) sets up the code region that is to be checked. The dark gray part (the loop) computes a hash value over the region using some operation $\oplus$. The light gray and dark gray parts together form the `CHECK` function, while the dashed and dotted parts show two possible `RESPOND`ers. The dashed responder simply terminates the program if the loop computes the wrong value (i.e., if the code region has been tampered with). The dotted responder instead uses the value computed by the loop to jump to the next program location. If the code region has been tampered with, the program will jump to the wrong location and eventually malfunction. This idea is used in the tamperproofing of the Skype client, which you'll learn more about in Section 7.2.4▶431. There are many possible variants of this idea. You could use the hash value as part of an arithmetic computation, for example, causing the program to eventually produce the wrong output, or you could use it in an address computation, eventually causing a segmentation fault.

Prototypical attacks on an introspective tamperproofing systems are based on pattern matching. They run in two steps:

1. Find the location of the checker and/or responder, either by
    (a) searching for suspicious patterns in the static code itself, or by
    (b) searching for suspicious patterns in the dynamic execution.
2. Disable the response code, either by
    (a) replacing the if-statement by `if (0) ...` , or by
    (b) pre-computing the hash value and substituting it into the response code.

Static pattern matching could, for example, target the initialization section. If the loop bounds are completely unobfuscated, it should be easy for the adversary to look for two code segment addresses followed by a test:

```
start = 0xbabebabe;
end   = 0xca75ca75;
while (start < end) {
```

A dynamic pattern matcher could look for data reads into the code segment, which will be unusual in typical programs. Depending on the nature of the tamper response, disabling means ensuring that the test on the hash value never succeeds or that the jump goes to the correct location, regardless of what value the loop actually computes:

```
if (0)
   abort();
goto *expected_value;
```

In this section, we'll show you two algorithms that make these types of attacks harder. The idea is to have multiple checkers and for checkers to check each other so that it's never enough to just disable one of them—you have to disable many or all of them. The TPCA algorithm builds up a network of checkers and responders so that checkers check each other and responders repair code that has been tampered with. Algorithm TPHMST refines this further by hiding the *expected_value* hash which, because of its randomness, is an easy pattern-matching target.

Another strategy to make pattern-matching attacks harder is to use a library of a large number of hash functions and to be able to obfuscate these thoroughly. We discuss this in Section 7.2.2▸418.

In Section 7.2.4▸431, we'll show you the heroic effort the Skype engineers went through to protect their clients and protocol from attack (in part, using introspection), and the equally heroic effort two security researchers went through (in part, using pattern matching) to crack the protection.

As it turns out, pattern matching is not the only possible attack against introspection algorithms. We will conclude the section by showing you algorithm REWOS, which is a generic and very clever attack that with one simple hack to the operating system disables *all* introspection-based algorithms! We'll also show you Algorithm TPGCK, which uses self-modifying code to counter this attack.

7.2.1 Algorithm *TPCA:* Checker Network

The TPCA [24,59] algorithm was invented by two Purdue University researchers, Hoi Chang and Mikhail Atallah. The algorithm was subsequently patented and with assistance from Purdue a start-up, Arxan, was spun off. The basic insight is that it's not enough for checkers to check just the code: they must check each other as well! If checkers are not checked, they are just too easy to remove. The algorithm builds up a network of code regions, where a region can be a block of user code, a checker, or a responder. Checkers compute a hash over one or more regions and compare it to the expected value. Responders in this algorithm are typically *repairers*, and if the checker has discovered that a region has been tampered with, a responder will replace the tampered region with a copy stored elsewhere. Multiple checkers can

Algorithm 7.1 Overview of algorithm TPCA. P is the program to be protected, in a form that makes control explicit, such as a control flow graph or a call graph. G is a directed *guard graph* describing the relationship between code regions, checkers, and responders.

TAMPERPROOF(P, G):

1. Let P's nodes be $n_0, n_1, \ldots$, representing code regions.
2. Let G's nodes be $n_0, n_1, \ldots$ (representing code regions), $c_0, c_1, \ldots$ (checkers), and $r_0, r_1, \ldots$ (responders). G has a edge $c_i \xrightarrow{c} n_j$ if c_i checks region n_j and an edge $r_i \xrightarrow{r} n_j$ if r_i repairs region n_j.
3. Insert the responders in P so that they dominate the region they check.
4. Insert the checkers in P so that at least one checker dominates every corresponding responder.
5. Connect checkers to responders by inserting variables, as necessary.

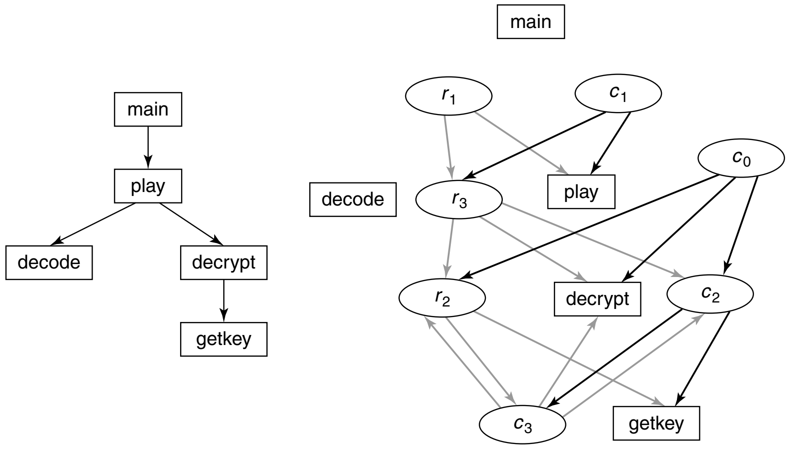
check the same region, and multiple responders can repair a tampered region. A nice consequence of this design is that you can achieve arbitrary levels of protection (at a concomitant increase in size and decrease in performance) by adding more checkers and responders and more complex relationships between them.

Have a look at the example in Listing 7.2▶416, a program that implements a simple DRM system. For simplicity, we've made code regions identical to functions, but in general this isn't necessary. A region could comprise multiple functions or parts of functions, and regions could be overlapping. We've inserted the checkers (in dark gray) and responders (light gray) so that they *dominate* the call to the function they check and repair. This way, you'll know that when a function gets executed it will be correct. For example, consider the `decrypt` function in Listing 7.2▶416. Before making the call to `getkey`, `decrypt` computes a hash of `getkey`'s code, and if it doesn't match (i.e., `getkey` has been tampered with), it repairs `getkey` by replacing it with a stored copy (`getkeyCOPY`).

Algorithm 7.1 gives an overview of the technique. To tamperproof a program, you need two graphs. P is the program's control flow graph (if you want to protect individual basic blocks) or a call graph (if you're content protecting one function at a time). A second graph G, the *guard graph*, shows the relationship between regions to be protected and the checkers and responders that check them. Corresponding to the example in Listing 7.2▶416, here is P (left) and G (right):

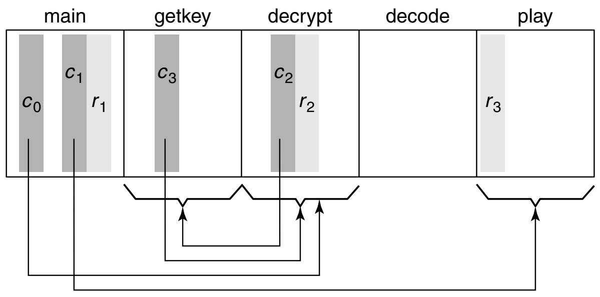
Listing 7.2 DRM player tamperproofed with algorithm TPCA.

```
#define getkeyHASH 0xce1d400a
#define getkeySIZE 14
uint32 getkeyCOPY[] = {0x83e58955,0x72b820ec,0xc7080486,...};
#define decryptHASH 0x3764e45c
#define decryptSIZE 16
uint32 decryptCOPY[] = {0x83e58955,0xaeb820ec,0xc7080486,...};
#define playHASH 0x4f4205a5
#define playSIZE 29
uint32 playCOPY[] = {0x83e58955,0xedb828ec,0xc7080486,...};

uint32 decryptVal;

int main (int argc, char *argv[]) {
   {uint32 playVal = hash((waddr_t)play,playSIZE);
   int user_key = 0xca7ca115;
   {decryptVal = hash((waddr_t)decrypt,decryptSIZE);
   int digital_media[] = {10,102};
   if (playVal != playHASH)
      {memcpy((waddr_t)play,playCOPY,playSIZE*sizeof(uint32));
   play(user_key,digital_media,2);
}

int getkey(in0 user_key) {
   {decryptVal = hash((waddr_t)decrypt,decryptSIZE);
   int player_key = 0xbabeca75;
   return user_key ^ player_key;
}

int decrypt(int user_key, int media) {
   {uint32 getkeyVal = hash((waddr_t)getkey,getkeySIZE);
   if (getkeyVal != getkeyHASH)
      {memcpy((waddr_t)getkey,getkeyCOPY,getkeySIZE*sizeof(uint32));
   int key = getkey(user_key);
   return media ^ key;
}

float decode (int digital) {
   return (float)digital;
}
void play(int user_key, int digital_media[], int len) {
   if (decryptVal != decryptHASH)
      {memcpy((waddr_t)decrypt,decryptCOPY,decryptSIZE*sizeof(uint32));
   int i;
   for(i=0;i<len;i++)
      printf("%f\n",decode(decrypt(user_key,digital_media[i])));
}
```



In G, edges $m \xrightarrow{c} n$ are black and represent m checking if n has been tampered with. Edges $n \xrightarrow{r} m$ (light gray) represent m responding to a crack of n. In this algorithm, responding usually means repairing, but this isn't necessary, of course.

Here's the corresponding code as it is laid out in memory:



Again, dark gray represents checkers, and light gray represents repairers.

Problem 7.1 Ideally, you want no checker to be unchecked itself, but TPCA doesn't allow such circularity among checkers. Chang and Atallah [59] write, Fermat style, "[W]e've solved the problem, but due to page limitation, we omit the discussion." Can you figure out what they had in mind?

7.2.2 Generating Hash Functions

To prevent pattern-matching and collusive attacks, it's important to be able to generate a large number of different-looking hash functions. If you've included more than one hash computation in your tamperproofed program, you even have to worry about *self-collusive* attacks. That is, an adversary who doesn't know what your hash functions look like can still scan through the program for pieces of similar-looking code: Any two pieces that look suspiciously similar and include a loop would warrant further study. In Chapter 10 (Software Similarity Analysis), you'll see *clone detectors* that are designed to locate exactly this kind of self-similarity.

In contrast to other applications that need hash functions, there's no need for ours to be "cryptographically secure," or even to compute a uniform distribution of values. Cryptographic hash functions such as SHA-1 and MD5 are large and slow, and have telltale static and dynamic signatures that an attacker could exploit in pattern-matching attacks. Once the attacker has located a hash function, disabling it isn't hard, and it matters little if it's SHA-1 or one of the trivial ones you'll see below. Therefore, the most important aspects of a hash function used in introspection algorithms are size (you'll include many of them in your program), speed (they may execute frequently), and above all, stealth (they must withstand pattern-matching attacks).

Here's a straightforward function that computes the exclusive-or over a region of words:

```
typedef unsigned int uint32;
typedef uint32* addr_t;

uint32 hash1 (addr_t addr,int words) {
   uint32 h = *addr;
   int i;

         for(i=1; i<words; i++) {
            addr++;
            h ^= *addr;
         }
         return h;
      }
```

To increase stealth in an actual implementation, you should inline the function.

Any tamperproofing algorithm based on introspection will need a library of different-looking hash functions. Here's one simple variant of hash1 above:

```
uint32 hash2 (addr_t start,addr_t end) {
   uint32 h = *start;
   while(1) {
      start++;
      if (start>=end) return h;
      h ^= *start;
   }
}
```

You have to be careful, however, to make sure that superficial syntactic changes at the source-code level actually lead to significantly different compiled code. It's entirely possible that a good optimizing compiler will generate very similar code for hash1 and hash2. For this reason, some tamperproofing implementations generate their hash functions directly at the assembly-code level.

You can add a parameter to step through the code region in more or less detail, which allows you to balance performance and accuracy:

```
int32 hash3 (addr_t start,addr_t end,int step) {
   uint32 h = *start;
   while(1) {
      start+=step;
      if (start>=end) return h;
      h ^= *start;
   }
}
```

There's, of course, no particular reason to scan forward, you can go backwards as well, and you can complicate the computation by adding (and then subtracting out) a random value (rnd):

```
uint32 hash4 (addr_t start,addr_t end,uint32 rnd) {
   addr_t t = (addr_t)((uint32)start + (uint32)end + rnd);
   uint32 h = 0;
   do {
      h += *((addr_t)(-(uint32)end-(uint32)rnd+(uint32)t));
      t++;
   } while (t < (addr_t)((uint32)end+(uint32)end+(uint32)rnd));
   return h;
}
```

The following function is used by algorithm TPHMST, where C is a small constant, odd multiplier:

```
uint32 hash5 (addr_t start, addr_t end, uint32 C) {
   uint32 h = 0;
   while (start < end) {
      h = C*(*start + h);
      start++;
   }
   return h;
}
```

To prevent pattern-matching attacks, Horne et al. [168] describe how to generate a large number of variants of `hash5`. The variants are generated by reordering basic blocks: inverting conditional branches; replacing multiplication instructions by combinations of shifts, adds, and address computations; permuting instructions within blocks; permuting register assignments; and replacing instructions with equivalents. This results in a total of 2,916,864 variants, each less than 50 bytes of x86 code in length.

Problem 7.2 Is generating three million different hash functions enough to confuse an attacker? Does knowing our hash function obfuscation algorithm help him, if the number of generated functions is large but finite? Can you combine the obfuscation ideas of `hash5` with those of `hash4` (adding redundant computations) to generate an infinite number of functions without losing too much performance?

This final hash function we're going to show you has been designed with some very special properties. You will see it used in Algorithm TPSLSPDK, in Section 7.5.4▸459, to verify that a client on a remote machine is actually running the correct code. An important part of this check is to measure the time it takes the client to compute a hash over the code. For this to work, it's essential that the hash function runs in very predictable time. A malicious user who can modify the function to run faster than you were expecting will be able to cheat you into believing they're running correct code when in fact they're not. To accomplish predictability, you must make sure that there are no ways for the user to optimize the code, for example, by parallelizing it or rearranging it into a faster instruction schedule. To make sure that the adversary can't rearrange the code, you must make it *strongly ordered*, i.e., each instruction must be data-dependent on previous ones. Specifically, these functions interleave adds and xors.

Also, you need to make sure that the adversary can't predict the order in which you're hashing the data. Therefore, rather than sweeping sequentially through the data, as in the previous functions, you should generate a random sequence of probes into the data segment. You can use a *T-function* [199] $x \leftarrow x + (x^2 \vee 5) \bmod 2^{32}$ to provide the necessary randomness. It's enough to probe $n \log n$ times (n is the length of the segment), since this makes you visit each word with high probability.

To make sure that the user doesn't try to save time by computing the function in advance, you should initialize the hash value to a random number `rnd` that you don't give him until it's time to compute the hash. Here, then, is the function:

```
#define ROTL(v) (((0xA0000000&v)>>31)|(v<<1))

// The segment should be a power of two long.
uint32 hash6 (addr_t start, addr_t end, uint32 rnd) {
   uint32 h = rnd;
   uint32 x = rnd;
   uint32 len = end-start;
   uint32 bits = ceil(log2(len));
   uint32 mask = 0xFFFFFFFF >> (32-bits);
   uint32 n = len*ceil(log2(len))/2;
   addr_t daddr = start;
   while (n>0) {
      x = x + (x*x | 5);
      daddr = start+(((uint32)daddr^x)&mask);
      h = h ^ *daddr;
```

```
         x = x + (x*x | 5);
         daddr = start+(((uint32)daddr^x)&mask);
         h = h + *daddr;
         h = ROTL(h);
         n- -;
      }
      return h;
   }
```

(The rotation makes the function immune to a particular attack [324].) To ensure predictable execution time, you must also consider cache effects. First, the hash function itself must be small enough to fit inside the instruction cache of the CPU, and second, the region you're hashing must fit into the data cache.

The Skype VoIP client inserts hundreds of hash functions that check each other and check the actual code. We'll show you the details of their protection scheme later, in Section 7.2.4▸431, but for now, here is the family of functions they're using:

```
uint32 hash7() {
   addr_t addr;
   addr = (addr_t)((uint32)addr^(uint32)addr);
   addr = (addr_t)((uint32)addr + 0x688E5C);
   uint32 hash = 0x320E83 ^ 0x1C4C4;
   int bound = hash + 0xFFCC5AFD;

   do {
      uint32 data = *((addr_t)((uint32)addr + 0x10));
      goto b1;
         asm volatile(".byte 0x19");
      b1:
      hash = hash ⊕ data;
      addr -= 1;
      bound- -;
   } while (bound!=0);
   goto b2;
      asm volatile(".byte 0x73");
   b2:
   goto b3;
      asm volatile(".word 0xC8528417,0xD8FBBD1,0xA36CFB2F");
      asm volatile(".word 0xE8D6E4B7,0xC0B8797A");
      asm volatile(".byte 0x61,0xBD");
   b3:
   hash-=0x4C49F346;
   return hash;
}
```

To prevent the address of the region we're checking from appearing literally in the code, the initialization section is obfuscated so that the address is instead computed. The routine is further obfuscated by inserting random data within the code, selecting a different operator $\oplus$ (add, sub, xor, and so on) for the hash computation, and stepping through the data in different directions and with different increments. As you'll see, `hash7` wasn't obfuscated sufficiently: The Skype protocol was broken by looking for a signature of the address computation.

7.2.3 Algorithm *TPHMST:* Hiding Hash Values

In real code, large literal integer constants are unusual. So a simple attack on a hash-based tamperproofing algorithm is to scan the program for code that appears to compare a computed hash against a random-looking expected value:

```
h = hash(start,end);
if (h != 0xca7babe5) abort();
```

If every copy of the program you're distributing is different, perhaps because you're employing fingerprinting, then you leave yourself open to easy collusive attacks: Since the code of every program is different, the hash values that your tamperproofing code computes must also be different and will reveal the location of the hash computation!

A simple fix is to add a copy of every region you're hashing to the program and then compare the hashes of the two regions:

```
h1 = hash(orig_start,orig_end);
h2 = hash(copy_start,copy_end);
if (h1 != h2) abort();
```

Algorithm 7.2 Overview of algorithm TPHMST. P is the program to be obfuscated, and n is the number of overlapping regions.

```
TAMPERPROOF($P$, $n$):
  1. Insert $n$ checkers of the form if (hash(start,end)) RESPOND() randomly
     throughout the program.
  2. Randomize the placement of basic blocks.
  3. Insert at least $n$ corrector slots $c_1, \dots, c_n$.
  4. Compute $n$ overlapping regions $I_1, \dots, I_n$, each $I_i$ associated with one
     corrector $c_i$.
  5. Associate each checker with a region $I_i$ and set $c_i$ so that $I_i$ hashes to zero.
```

An obvious disadvantage of this fix is that, in the worst case, your program has now doubled in size! Also, $f() = f()$ may not be all that common in real code, in which case the adversary might be able to guess what tricks you're up to.

Algorithm TPHMST [167,168] uses a very clever way of hiding the expected value literals. The idea is to construct the hash function so that unless the code has been hacked, the function always hashes to zero. This yields much more natural code:

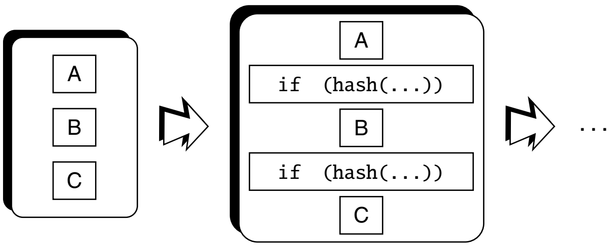
```
h = hash(start,end);
if (h) abort();
```

To accomplish this, TPHMST [167,168] uses the `hash5` hash function from Section 7.2.2▸418, which has the advantage of being *invertible*. This allows you to insert an empty slot (a 32-bit word, shown here in light gray) within the region you're protecting, and later give this slot a value that makes the region hash to zero:

```
start: 0xab01cd02
       0x11001100
slot:  0x????????
       0xca7ca7ca
end:   0xabcdefab

       h = hash(start,end);
       if (h) abort();
```

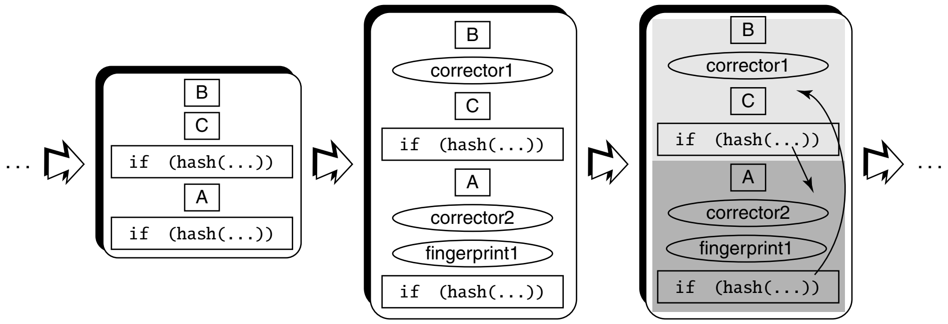
7.2.3.1 System Design Algorithm 7.2 is also interesting, because the paper [167] in which it was first presented, and the subsequent U.S. patent application [168], describe a complete and practical system for tamperproofing and fingerprinting. To build a functioning system, you have to solve many practical problems. For example, when during the translation and installation process do you insert fingerprints and tamperproofing code? Do you do it at the source-code level before compilation, at the binary code level at post-link time, or during installation on the end user's site? Each has different advantages and problems. The more work you do on the user's site during installation, the more he can learn about your method of protection, and the more you leave yourself open to attack. On the other hand, if all the work is done before distribution (that is, every distributed copy has already been individually fingerprinted and tamperproofed), then the distribution process itself becomes difficult: Are you only going to allow downloads and not sell shrink-wrapped CDs in normal retail outlets? If so, are you going to generate and store thousands of different copies of your program in anticipation of a surge of downloads? It's also important that the protection work doesn't interfere with the normal development cycle, including making debugging and quality assurance harder.

TPHMST spreads fingerprinting and tamperproofing work out over compile time, post-link time, *and* installation time. To illustrate the process, let's look at a program P with three basic blocks, `A`, `B`, and `C`. At the source-code level, you insert checkers of the form `if (hash(start,end)) RESPOND()`:



You want to make sure that any protection code added at the source-code level doesn't interfere with the normal development process. In this particular case, you can compile and link the program as usual, and you can set the `[start,end]` interval so that during development the response mechanism isn't triggered. Inserting the testers at the source-code level also has the advantage that the compiler will take care of register allocation for you.

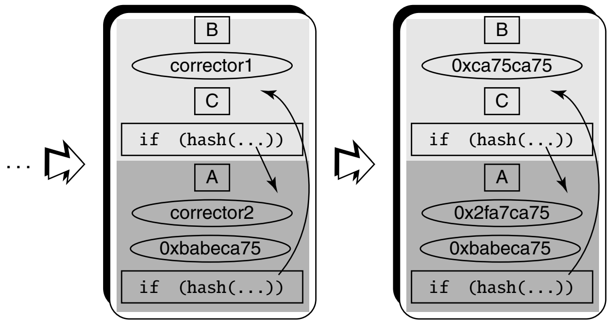
Next, you randomize the placement of the basic blocks and checkers. This is done on the binary executable. Randomization spreads the checkers evenly over the program and also helps with preventing collusive attacks. You then insert empty 32-bit slots for correctors and fingerprints. They will be filled in with actual values during installation. Finally, you create overlapping intervals and assign each checker to a region by filling in `start` and `end` of each `if (hash(start,end)) RESPOND()`:



Here, the first checker (in light gray) checks the dark gray region, and the second checker (in dark gray) checks the light gray region.

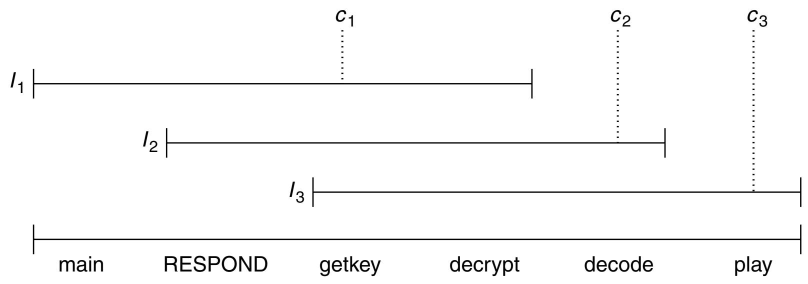
The fingerprinting and corrector slots can be added between basic blocks or after unconditional jump instructions. Finding suitable non-executed locations at the source-code level is difficult (a good compiler will typically optimize away any dead locations!), which is a good reason why this step is best done at the binary-code level.

The final stage occurs on the customer's site, during installation. In the form the program is in after being downloaded or bought on a CD, it's unusable. Since all the corrector slots are still empty, *every* checker will always trigger the response mechanism. Your first step during installation is to fill in the user's fingerprint value, possibly in multiple locations. Then you compute and fill in corrector values so that each checker hashes to zero:



Since the fingerprint slots are in the executable code, they are covered by the checkers (this is what makes them tamperproof!), but that also means that you cannot compute corrector values until the fingerprints have been filled in. As a result, if you want to fill in the fingerprints at installation time, you also must fill in the correctors at installation time.

7.2.3.2 Interval Construction In algorithm TPCA, you insert checkers so that they *dominate* the piece of code they're checking. This way, you can be sure that before control reaches a particular function or basic block, you've checked that it hasn't been tampered with. TPHMST instead randomly places large numbers of checkers all over the program but makes sure that every piece of code is covered by *multiple* checkers. To see how this works, have a look at Listings 7.3▶428 and 7.4▶429. The checkers are in light gray, the responders in dark gray, and the corrector slots are dashed. This is the same example program that we used to illustrate TPCA, but this time it is tamperproofed using algorithm TPHMST. We've defined three overlapping intervals, like this:



Each interval has a checker that tests that interval, and each interval I_i has a corrector c_i that you fill in to make sure that the checker hash function hashes to zero. You must compute the correctors in the order $c_1, c_2, c_3, \ldots$ to avoid circular dependencies. That is, first you set c_1 so that interval I_1 hashes to zero, after which I_2 only has one empty corrector slot, c_2. You next fill in c_2 so that I_2 hashes to zero, and so on.

In this example, the *overlap factor* is 2, meaning most bytes in the program are checked by at least two checkers. The authors of TPHMST suggest that an overlap factor of 6 gives the right trade-off between resilience and overhead.

In the examples in Listings 7.3▶428 and 7.4▶429, we're inserting correctors at the source-code level. This is complicated, because the corrector (not being executable code, just a 32-bit data word inserted in the instruction stream) has to be inserted in a dead spot. A smart compiler (one that is too smart for our needs!), however,

Listing 7.3 DRM player tamperproofed with algorithm TPHMST. Continues in Listing 7.4▶429.

```
#define interval1K         3
#define interval1START     (waddr_t)main
#define interval1END       (waddr_t)decode
#define interval1CORRECTOR "0x2e1e55ec"

#define interval2K         5
#define interval2START     (waddr_t)RESPOND
#define interval2END       (waddr_t)play
#define interval2CORRECTOR "0x2cdbf568"

#define interval3K         7
#define interval3START     (waddr_t)getkey
#define interval3END       (waddr_t)LAST_FUN
#define interval3CORRECTOR "0x28d32bb6"

//---------------- Begin interval 1 ---------------
uint32 main (uint32 argc, char *argv[]) {
   uint32 user_key = 0xca7ca115;
   uint32 digital_media[] = {10,102};
   play(user_key,digital_media,2);
}

//---------------- Begin interval 2 ---------------

void RESPOND(int i){
   printf("\n*** interval%i hacked!\n",i);
   abort();


//---------------- Begin interval 3 ---------------

uint32 getkey(uint32 user_key) {
   uint32 player_key = 0xbabeca75;
   if (hash5(interval1START,interval1END,interval1K)) {
      RESPOND(1);
      asm volatile (
         "      .align  4                       \n\t"
         "      .long " interval1CORRECTOR " \n\t"
      );
   }
   return user_key ^ player_key;
}
```

Listing 7.4 DRM player tamperproofed with algorithm TPHMST. (Continued from Listing 7.3 ▶428.)

```
uint32 decrypt(uint32 user_key, uint32 media) {
   uint32 key = getkey(user_key);
   return media ^ key;
}

//---------------- End interval 1 ---------------

float decode (uint32 digital) {
   if (hash5(interval2START,interval2END,interval2K)) {
      RESPOND(2);
      asm volatile (
         "       .align 4                       \t\n"
         "       .long " interval2CORRECTOR " \t\n"
      );
   }
   return (float)digital;
}

//---------------- End interval 2 ---------------

void play(uint32 user_key, uint32 digital_media[], uint32 len) {
   uint32 i;
   for(i=0;i  len;i++)
      printf("%f\n",decode(decrypt(user_key,digital_media[i])));
   asm volatile (
      "       jmp  L1                        \t\n"
      "       .align 4                       \t\n"
      "       .long " interval3CORRECTOR " \t\n"
      "L1:  \t\n"
   );
   if (hash5(interval3START,interval3END,interval3K))
      RESPOND(3);)
}

//---------------- End interval 3 ---------------
void LAST_FUN(){}
```

will remove dead code! In the example, therefore, we've either inserted the corrector right after a call to `RESPOND` (which will never return), and where our compiler's lack of interprocedural analysis will stop it from removing the slot, or by adding a jump around the slot.

7.2.3.3 Computing Corrector Slot Values Algorithm TPHMST uses the chained linear hash function `hash5` from Section 7.2.2▸418. This function has the advantage that you can hash an *incomplete* range (incomplete means that the corrector slot value is unknown) and then later solve for the corrector slot. Let's see how that's done.

Let $x = [x_1, x_2, \ldots, x_n]$ be the list of n 32-bit words that make up the region you want to protect. The region can be made up of code and static data, and will have one empty corrector slot. The region hashes to $h(x)$:

$$h(x) = \sum_{i=1}^{n} C^{n-i+1} x_i$$

C is a small, odd, constant multiplier. All computations are done modulo 2^{32}. Let's assume that one of the values in the region, say x_k, is the empty corrector slot. You want to fill this in so that $h(x) = 0$. Let z be the part of the hash value that excludes x_k:

$$z = \sum_{i \neq k}^{n} C^{n-i+1} x_i$$

This means you're looking for a value for x_k so that

$$C^{n-k+1} x_k + z = 0 \pmod{2^{32}}$$

This is a modular linear equation that you solve according to this theorem [94]:

Theorem 7.1: (Modular linear equation) The modular linear equation $ax \equiv b \pmod{n}$ is solvable if $d|b$, where $d = \gcd(a, n) = ax' + ny'$ is given by Euclid's extended algorithm. If $d|b$, there are d solutions:

$$\begin{aligned} x_0 &= x'(b/d) \bmod n \\ x_i &= x_0 + i(n/d) \quad \text{where} \quad i = 1, 2, \ldots, d-1 \end{aligned}$$

You get,

$$\begin{aligned} C^{n-k+1} x_k &= -z \pmod{2^{32}} \\ d &= \gcd(C^{n-k+1}, 2^{32}) = C^{n-k+1} x' + 2^{32} y' \\ x_0 &= x'(-z/d) \bmod 2^{32} \end{aligned}$$

Since C is odd, $d = 1$, and you get the solution

$$x_0 = -zx' \pmod{2^{32}}$$

To illustrate, given a region $x = [1, 2, x_3, 4]$ and a multiplier $C = 3$, let's find a value for x_3 so that $h(x) = 0$:

$$
\begin{aligned}
z &= \sum_{i \neq 3}^{4} C^{n-i+1} x_i = 1 \cdot 3^4 + 2 \cdot 3^3 + 4 \cdot 3^1 = 147 \\
3^2 x_3 &= -147 \pmod{2^{32}} \\
d &= \gcd(3^2, 2^{32}) = 1 = 3^2 \cdot 954437177 + 2^{32} \cdot (-2) \\
x_3 &= 954437177 \cdot (-147/1) \bmod 2^{32} = 1431655749
\end{aligned}
$$

Thus you get

$$h(x) = (1 \cdot 3^4 + 2 \cdot 3^3 + 1431655749 \cdot 3^2 + 4 \cdot 3^1) \bmod 2^{32} = 0$$

as expected.

Problem 7.3 Like TPCA, TPHMST doesn't allow circularity among checkers; some region will always be unchecked! The authors state [167] that it's possible to modify the interval construction by solving "the resulting system of linear equations," giving no further details. Work out the details!

7.2.4 The Skype Obfuscated Protocol

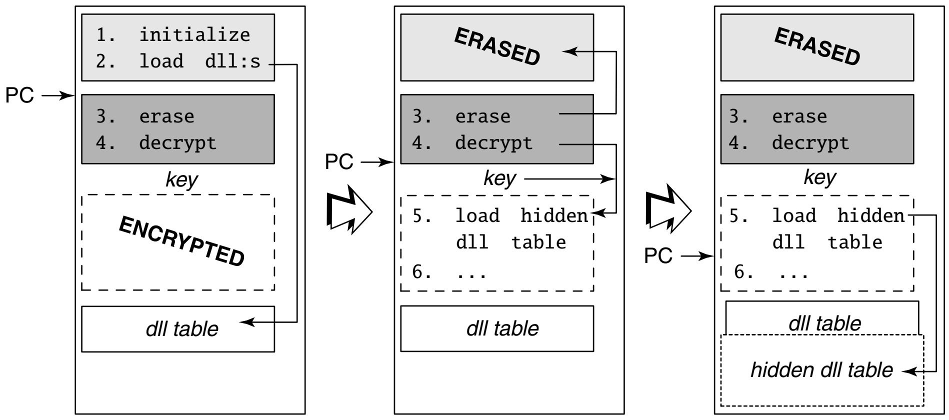
Skype is a Voice-over-IP service that operates on a peer-to-peer model. Users can make computer-to-computer voice calls for free but are charged for computer-to-phone and phone-to-computer calls. Skype was bought by eBay in September 2005 for $2.6 billion.

The Skype client is heavily tamperproofed and obfuscated. The protocol by which clients communicate with each other is proprietary and also obfuscated. The clients remained unhacked for quite some time, but the protection techniques were eventually revealed by some clever work by two researchers at the EADS Corporate Research Center in France and subsequently published at the BlackHat Europe conference in 2006. We'll sketch their attack algorithm REBD [41] below.

The client binary contains hard-coded RSA keys and the IP address and port number of a known server that is used when the client first connects to the network. If you could break the protection and build a new client binary with your own keys and IP addresses, you could create your very own Voice-over-IP network and steal Skype's customer base by undercutting their calling rates.

Here, we'll only concern ourselves with how Skype protects their secrets by obfuscating and tamperproofing the client binary. This can give us interesting and uncommon insights into how protection techniques are actually used in the field! Equally interesting is how Skype obfuscates the network protocols; you can read more about that in reference [41].

Here's an overview of the initial execution stages of the Skype client:

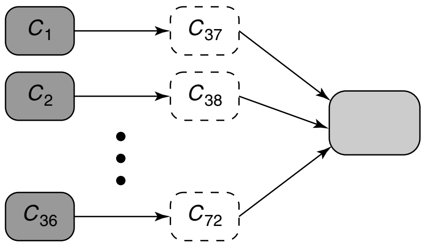


The client starts executing the light gray cleartext code, performing initializations and loading any necessary dynamic linked libraries (dll:s). The execution continues with the dark gray code, which first erases the light gray code and then decrypts the remaining dashed code. The executable itself contains the decryption key, *key*. The encryption is a simple `xor` with the key. Erasing the light gray code after it has executed makes it difficult for the adversary to create a new binary by dumping the in-memory process to a file. From this point on, the binary is in cleartext—it is never re-encrypted. In the final step, the decrypted dashed code gets executed. It starts by loading a *hidden* dll table (dotted), partially overwriting the original one. In total, the client loads 843 dynamically linked libraries, but 169 of these are not included in the original dll table. Hiding the dll table also makes it hard for the attacker to create a new binary by writing out the memory image to a file.

The client continues by checking for the presence of debuggers, using techniques similar to those you saw in Section 7.1.1▸406: It checks for signatures of known debuggers and also does timing tests to see if the process is being run under debugging.

In the final tamperproofing stage, a network of nearly 300 hash functions checks the client code, and also checks each other. Each hash function is different and is based on the `hash7` family of functions you saw in Section 7.2.2▶418. They are executed randomly. The test on the hash function value is not a simple `if (hash() !=` *value*`)` . . . . Instead, the hash function computes the address of the next location to be executed, which is then jumped to.

You've seen the technique of hash functions checking each other before, namely, in Algorithm TPCA. Here, however, the network is much simpler, with each real region (light gray) checked by a large number of checkers (dashed), each of which, in turn, is checked by one other checker (dark gray):



Also, unlike Algorithm TPCA, the Skype client doesn't attempt to repair itself when it has detected tampering. Instead, it simply crashes, but does so in a clever way. On detection, the client allocates a random memory page, randomizes the register values, and then jumps to the random page. This loses track of all the stack frames, which makes it hard for the attacker to trace back to the location where the detection took place.

In addition to the tamperproofing, the client code is also obfuscated. The target address of function calls are computed at runtime, i.e., all function calls are done indirectly. Dummy code protected by opaque predicates is also inserted. The code is also obfuscated by occasionally raising a bogus exception only for the exception handler to turn around, repair register values, and return back to the original location.

Problem 7.4 It is interesting to note that although Skype is a distributed application, it doesn't use any of the distributed tamperproofing techniques you'll see later in Section 7.5▶453. The reason might be that much of the communication is client-to-client rather than client-to-server. Can you think of a way for clients to check each other in a peer-to-peer system without being able to collude?

7.2.4.1 Algorithm *REBD:* Attacking the Skype Client The ultimate goal of an attack on the Skype client is to be able to build your own binary, complete with your own RSA keys. To do that, you need to remove the encryption and tamperproofing.

The first steps of Algorithm REBD do the following:

1. Find the keys stored in the binary and decrypt the encrypted sections.
2. Read the hidden dll table and combine it with the original one, making a complete table.
3. Build a script that runs over the decrypted binary and finds the beginning and end of every hash function.

If you look at `hash7` in Section 7.2.2▶418, you'll notice that the routine has a distinctive structure, consisting of initialization, looping, read memory, and compute hash. Unfortunately (for Skype), there's not enough variability in this code, and it's possible to build a pattern matcher that can reliably find the beginning and end of all the functions.

The next step is to run every hash function, collect their output values, and replace the body of the function with that value. You could just set software breakpoints on every function header, but since software breakpoints change the executable by replacing an instruction with a trap, that is sure to trip the tamper detectors! The solution is to use *hardware* breakpoints, which don't affect the executable. However, processors only have a small number of such breakpoints, typically four. To get past that limitation, you can run Skype twice, in parallel, with both processes under debugging but one using hardware breakpoints and the other software breakpoints. Here's the idea:

4. Run Skype to collect the values computed by all the hash functions, using twin-processes debugging:
   (a) Start one Skype process S_{soft}, setting software breakpoints at the beginning of every hash function.
   (b) Start another Skype process S_{hard}.
   (c) Run S_{soft} until a breakpoint at the beginning of a hash function is reached at some address *start*.
   (d) Set a hardware breakpoint at *start* in the S_{hard} process and also at the end of the hash function, at address *end*.
   (e) Run S_{hard} until *end* is reached.
   (f) Record the result *hash* of the hash computation.
   (g) Restart S_{soft} starting at address *end* and with the return value of the hash function set to *hash*.

5. Replace all hash function bodies with their computed values.

An alternative attack is to run each function in an emulator (see Section 3.2.4▶168) to find the value it computes.

The final step bypasses the obfuscation and removes the tamper response code:

6. Put a breakpoint on `malloc` and wait for a page with the special characteristics of the random tamper response page to be created. Put a hardware breakpoint on the pointer that stores the page address to locate the detection code.

7.2.5 Algorithm *REWOS:* Attacking Self-Hashing Algorithms

When you think about attacking a tamperproofed program, what first comes to mind is directly removing any tamperproofing code from the executable. This, of course, means that you first have to analyze the code, then locate the checkers or responders, and finally remove or disable them without destroying the remainder of the program. But an attack can just as well be *external* to the program, modifying the environment in which it is executed. Algorithm REWOS [379], which we'll show you next, does exactly that. By adding just a few instructions to the memory management routines of the operating system kernel, you can craft an attack that is successful against *all* hash-based tamperproofing algorithms!

The basis for the attack is that modern processors treat code and data differently. Code and data have different access patterns (a small region of contiguous instructions might constitute a loop that fits easily in the cache but which accesses a large scattered data set that doesn't), and hardware designers have taken advantage of this by splitting TLBs (Translation Lookaside Buffers) and caches in separate parts for code and data. In the hash-based algorithms you've seen, code is accessed in two different ways: as code (when it's being executed) and as data (when it's being hashed). So sometimes a function will be read into the I-cache and sometimes into the D-cache. The hash-based algorithms assume that the function, regardless of how it's read, will be the same—i.e., that the code being executed through the I-cache is the same as that hashed through the D-cache. By a simple hack to the operating system, REWOS violates that assumption! The idea is this: Redirect *reads* to the code to the original, unmodified program (this ensures that the hash values will be computed as expected) and redirect *execution* of the code to the modified program (this will ensure that the modified code gets executed). You can see a sketch of this idea in Algorithm 7.3▶436.

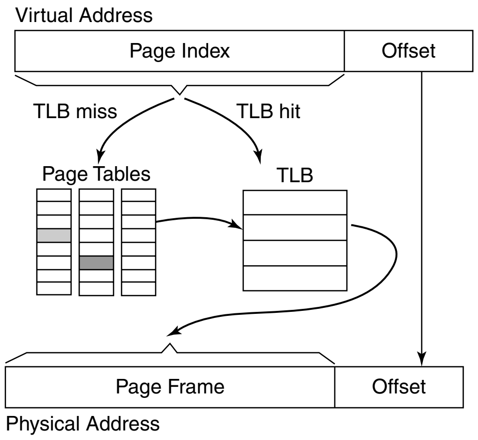
The actual implementation of the attack depends on the capabilities of the processor and the design of the memory management system of the operating system.

Algorithm 7.3 Overview of algorithm REWOS.

```
ATTACK($P$, $K$):
  1. Copy program $P$ to $P_{\text{orig}}$.
  2. Modify $P$ as desired to a hacked version $P'$.
  3. Modify the operating system kernel $K$ so that data reads are directed to
     $P_{\text{orig}}$, instruction reads to $P'$.
```

What we'll show you here is the implementation for the UltraSparc processor, because it displays the basic idea very simply. The details for other architectures will be different, but the principle will remain the same.

Here's a sketch of a typical memory management system:



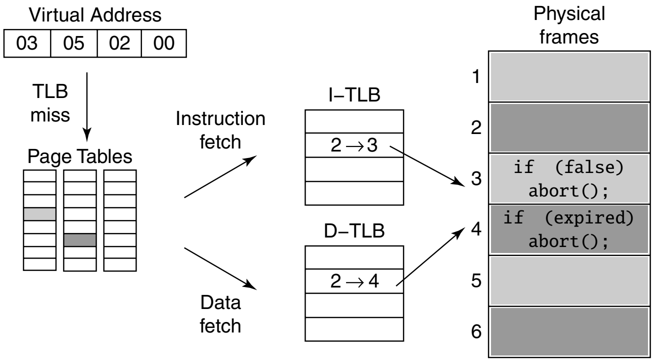
Each process operates within its own seemingly contiguous and large virtual address space. The operating system or the CPU maintains a set of *page tables* that map the virtual addresses seen by the process to physical addresses of the underlying hardware. To avoid having to look up every memory access in the page tables, you first consult a translation lookaside buffer, which caches recent lookups. On a *TLB hit* (which is very fast), you don't have to do anything else, but on a *TLB miss* you must walk the page tables (slow) and update the TLB with the new virtual-to-physical address mapping. Depending on the system architecture, these lookups and updates can be done in hardware or in the operating system kernel.

On the UltraSparc, the hardware gives the OS control on a TLB miss by throwing one of two exceptions depending on whether the miss was caused by a data or an instruction fetch. The OS then looks up the virtual address in the page tables and either updates the instruction TLB or the data TLB with the new virtual-to-physical mapping.

To implement the attack against a self-hashing algorithm, you need to do four things:

1. Copy P to P_{orig}.
2. Modify P however you like.
3. Arrange the physical memory so that frame i comes from the hacked P and frame $i + 1$ is the corresponding original frame from P_{orig}.
4. Modify the kernel so that if a page table lookup yields a $v \to p$ virtual-to-physical address mapping, I-TLB is updated with $v \to p$ and D-TLB with $v \to p + 1$.

Here's an example to illustrate this:



Here, the attacker has modified the program to bypass a license-expired check. The original program pages (dark gray) are interleaved with the modified program pages (light gray) in physical memory. Page 3 and page 4 are actually the same page, except that in page 3 the license check has been hacked. When the program tries to read its own code in order to *execute* it, the processor throws an *I-TLB-miss* exception, and the OS looks up the correct mapping in the page tables and updates the I-TLB to refer to the modified page. When the program tries to read its own code in order to *hash* it, the processor throws a *D-TLB-miss* exception, and the OS updates the D-TLB to refer to the original, unmodified page. The result is that during hashing the original dark gray pages will be read and the tamper-response code won't be triggered, and during execution the light gray code that the attacker has modified to his heart's content will be run.

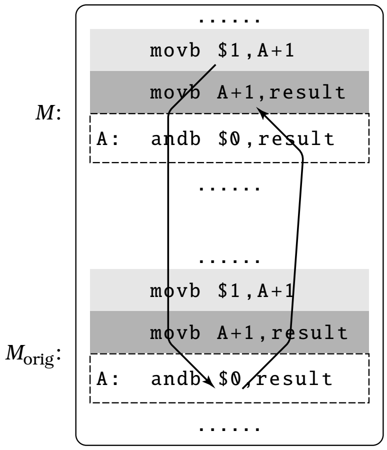
7.2.5.1 Algorithm TPGCK: Detecting Memory Splits Algorithm REWOS attacks tamperproofing through self-hashing by essentially *splitting* the memory into separate pieces, one for code and one for data. Algorithm TPGCK [138,139] uses a self-modifying code trick to detect this split—if a program detects that it's being run in an environment where writes to the code segment don't affect the data segment, it can invoke its normal tampering response.

Have a look at this self-modifying C and assembly code:

```
char result;
asm volatile (
    "     movb $1,A+1\n\t"
    "     movb A+1,\%[output]\n\t"
    "A:   andb $0,\%[output]\n\t"
    "     movb $0,A+1\n\t"
      : [output] "=r" (result));
if (result != 1) {
    printf("UNDER ATTACK!\n\t");
    abort();
}
```

Let's first assume that the operating system hasn't been hacked, i.e., code and data share the same address space. Then, the light gray instruction overwrites the 0 value in the dashed instruction with a 1. The dark gray instruction reads that value back into the `result` variable, and thus `result=1`. The modified dashed instruction, finally, executes, setting `result = result && 1`, i.e., `result=1`. The last instruction resets the dashed instruction back to its original.

But what if you're under attack from algorithm REWOS? Assume, as before, that the attacker has duplicated an original memory page M_{orig} as a new page M. Instructions are executed from the modified M page and data reads are redirected to the unmodified M_{orig}:



The light gray write into the instruction stream will propagate through the data cache and alter the dashed instruction on M_{orig}. When the dark gray instruction tries to read data from the instruction stream, it accesses M_{orig} and the correct value 1. However, the dashed instruction will execute from page M, which was not modified by the light gray instruction. The program will execute `result = result && 0`, resulting in `result=0`.

On modern processors, self-modifying code inflicts a performance overhead: Caches and processor pipelines may have to be flushed whenever the processor detects that the code segment has been modified. In reference [138], the authors estimate that the worst-case cost of the check above will be similar to a lightweight system call.

7.2.6 Discussion

The nice thing about both TPCA and TPHMST is that security is tied to easily modified parameters. In the case of TPCA, you adjust the number of guards and the complexity of the guard graph. In the case of TPHMST, the interval overlap serves the same purpose.

We have no hard performance numbers for either algorithm. Obtaining meaningful numbers is actually difficult, because these will always depend on the number of checkers you insert, where in the executable you insert them, and therefore how often they will run. It's certainly possible to say, "Using the SPEC benchmark suite, we're inserting *n* checkers, each hashing *k* kilobytes of code, at random locations, and we get the following reduction in performance." However, the resulting numbers will tell you very little, since you have no way of relating that to how much security this actually buys you! TPHMST suggests an interval overlap of 6 but gives no support for why this results in the right level of protection.

Checkers are typically small in size, less than 50 x86 code-bytes for TPHMST and more than 62 bytes for TPCA.

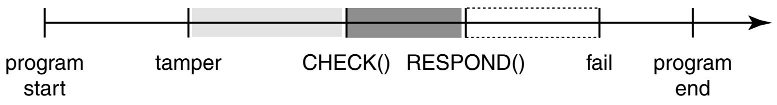
In practical situations, you will not want to insert checkers in random locations, since that may land you inside a tight loop. Instead, you will need some way for the user to specify where checkers should be inserted, what range of code they should check, and how often they should be executed. TPCA's guard graph serves this purpose. They also suggest using a graphical user interface to allow users to interactively select regions to be protected. TPHMST, on the other hand, avoids hot spots by having programmers insert testers manually at the source-code level. If you don't anticipate dynamic attacks (i.e., the code being changed as it is running), then you can reduce the performance penalty by hashing each segment of code only once, right before it is first executed or when it is first loaded into memory.

Problem 7.5 Neither TPCA nor TPHMST has public implementations. It would be interesting to develop a complete implementation that combines the best features of each: the automatic repairs of TPCA and the watermarking and implicit hash values of TPHMST.

Algorithm REWOS is interesting because it is clearly a *class attack*—in one fell swoop it makes *all* self-hashing algorithms impotent. Its one drawback is that it requires an operating system patch, which the average user might be reluctant to or find difficult to install. Algorithm TPCA is the cornerstone of Arxan's (`arxan.com`) *GuardIT* product line. If GuardIT becomes prevalent for desktop applications, it will be interesting to see how long it will take for REWOS-style operating system hacks to become available, and how long it will take Arxan to add TPGCK-style counterattacks.

7.3 Algorithm *RETCJ:* Response Mechanisms

So far, we've told you that tamperproofing happens in two stages, implemented by the functions `CHECK` and `RESPOND`. `CHECK` tests whether the program has been tampered with, sets a flag if it has, and some time later `RESPOND` takes action, making the program fail, phone home, and so on. Actually, however, in a real system you want a *three-pronged* approach, where `CHECK` checks for tampering, later `RESPOND` takes action, and later still, the program actually fails:



This is the basic premise behind algorithm TPTCJ [344]. The idea is that `RESPOND` corrupts program state so that the actual failure follows much later:

```
boolean tampered = false;
int global = 10;
        ...
if (hash(...)!=0xb1acca75) tampered = true;
        ...
if (tampered) global = 0;
        ...
printf("%i",10/global);
```

Here, `CHECK` (light gray) and `RESPOND` (dark gray) communicate through the variable `tampered`, and `RESPOND` manipulates the variable `global` so that the program will eventually fail on a divide-by-zero exception (dashed). It is, of course, possible to merge CHECK and RESPOND so that CHECK sets the failure variable directly.

You could also introduce a number of failure sites and probabilistically choose between them at runtime:

```
#include <time.h>
int global = 10;
        ...
if (time(0) % 2 == 0)
    printf("%i",10/global);
        ...
if (getpid() % 2 == 0)
    x = 5/global;
        ...
x = 3/global;
```

In this way, every time the attacker runs the hacked program, it is likely to fail in one of the two dashed spots. You will need to add a "catchall" failure site (light gray) to make sure that, no matter what, if the program has been hacked it *will* eventually fail.

In general, you want the response mechanism to display the following characteristics:

spatial separation: There should be as little static and dynamic connection between the RESPOND site and the failure site as possible. To accomplish this, you could make them be statically far away from each other by reorganizing the executable. You could also make them be *dynamically* separated by ensuring that RESPOND is not on the call stack when the failure occurs, or by ensuring that as many function calls have occurred between them as possible.

temporal separation: A significant length of time should pass between the execution of RESPOND and the eventual failure, since this will make it harder for the attacker to trace back from the failure site to the response that caused it. At the same time, not *too* much time must pass, or the attack code will be able to cause damage before the program exits.

stealth: The test, response, and failure code you insert in the program should exhibit both local stealth (they have to fit in with the code in their immediate surroundings) and global stealth (they can't be too different from code found in typical programs).

predictability: Once the tamper response has been invoked, the program should eventually fail.

Algorithm 7.4 Overview of algorithm TPTCJ. P is the program to be protected, I is the profiling input, δ is the desired threshold distance between corruption and failure sites, and T is a function-distance matrix. SELECT computes a set of good corruption sites C for each global variable v.

PROTECT(P, I, δ):

1. Execute P with I as input and construct matrix T so that $T[f, g]$ expresses the distance (in terms of elapsed time and number of function calls) between functions f and g.
2. Let $R \leftarrow$ SELECT(P, T, δ) be a set of possible variable/corruption sites.
3. Let $R' \leftarrow$ be a set of random variable/corruption sites from R.
4. Modify P by adding a layer of indirection to any non-pointer global variables in R'.
5. Modify P by inserting tamper-detection code that corrupts the global variables in R'.

SELECT(P, T, δ):

```
$V \leftarrow$ set of $P$'s global variables
$G \leftarrow$ $P$'s call graph
for $v \in V$ do
    $C \leftarrow$ set of functions of $P$
    $F \leftarrow$ set of functions of $P$ in
          which $v$ is used
    for each $f \in F$ do
        for each ancestor $g$ of $f$ in
              the call graph $G$ do
            $C \leftarrow C - \{g\}$
        for each function $c \in C$ do
            if $T[c, f] < \delta$ then
                $C \leftarrow C - \{c\}$
    return $(v, C)$
```

It's important that you keep any legal implications of your tamper-response mechanism in mind. Deliberate destruction of user data is likely to invite legal repercussions, particularly if the user can show that the tamper response was issued erroneously ("I forgot my password, and after three tries the program destroyed my home directory!") But what about data that gets destroyed as an unintended consequence of the tamper response? If the tamper response is the least bit probabilistic (which you would like it to be!), then how can you ensure that the eventual failure happens in a "safe" place? It's easy to imagine a scenario where the program crashes with a file open and the last write still pending, leaving user data in a corrupted and unrecoverable state.

Algorithm 7.4 shows an overview of TPTCJ [344]. The basic idea is for `RESPOND` to set a global pointer variable to `NULL`, causing the program to crash when the pointer is later de-referenced. If the program doesn't have enough pointer variables, TPTCJ creates new ones by adding a layer of indirection to non-pointer variables. The algorithm assumes that there are enough global variables to choose from; while this may be true for procedural programming languages like C, it may be less true for object-oriented languages like Java.

Problem 7.6 In their example program, the authors of TPTCJ found 297 globals over 27,000 lines of code, or one global for every 90 lines of code. Is this normal? How many usable global variables can you find in real code? For cases where there are not enough global variables, can you develop a static analysis algorithm that allows you to create new usable globals, either by "globalizing" local variables or by creating completely bogus new ones?

Here's an example where `main` sets the variable `tampered` to 1 if it detects that the program is under attack:

```
int tampered=0;
int v;

void f() {
   v = 10;
}

void g() {
   f();
}

void h() {
}

int main() {
   if (...)
      tampered=1;
   h();
   g();
}
```

```
int tampered=0;
int v;
int *p_v = &v;

void f() {
   *p_v = 10;
}

void g() {
   f();
}

void h() {
}

int main() {
   if (...)
      tampered=1;
   h();
   g();
}
```

```
int tampered=0;
int v;
int *p_v = &v;

void f() {
   *p_v = 10;
}

void g() {
   f();
}

void h() {
   if (tampered)
      p_v = NULL;
}

int main() {
   if (...)
      tampered=1;
   h();
   g();
}
```

In the first transformation step, the algorithm creates a global pointer variable `p_v` that references a global `v` indirectly. This is the code in light gray. Variable `v` is assigned to function `f`, so one way to make the program crash if it's been tampered with is to set `p_v` to `NULL` before `f` is called. But where? You could set `p_v=NULL` in `g` or `main`, but this would be a bad idea. You want to avoid `g` and `main`, since they will be on the call stack when `f` throws the *pointer-reference-to-nil* exception. Most systems will provide a backtrace of what was on the stack when the program failed, and it would be easy for the attacker to trace back to the cause of the crash using a debugger. Instead, you should choose to insert the failure-inducing code in `h`, which is "many" calls away (dark gray code), and not in the same call chain as `f`.

Problem 7.7 Can you think of interesting ways to construct stealthy tamper-response mechanisms for different classes of programs, such as concurrent programs, distributed programs, database programs, programs with complex graphical user interfaces, and so on? For example, would it be useful to make a concurrent program deadlock or the buttons of a user interface stop responding?

• • •

Problem 7.8 Can you find any statistical anomalies for the kind of code that TPTCJ adds that could be used to stage a counterattack? What do real pointer manipulations look like in a C/C++ program? Is `p=NULL` all that common? Does this usually occur under very special circumstances, such as close to a `malloc`, where a new data structure node is created?

Comparatively little work has been done developing stealthy tamper-response mechanisms. This is a shame, because when the response *isn't* stealthy, it gives hackers a really straightforward entry point for exploring the tamperproofing mechanisms in a program: Run the program until it crashes, use a debugger to examine the call stack, and then trace backwards. Similar techniques can be used to disable TPTCJ: Find the offending pointer variable that caused the failure, restart the program, set a data breakpoint on the variable, run with the same input as before, and then find the location that last changed it. This tells you that a straightforward implementation of TPTCJ may not be enough. You may have to add multiple levels of indirection, and above all, make sure that the program has multiple failure points and multiple locations where `RESPOND` induces the failure.

7.4 State Inspection

There are two fundamental problems with the introspection algorithms in Section 7.2▶412:

1. They perform operations that are highly unusual—real programs typically don't read their own code segment!
2. They only check the validity of the code itself—the user can still affect the program by modifying runtime data, for example, by using a debugger to change a function's return value right before it returns.

So what's the alternative? If you can't check that your program is intact by verifying the correctness of the code, what can you do? Well, you can try to verify the correctness of the *data* or the *control flow*! The idea is that you might be able to detect tampering of the code by the *side effects* the code produces, i.e., in the way that the code makes data change and in the way it forces control to take different execution paths.

In addition to being more stealthy than introspection, these techniques have the advantage that you can use them not only on binary code but also on type-safe code like Java bytecode. You might want to think of this as an advanced form of assertion checking—you'll be adding checks to the program to ensure that at no point does it get into an unacceptable state. You're not checking, "Is this code correct?" but rather, "Is the program in a reasonable state?" The state is a combination of all your static data, the stack, the heap, and the program counter, and you can access all of these whether you're running a binary executable or Java bytecode. (The PC you can't access directly from Java, of course, but you know *where* in the code you are when you're doing the assertion check, which amounts to the same thing.)

Unfortunately, automatically adding assertion checks to a program isn't easy. The current state of a program depends on all previous states, so how could you possibly analyze your user's program to come up with non-trivial invariants to add as checks?

An alternative to adding assertions on the entire state of the running program is to call, say, a function at a time, feeding it challenge data and checking that the result is as expected:

```
   ...
int challenge = 5;
int expected  = 120;
int result    = factorial(challenge);
if (result != expected)
   abort();
   ...
```

"Hash values" (the expected results of the functions, given the challenge inputs) are easy to compute at protection time—just generate random challenge inputs, call the functions, and record the results! Just like you saw in Algorithm TPHMST, however, you have to be careful not to generate suspicious-looking hash values or challenge data that the attacker could easily spot as unusual:

```
if (factorial(17) != 355687428096000)
   abort();
```

Neither 17 nor 355687428096000 is likely to occur frequently in real programs.

If you're willing to sacrifice code space, you could make copies of every function, forcing the adversary to hack two functions simultaneously in order to avoid detection. This will hide the hash value but not the challenge data:

```
int challenge = 17;
if (factorial_orig(challenge) != factorial_copy(challenge))
   abort();
```

Automatically generating challenge data that actually exercises important aspects of a function is not easy, particularly for functions that take complex data structures as input. *Hiding* that data in your program is also an issue. Imagine inserting tests in your program for an all-pairs-shortest-path algorithm:

```
int[][] challenge = {{1,1,0},{0,1,1},{0,0,1}};
int[][] expected  = {{1,1,1},{0,1,1},{0,0,1}};
int[][] result    = warshall(challenge);
for(int i=0;i<3;i++)
  for(int j=0;j<3;j++)
     if (result[i][j]!=expected[i][j])
        abort();
```

It wouldn't be hard for an adversary to pick out this code as being suspicious—real programs typically don't build many large, complex, literal data structures. Functions that get their inputs externally, such as reading from a file or accepting network packets, will also be difficult to challenge.

Another problem when trying to automate this technique is that it's hard to predict what side effects a function might have—if you call it an extra time on challenge data to verify that it hasn't been tampered with, it might have the undesirable side effect of destroying valuable global data or allocating extraneous dynamic memory that will never be properly freed. Have a look at this class:

```
class Widget {
   String thing;
   static List all = new List();
   public Widget(String thing) {
      this.thing = thing;
      all.add(thing);
   }
}
```

If you make many challenges to this class without realizing that it keeps track of every instance ever created, you might get into nasty memory problems. To avoid (some of) these problems, you could challenge functions only at certain times during execution, such as at start-up time, when changes to global state are less likely to have ill effects. Of course, this gives the adversary more opportunities to attack your code.

Finally, what should you do about functions that have non-deterministic behavior? If you're lucky, you'll at least discover this at protection time, by running the function multiple times on the same challenge input and realizing it doesn't always generate the same output. But then what? You can always rely on programmer code annotations, but as programs evolve these are notoriously difficult to maintain.

In the remainder of this section, you will see two *oblivious hashing* algorithms. They're called *oblivious* because the intent is that the adversary should be unaware that his code is being checked. The first algorithm, TPCVCPSJ, inserts hash computations at the source-code level (although there's no reason why it couldn't be done at lower levels). The hash is computed based on the values of variables and the outcome of control flow predicates. The second algorithm, TPJJV, very cleverly hashes binary instructions without actually reading the code!

7.4.1 Algorithm *TPCVCPSJ:* Oblivious Hash Functions

To verify that a program is executing correctly, you could, conceptually at least, collect the entire execution trace, compress it, and compare it to the expected trace. For example, in Section 3.2.3▸163 you saw how Algorithm RELJ compresses a trace into a context-free grammar. In practice, even these compressed traces are too large for our purposes, and as is always the case when you add protection code to a program, you also have to worry about stealth. Algorithm TPCVCPSJ [61,178] computes a *hash* over the execution trace by inserting instructions that monitor changes to variables and control flow. You can balance the level of protection against the amount of overhead by varying the number of hash computations that you insert.

At the source-code level, you can imagine defining a global hash variable and a macro `UPDATE(h,v)`, which uses some combination of arithmetic and logical operations to include a value `v` into the hash:

```
int hash;
#define UPDATE(h,v) (h+=v)
```

You'd then transform assignment statements like this:[2]

2. The expression $(e_1,e_2,\ldots,e_n)$ uses C's comma operator to evaluate the e_is in order and then returns the value of e_n.

```
x = expr
```

⇨

```
int tmp;
x = (tmp=expr,UPDATE(hash,tmp),tmp)
```

To hash the outcome of a conditional expression, you'd perform this transformation:

```
if (expr) ...
```

⇨

```
int tmp;
if (tmp=expr,UPDATE(hash,tmp),tmp) ...
```

Have a look at this example, an extract from a DRM system:

```
int play (int user_key, int player_key,
          int digital_media[], int len) {
   int i=0;
   while(i<len) {
      printf("%f\n",(float)(user_key^player_key^
                            digital_media[i]));
      i++;
   }
}
```

The procedure unlocks an encrypted media file by decrypting it with a user and a player key. To protect `play`, you first modify it to compute a hash value using the transformations above to insert calls to the `UPDATE` macro:

```
int play (int user_key, int player_key,
          int digital_media[], int len) {
   int i=0;
   int t;
   while(t=i<len,UPDATE(hash,t),t) {
      printf("%f\n",(float)(t=user_key,UPDATE(hash,t),
                            t^player_key^digital_media[i]));
      i=(t=i+1,UPDATE(hash,t),t);
   }
}
```

You then generate some challenge input for `play`, run it on that input, and record the result. At some appropriate places in the program, you then insert a challenge call and check the result against the expected hash value:

```
int player_main (int argc, char *argv[]) {
   ...
   int user_key = 0xca7ca115;
   int player_key = 0xbabeca75;
   int digital_media[] = {10,102};
   hash = 0;
   play(user_key, player_key, digital_media, 2);
   if (hash != -1795603921) {
      printf("HACKED!");
      abort();
   }
   ...
}
```

Regardless of whether you call `play` as part of the regular computation or as a challenge, the hash is still computed, so there will be a general slowdown, depending on how many hash computations you add.

Global variables are quite uncommon, so to avoid inserting code that is too unstealthy you can pass the hash variable as an extra argument to the function:

```
int player_main (int argc, char *argv[]) {
   int hash = 0;
   play(user_key, player_key, digital_media, 2, &hash);
   ...
}

int play (int user_key, int player_key,
          int digital_media[], int len, int* hash) {
   ...
}
```

Also for stealth reasons, you should vary the operators used in the hash computation to match those found in the function you're protecting. This is easy to do by providing a collection of `UPDATE` macros:

```
#define UPDATE1(h,v) (h+=v)
#define UPDATE2(h,v) (h^=v)
#define UPDATE3(h,v) (h-=v)
```

A serious issue is what to do with functions that have side effects. You could always add an extra parameter to functions that access a global variable and pass a reference to the global variables along to all calls. This assumes you can do an accurate static analysis to determine which functions need access to which global variables (either to use themselves or to pass along to functions *they* call that may need them), and this becomes difficult in the presence of function pointers, for example.

An even worse problem is functions that are non-deterministic because they depend on the time of day, network traffic, thread scheduling, and so on. You could try to statically detect such locations in the code, you could rely on code annotations provided by the programmer, or you could try to run the program a few times on the same data and find locations where the computed hash varies. None of these solutions is particularly attractive. Static analysis of non-deterministic programs is likely to be hard and produce overly conservative results, programmer annotations are notoriously unreliable, and dynamic analysis is unlikely to catch all non-deterministic locations in the program.

7.4.2 Algorithm *TPJJV:* Overlapping Instructions

The x86 architecture has some interesting properties that allow you to play cute tricks unavailable on a RISC architecture. In particular, the x86 uses a variable-length instruction encoding (instructions can be anywhere from one to fifteen bytes long) and has no alignment requirements for instructions (an instruction can start at any address). This allows you to have one instruction *inside* another, or one instruction *overlapping* another, and to share instructions between two blocks of code. Overlapping two blocks in itself adds a level of tamperproofing, since modifying one instruction will affect both pieces of code.

Algorithm TPJJV [178,179] takes the tamperproofing to another level by overlapping basic blocks of x86 instructions so that when one block executes, as a side effect it also computes a hash over the second block. The hash value can then be compared to the expected value, much as you've seen in previous algorithms. The real advantage of this technique is that (unlike Algorithms TPHMST and TPCA) the hash is computed *without* explicitly reading the code. This makes the algorithm invulnerable to memory-splitting attacks like REWOS.

Here are two basic blocks with entry points B_0 and B_1:

B_0:

```
shll 2,%eax
incl %eax
ret
```

B_1:

```
decl %eax
shrl 3,%eax
ret
```

The easiest way to merge them is to interleave the instructions and insert jumps to maintain semantics:

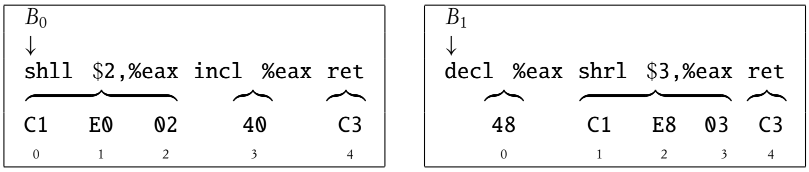
```
B0:
        shll 2,%eax
        jmp  I1

B1:
        decl %eax
        jmp  I2

I1:
        incl %eax
        jmp  I3

I2:
        shrl 3,%eax

I3:
        ret
```

The merged block has two entry points, B_0 and B_1. This merging by itself doesn't accomplish much. What you really want is for the two blocks to also *share instruction bytes*. To accomplish this, you replace the jump instructions with bogus instructions that take a (large) immediate value as one of its operands. This operand will *mask* out the instruction from the other basic block by including the instruction in its immediate operand. Look here:

```
B0:
        shll 2,%eax
        xorl %ecx,next 4 bytes // used to be jmp I1
B1:
        decl %eax
        jmp  I2
        nop
        incl %eax
        ...
```

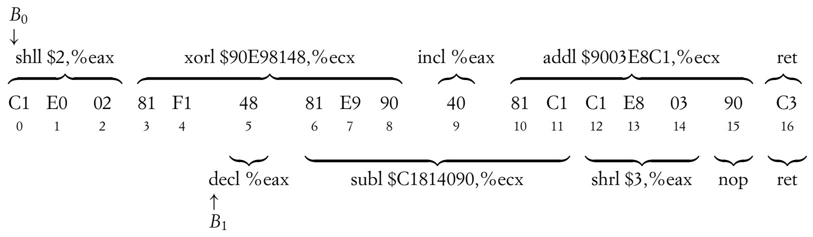
What happened? Well, we replaced the jump from B_0's first instruction to its second with an `xorl` instruction. This instruction takes a four-byte immediate operand, the first byte of which will be the `decl %eax` instruction, the second and third byte will be `jmp` I_2, and the fourth will be a `nop` that we had to add as padding. What does this mean? Well, at runtime when you jump to B_1, you will execute the `decl` instruction just as before and then jump to B_1's second instruction, also as before. In other words, the B_1 block will execute unchanged. When you jump to B_0, however, things are a little different. You start by executing the `shll` instruction as before, and then proceed to execute the new `xorl` instruction, which has embedded in its immediate operand the four bytes from `decl;jmp;nop`! After the `xorl`, you go straight to B_0's second instruction, `incl`. If you had properly initialized register `%ecx`, the `xorl` instruction would compute a hash over the instructions from block B_1. You could test this value later, and if it's incorrect execute the appropriate response.

Let's look at this example in more detail. Here are the two blocks again, but this time we give the x86 code bytes explicitly:



Each block has three instructions, which translates into five code bytes each. We've given the offset of each code byte below the byte and indicated the start of execution for each block with an arrow.

After merging the two blocks and inserting hashing and padding instructions, you get the following seventeen code bytes:



On top of the bytes, we indicate what instructions we execute when we start at offset 0 in order to execute block B_0. Below the code bytes, we show what gets executed when you start at offset 5 in order to execute block B_1. In either case, as you execute one block, you also compute a hash over the other block into register `%ecx`. If the adversary were to change the byte at offset 9, for example, changing the `incl %eax` instruction to something else, this will be caught when you execute B_1 and use the `subl` instruction to compute the hash.

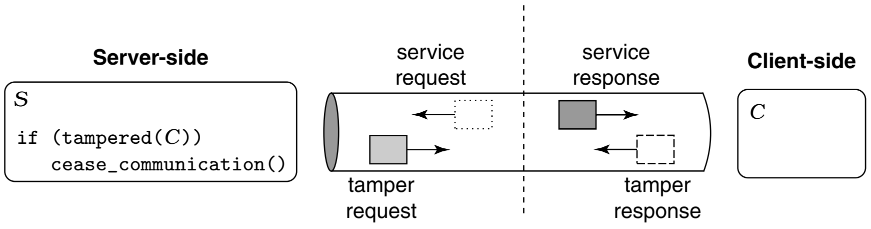
One problem with this algorithm is that it's hard to predict its precision. When will you actually detect tampering? In our example, you won't detect any tampering of B_0 until you execute B_1 and when that occurs will, of course, depend on the flow of control.

Whether this algorithm as *state inspection* or *introspection* is debatable. Since you're hashing instructions, this makes the algorithm closer in flavor to TPHMST and TPCA, but on the other hand when you're actually computing the hash you're working on runtime values (or, at least, runtime immediate operands of instructions)! Classification aside, this is a clever use of the x86's architectural (mis-)features.

The overhead depends on the level of overlap. Jacob et al. [178] (from where the example above has been adapted) report that the protected binary can be up to three times slower than the original.

7.5 Remote Tamperproofing

Remote tamperproofing is an important special case of tamperproofing. In this scenario, the program you want to protect (which we'll call C) runs remotely on the adversary's untrusted site (the *client* site) but is in constant communication with a trusted program S on your (*server*) site. In addition to providing services to the client, the server wants to detect and respond to any tampering of C:



As in typical client-server scenarios, the client sends *request-for-service* packets (dotted) over the network to the server, which returns *service-response* packets (dark gray). In a computer game, for example, the client may tell the server, "I just entered dungeon 372!" to which the server responds with a list of the nearby monsters. The server may also ask the client for information about the state it's in (or deduce that state from the service-request messages it receives) in order to detect if C is under attack.

There are many applications that fit neatly into this model. In networked computer games, for example, the game server provides continuous information to the player clients about their surroundings. Players will often try to hack the clients to gain an unfair advantage over other players, for example, by being able to see parts of a map (which, for performance reasons, is being held locally) that they're not supposed to see.

7.5.1 Distributed Check and Respond

Just like in single-processor tamperproofing, in the client-server scenario you need two functions, `CHECK` and `RESPOND`. As you've seen, responding to tampering can take many forms, including reporting violations back to you. In the client-server scenario, response is even simpler: Just terminate communication! The assumption here is that the server is providing a service without which the client can't make progress, and this means that refusing to provide that service is the ultimate punishment.

Unfortunately, while responding is easy, the `CHECK` function is harder to implement. The reason is that you have no direct access to C's code—it's running on a remote site! For example, while it's certainly possible for you to ask the client to send you a hash of C's code to verify he hasn't tampered with it, there's nothing stopping him from lying about the hash value!

There are four classes of attacks:

1. The attacker can reverse engineer and modify C's code.
2. The attacker can modify the running environment of C, for example, by running it under debugging or emulation, or under a modified operating system.
3. The attacker can execute multiple simultaneous instances of C, some of which might have been modified.
4. The attacker can intercept and replace network messages.

7.5.2 Solution Strategies

Many solutions to the remote tamperproofing problem are variants of various levels of "sharing" the C code and data between the server and the client. At one extreme, all of the C code and data resides on and is executed by the server. This is sometimes known as *software as a service*. Whenever the client wants to make progress, it has to contact the server, passing along any data it wants the server to process, and wait for the server to return computed results. This kind of server-side execution can lead to an unacceptably high compute load for the server and unacceptably high latency for the client. On the other hand, since all the sensitive code resides server-side, there is no risk of the client tampering with it. At the other extreme, the client runs all its own code and does all the work. This requires the server to share all its data with the client, which can be bandwidth-intensive. Since all computation is done client-side, it's more difficult for the server to guarantee that the client has not tampered with the code. Most systems will settle on an intermediate level solution: Some computation is done server-side, some client-side, and this balances computation, network traffic, and tamper-detection between the two. In Section 7.5.3 we'll show you Algorithm TPZG, which automatically splits the computation of a sequential program over the client and the server. It was originally designed to protect against piracy, but subsequent development [57] has extended this idea to the remote tamperproofing scenario.

A second idea is to extend the introspection algorithms of Section 7.2▸412 to the client-server scenario. The idea is for the server to ask the client to compute a hash over its code and compare it to the expected value. However, this isn't enough, since there's nothing stopping the client from lying about the hash! Algorithm TPSLSPDK(Section 7.5.4▸459) solves this problem by measuring the time it took for the client to compute the hash, making sure that it didn't have time to tamper with the code.

Finally, Algorithm TPCNS (Section 7.5.5▸462) extends the idea of dynamic obfuscation from Chapter 6 to the client-server scenario. The idea is for the server to force the client to accept newly obfuscated versions of its code, ideally at such a quick pace that it can't manage to keep up its reverse engineering efforts.

7.5.3 Algorithm *TPZG:* Slicing Functions

Algorithm TPZG [382] was developed to prevent piracy. The idea is to slice the program into an open part that resides on the client and a hidden part that resides on the server. Extra communication is added so that the client can access those parts of the code that reside on the server. Assuming that this extra communication doesn't leak too much information, it should be difficult for the adversary to reconstitute the hidden parts. This prevents piracy (if the adversary can't get access to all the code, he can't copy it!) but also prevents tampering with the part of the code that is hidden on the server.

There are three main obstacles that make this idea impractical. First, being tied to a server works well when you're connected to a network but less well when you're on an airplane. This is a problem for piracy prevention, but not for remote tamperproofing, where the assumption is that you *are* always connected. The second

Algorithm 7.5 Overview of algorithm TPZG. f is a function selected for splitting, and v is a local variable in f.

```
PROTECT($f$, $v$):
  1. Compute a forward slice of $f$, starting with the statements that define $v$.
  2. Determine which variables should be completely hidden (i.e., should reside
     only on the server) and which should be partially hidden (i.e., should reside
     both on the client and the server).
  3. Examine each statement in the slice and split it between the client function Of
     and the server function $\mathtt{Hf}_i$.
  4. If $x$ is a partially hidden variable, then
     • translate $x \leftarrow rhs$ to $x \leftarrow rhs$; $\mathtt{Hf}_i(x)$ where $\mathtt{Hf}_i(x)$ updates $x$ on the
       server.
     • translate
           $lhs \leftarrow \ldots x \ldots$
       to
           $x \leftarrow \mathtt{Hf}_i()$;
           $lhs \leftarrow \ldots x \ldots$
       where $\mathtt{Hf}_i(x)$ gets the current value of $x$ from the server.
```

problem is *latency*. Networked applications are carefully designed to tolerate high latency—functions where high latency is unacceptable are kept on the client side, and the remaining functions can be kept on the server. If you move additional functions to the server side, latency may become intolerable. The final problem is *bandwidth*. If the client keeps large data structures and some of the operations on these structures are moved to the server, there may not be enough bandwidth to move the data back and forth.

Algorithm TPZG bypasses the network latency and bandwidth problems by only considering scalar data (functions on arrays and linked structures are kept on the client) and restricting the client and server to both run on the same local area network.

Algorithm 7.5 sketches how to split a function f into one part, `Of`, which runs on the client, and several parts, $\mathtt{Hf}_i$, which run on the server and which the client accesses through remote procedure calls.

Let's look at a simple example to illustrate the algorithm. Below left is the original function `f` that runs client-side. You've determined that you want to hide variable `a` on the server. You start by computing a forward slice on `a` (see Section 3.1.5▶141). We show this in light gray to the right:

```
static int f(int x, int y) {
   int a = 4*x + y;

   int c;
   if (y < 5)
      c = a*x+4;
   else
      c = 2*x+4;

   int sum = 0;
   for(int i=a;i<10;i++)
      sum += i;

   return x*(sum+c);
}
```

```
static int f(int x, int y) {
   int a = 4*x + y;

   int c;
   if (y < 5)
      c = a*x+4;
   else
      c = 2*x+4;

   int sum = 0;
   for(int i=a;i<10;i++)
      sum += i;

   return x*(sum+c);
}
```

You want to protect all the light gray code, so you put it server-side in the six functions `Hf1` ... `Hf6`:

```
static int Ha = 5;
static int Hc = 0;
static int Hsum = 0;

static void Hf1(int x, int y) {
   Ha=4*x+y;
}
static boolean Hf2(int y, int x) {
   if (y < 5) {
      Hc = Ha*x + 4;
      return true;
   } else
      return false;
}
```

```
static void Hf3(int c) {
   Hc = c;
}
static void Hf4(int sum) {
   Hsum = sum;
}
static void Hf5() {
   for(int i=Ha;i<10;i++)
     Hsum += i;
}
static int Hf6() {
   return Hsum+Hc;
}
```

Finally, you rewrite `f` to `Of`, the new function that will run client-side:

```
static int f(int x, int y) {
   int a = 4*x + y;


   int c;
   if (y < 5)
      c = a*x+4;
   else
     c = 2*x+4;


   int sum = 0;
   for(int i=a;i<10;i++)
      sum += i;


   return x*(sum+c);
}
```

```
static int Of(int x, int y) {
   Hf1(x,y);



   int c;
   if (!Hf2(y,x)) {
      c = 2*x+4; Hf3(c);
   }



   int sum = 0; Hf4(sum);
   Hf5();



   return x*Hf6();
}
```

The client accesses the hidden functions by making remote procedure calls to the server. The variable `c` is *partially hidden*. This means that, for performance reasons, it resides both on the client and the server, and the code that updates it is split between the two.

Zhang and Gupta [382] report runtime overhead from 3% to 58%, but as always, this depends on the amount of protection that is added, in this case, how much of the program is hidden on the server and the amount of extra communication overhead this introduces. Zhang and Gupta's measurements were done over a local area network. In many scenarios, it is more likely that the server and client are farther away on the network and the extra latency that is introduced may well make this method too slow, or at the very least, will significantly reduce the size of the slice that can remain hidden. Here's a rough measurement of packet turnaround times, as reported by `tracepath`, starting at `york.cs.arizona.edu`:

target site	**# hops**	**ms**
`rorohiko.cs.arizona.edu`	1	0.2
`cse.asu.edu`	10	5
`www.stanford.edu`	12	25
`www.usp.ac.fj`	12	153
`www.eltech.ru`	23	201
`www.tsinghua.edu.cn`	19	209

Going to `www.stanford.edu` from `york.cs.arizona.edu` is 125 times slower than staying within the local area network, even though Stanford is geographically in the state next to Arizona.

7.5.4 Algorithm *TPSLSPDK:* Measuring Remote Hardware

If you find yourself in a *very* restricted environment, it should be possible to *measure* aspects of the untrusted client to verify that it is running the correct software. This is the premise behind Algorithm TPSLSPDK [322–324], known as the *Pioneer* system. After the Pioneer protocol has run, the server can be sure that

1. a particular executable E on the client hasn't been modified,
2. the client has executed E, and
3. E was not tampered with during execution.

To get these guarantees, you have to assume a system configuration with some very limiting properties:

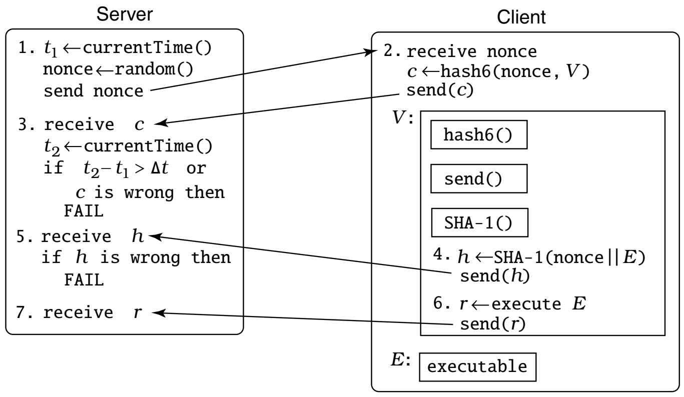
1. The server knows the exact hardware configuration of the client, including CPU model, clock speed, memory latency, and memory size. The client only has one CPU.
2. The communication latency between the server and the client is known, for example, as a result of their being on the same LAN segment of the network.
3. The network is configured so that during verification the client is unable to communicate with any system other than the server.
4. The executable the server wants the client to run should not need to invoke any other software on the client, and it can execute at the highest processor privilege level with interrupts turned off.

Given these restrictions, the server can ask the client to compute a hash over its own code and return the value to the server. The hash function is carefully constructed so that if the client tries to cheat, he will either return the wrong hash value or the hash computation will take longer than expected to compute. This explains the very strict requirements above: If the server doesn't know the client's computational power and the speed of communication, it can't estimate what is a reasonable time in which to compute the hash, and if the client isn't prevented from arbitrary network activity, it could farm out the hash computation to a faster machine. A successful cheat by the client means that he is able to return the correct hash value within the expected time while at the same time running tampered code.

7.5.4.1 Applications In spite of such a restricted scenario, there are potential applications. For example, say that you want to check a small device such as a cell phone, PDA, or smartcard for viruses. You could plug it in directly to your computer over a dedicated wire and use the Pioneer protocol to verify that the phone is running uncompromised code. Similarly, if you're a network administrator, you could configure the routers on your LAN so that the machine you want to check for malware cannot communicate with any other machine during the verification process. Or say you're an inspector of voting machines. You could unplug a machine from its network connection, connect it to your laptop, and then use Pioneer to check that the machine is running the correct certified code.

In all these cases, it is reasonable to assume that you have perfect knowledge of the computing power of the client—you just have to make sure that your communication with it has predictable latency and that there is no way for it to communicate with other systems. Given that, Pioneer guarantees that the desired executable is loaded into memory and executed, and that there is no rogue code on the client that can interfere with the execution.

7.5.4.2 The Pioneer Protocol Have a look at this overview of the Pioneer protocol (numbers indicate the order of events):



On the client-side is the executable E, which the server wants the client to run untampered. At the heart of the system is the verifier V, which the client will use to prove its trustworthiness to the server. V includes the three functions `hash6()`, `send()`, and

`SHA-1()`. The `send()` function is used for data transfer with the server, `hash6()` we defined in Section 7.2.2▸418, and `SHA-1()` is a cryptographic hash function.

In the first steps of the protocol, the client must convince the server that none of the functions in V has been tampered with. The client can then use these functions to send the server a SHA-1 hash over E. If the hash matches, the server can be sure that E has not been tampered with. The final step is for the client to run E and return the result to the server.

To convince the server that he hasn't tampered with V, the client sends him a hash (computed by `hash6()`) over the verifier code. Notice that `hash6()` computes the hash over itself, thus verifying itself! The server then compares the hash against the expected value.

There are several potential problems here. First, the client could pre-compute the hash value, tamper with the code, and when challenged, send the server the value it expects. The protocol, therefore, starts with the server creating a *nonce* (a random value) and sending it to the client. The client is forced to use the nonce to initialize `hash6()`, and this makes it impossible for him to cheat by pre-computing the hash value.

The second problem is that the client can cheat by executing extra instructions. To prevent this, the server measures the time it takes for the client to compute the hash over V. This leads to a third problem: The client could run an optimized version of `hash6()` that allows him to slip in extra instructions and still return the hash value within the expected time! For this reason, `hash6()` has been designed to be *time optimal*, i.e., there should be no way to speed it up by reordering instructions, replacing one instruction with another, and so on.

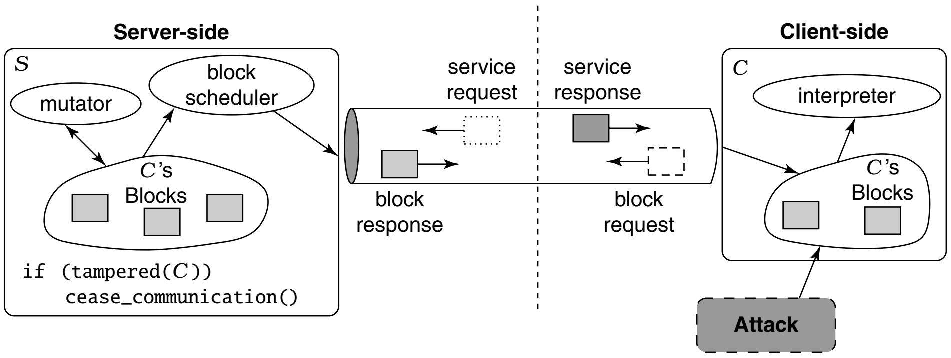
In step 3 of the protocol, the server verifies that the client has computed the expected hash value over V within the expected amount of time. In steps 4 and 5, the client computes a SHA-1 hash of the executable E and the server compares it against the expected value. Again, the client has to factor in a nonce to prevent precomputation. At this point, the server knows that E hasn't been tampered with. In steps 6 and 7, finally, the client runs E and transfers its return value to the server.

You must also make sure that no other process is running on the system until after V and E have finished executing. To accomplish this, you must require V and E to run at the highest CPU privilege level with all maskable interrupts disabled. We don't show it in the protocol sketch above, but the hash computation in step 2 also includes processor state in order to assure the server that the V and E run unperturbed. During step 2, you therefore install new handlers for all non-maskable interrupts and exceptions.

Problem 7.9 This algorithm extends the code-hashing idea to the remote tamperproofing scenario. Can you do the same for the *oblivious* hashing idea from Section 7.4▶444?

7.5.5 *TPCNS:* Continuous Replacement

The final idea to prevent tampering of remotely executing code that we're going to show you we call *remote tamperproofing by continuous replacement*. The basic idea is to make the client code difficult to analyze by keeping it in constant flux, i.e., by continuously obfuscating the client code. Ideally, the adversary will find that the client code is changing so quickly that before he has managed to analyze and modify the current version, the server has generated a new one. Algorithm TPCNS [83] presents this high-level overview of a continuous replacement system:



In this design, both the server and the client maintain a representation of the *C* code (the code run by the client) in what we'll call a *bag of blocks*. The client executes out of its bag by selecting a block to run, jumping to it, selecting the next block, and so on. The server, on the other hand, has a *mutator* process that continuously modifies its bag of blocks and shares any modified blocks with the client. Sharing can happen if the client asks the server for a block it doesn't have (at any one point in time, the client might hold only a subset of all the code blocks), sending a *request-block* packet (in dashed) to the server, and getting a new block (in light gray) in return. The server may also *push* blocks onto the client. A *block scheduler* process on the server determines which blocks to return or push to the client and at what time.

The level of tamperproofing you achieve through this setup is determined by

1. the rate at which the server generates mutated blocks and pushes them onto the client, and
2. the rate at which the adversary can analyze the client code by monitoring the continuously changing bag of blocks.

To reduce network traffic, you want to keep the block replacement rate as low as possible. At the same time, you want to make the rate high enough to be sure the client doesn't have enough time to analyze the program between updates!

The obfuscating block transformations should have the following properties:

- The server must be able to apply the transformations over and over again, creating an infinite stream of differently obfuscated blocks.
- It should be more resource-consuming for the client to analyze new blocks than for the server to generate them.
- The client shouldn't be able to simply ignore new blocks pushed to it.

The last point is important. An adversary can simply monitor the bag of blocks, and once it has received all of *C*'s code, just take a snapshot of the bag and analyze it off-line.

Several of the obfuscating transformations you saw in Chapter 4 (Code Obfuscation) can be applied multiple times. A simple implementation of TPCNS could, for example, have only two transformations, one that splits a block in two and another that merges two blocks together. Together these two transformations would generate an infinite sequence of different blocks.

Unfortunately, such transformations won't stop the adversary from ignoring new blocks sent to it and simply executing old blocks it has already seen, analyzed, and tampered with. To prevent such attacks, TPCNS requires the server to support a third type of transformation, *interface obfuscation*. The idea is to continuously modify the remote procedure call (RPC) interfaces by which the server provides services to the client. The server can transform these RPCs by, for example, renaming them, reordering their arguments, adding bogus arguments, changing argument types, and splitting and merging calls. An adversary who chooses to ignore block updates will eventually execute an old block that issues an expired RPC. This will alert the server that the client is under attack.

The main advantage of continuous replacement tamperproofing is that you have several knobs that you can tweak to balance the level of protection versus performance degradation:

1. You can increase the rate of block push (the server telling the client to invalidate an old block and replace it with a new one) in order to increase the amount of analysis work the client has to do, at the cost of increased network traffic.
2. You can make blocks more or less obfuscated using any of the algorithms in Chapter 4 (Code Obfuscation), with more obfuscation leading to longer analysis times but worse client performance.

There's no known implementation of the continuous replacement idea. We therefore don't yet know the push rate and level of obfuscation necessary to ward off attacks. It seems reasonable to believe, however, that "reasonable" replacement and obfuscation rates would be enough to defend the client against manual attacks, i.e., attacks where the adversary analyzes the client code by executing it interactively under a debugger. Whether it's enough to protect the client against automated attacks (where the adversary attaches a static analysis tool to the client in order to analyze new blocks automatically as they appear in the block bag) is an entirely different question.

7.6 Discussion

Tamperproofing addresses the *trustworthiness* of a piece of code and the environment in which it executes. As the producer of a program, you want to include certain restrictions on how it's used—maybe it can't be executed after a certain period of time—and you include code in the program to check that the conditions of use are met. If a user changes the program in any way, you can no longer trust it, since these usage checks may have been disabled. To trust a program, you have to be sure that absolutely nothing about it has changed: code can be neither removed nor added. If, for example, you have *removed* pieces of the program—to distribute a partially functional trial version, for example—you want to be sure that no one adds the missing pieces back in.

We've shown you five basic methods for making a program difficult to tamper with:

1. You can add code to the program that checks that the original bits have not been changed (Algorithms TPCA and TPHMST, and for remote tamperproofing, TPSLSPDK);
2. You can add code to the program to check that it's not running in a hostile environment (Algorithm TPGCK);

3. You can check that the program's runtime data structures are always in acceptable states and that the control flow follows reasonable paths (Algorithms TPCVCPSJ and TPJJV);
4. You can split the program into two pieces, where one piece is protected from the attacker by running it remotely or on tamper-resistant hardware (Algorithm TPZG).
5. You can use the obfuscation algorithms from Chapter 4 (Code Obfuscation) and Chapter 6 (Dynamic Obfuscation) to make the program harder to understand and hence tamper with (Algorithm TPCNS uses this for remote tamperproofing).

In practice, these algorithms are often combined. For example, we normally don't think of obfuscation as a tamperproofing technique in its own right, since it lacks the ability to execute a *response* when an attack has been discovered. However, obfuscation plays an important role in making tamperproofing code more stealthy. You saw this in Section 7.2.2▸418, where we obfuscated the hash functions used in introspection algorithms in order to make them less susceptible to pattern matching attacks. Introspection algorithms like TPCA and TPHMST are also commonly augmented with code to check if the program is running under a debugger or emulator.

In Section 2.2.10▸110, we presented the **detect-respond** defense model primitive. In the model, an object is protected by monitoring a desirable invariant and executing a response if the check fails. Algorithms TPCA, TPHMST, TPSLSPDK, TPCVCPSJ, and TPJJV all fit in this primitive. For TPCA and TPHMST, the invariant is a simple hash of the code, whereas for TPCVCPSJ and TPJJV, the invariant is the hash of runtime data values.

The TPZG algorithm first applies the **split** primitive, breaking up the application into two parts and making one part inaccessible to the attacker. Since the hidden part is protected by running it on a remote server or on tamperproof hardware, the attacker can't tamper with it at all. TPZG can combine splitting with tamperproofing by monitoring the communication between the two parts of the program, reporting any suspicious exchanges as possible tampering.

Problem 7.10 None of the algorithms in this chapter has public implementations, so it would be interesting to develop working systems that could be compared for efficiency and resilience to attack.

This page intentionally left blank