Virtual Leashing: Internet-Based Software Piracy Protection

Ori Dvir
Computer Science Department
Tel Aviv University
Ramat Aviv 69978, Israel
oridvir@hotmail.com

Maurice Herlihy
Computer Science Department
Brown University
Providence, RI 02912, USA
herlihy@cs.brown.edu

Nir N. Shavit
Computer Science Department
Tel Aviv University
Ramat Aviv 69978, Israel
shanir@cs.tau.ac.il

Abstract

Software-splitting is a technique for protecting software from piracy by removing code fragments from an application and placing them on a remote trusted server. The server provides the missing functionality but never the missing code. As long as the missing functionality is hard to reverse-engineer, the application cannot run without validating itself to the server.

Current software-splitting techniques scale poorly to the Internet because interactions with the remote server are synchronous: the application must frequently block waiting for a response from the server. Perceptible delays due to network latency are unacceptable for many kinds of highly-reactive applications, such as games or graphics applications.

This paper introduces virtual leashing, *the first* non-blocking *software-splitting technique. Virtual leashing ensures that the application and the server communicate asynchronously, so the application's performance is independent (within reason) of large or variable network latencies. Experiments show that virtual leashing makes only modest demands on communication bandwidth, space, and computation.*

1 Introduction

Software piracy is an enormous economic problem, reaching by some estimates a worldwide level of 40% in 2001 [1]. More importantly, piracy is a key obstacle in the way of electronic distribution of software, especially models such as software rental, secure try-before-buy, and so on. Existing technologies for protecting and controlling software using tamper-resistant hardware and software are based on a variety of cryptographic means, from wrapping (encrypting) parts of the code, to planting calls to cryptographic authentication modules. While these technologies protect against casual piracy, they are fundamentally insecure. Software must be unwrapped before it can be executed, and can then be captured, and cryptographic tests can be removed using widely-available tools.

Software-splitting [11] is a conceptually simple and appealing technique for protecting software from piracy. Remove small but essential components from the application and place them on a secure server, either on a secure coprocessor or across the Internet. The server provides the missing functionality, but never the missing components. If reverse engineering the components from the functionality is hard, the server will have absolute control over the conditions under which the software can be used.

Simple as it sounds, the software-splitting approach to security faces formidable technical obstacles. Perhaps the most daunting challenge is overcoming *communication latency* If application communicates with the trusted server over a network, then network delays can be long and unpredictable. Nevertheless, it is usually unacceptable for the application to block waiting for a response from the server. This non-blocking requirement is particularly compelling for highly-reactive applications such as games and graphics applications, where any perceptible delay will be unacceptable to users. (Communication latency is why most desktop applications cannot be run on remote ASP servers). Similar arguments apply to on-board secure coprocessors, USB devices, or smart cards, as their processors are likely to be substantially slower than the main processor, and the need to buffer data and to share a system bus with other activities (such as memory access) implies that communication delays can be substantial and unpredictable.

In this paper, we propose the first non-blocking software-splitting technique. We believe that this technique, called *virtual leashing*, provides a practical and effective defense against piracy of certain kinds of software (as discussed below). Experiments show that it has surprisingly low computation and communication costs.

1.1 Virtual Leashing

Applications typically perform two kinds of tasks: *active* tasks must be executed immediately, while *lazy* tasks may be executed at any point within a reasonable duration. Virtual Leashing splits an application into two new programs: a large *client* program carries out the original application's active tasks, while a much smaller *server* program carries out lazy tasks. As the names suggest, the client program is executed directly by the end-user, while the server program is executed on a secure off-site platform or on a secure co-processor.

The client and server programs communicate as follows. Whenever the original application would execute a lazy task, the client sends a message to the server, who executes that task and returns the results to the client. Because the offloaded tasks are lazy, the client is still able to react to interactive demands, even in the presence of some latency in the client-server communication.

To make software-splitting practical, we must identify a class of lazy tasks present in a wide variety of applications. There must be an effective way to split the application into client and server components without a detailed understanding of the application itself. Moreover, it must be difficult for a pirate to reverse-engineer the missing tasks by inspecting the client program, and by eavesdropping or tampering with the client/server message traffic. Finally, executing the missing tasks at the server should place modest demands on server computation and client/server bandwidth.

One task common to a wide range of applications is *dynamic memory management*. Allocating memory is an eager task: an application that calls `malloc()` needs that memory immediately. By contrast, freeing memory is lazy: an application that calls `free()` will not block if there is a reasonable delay between the `free()` call and the time when that memory actually becomes available for reuse. This asymmetry lies at the heart of virtual leashing[1]

Virtual leashing splits the original application's memory management activities between the client and server. Where the original application would have allocated a memory block, the client also allocates the block, but sends a message to the server. Where the original application would have freed a memory block, the client simply sends a message to the server. The client also sends the server a large number of "decoy" messages, ignored by the server, that are indistinguishable from the allocation and free messages. The server maintains an image of which parts of the client's memory are in use, and periodically sends the client a message releasing unused memory. As long as the client and server remain in communication, the client will be able to allocate memory without delay. Without such communication, however, the client program will quickly run out of memory.

The key to leashing's security is the practical difficulty of figuring out when memory can be freed. (We are all familiar with stories of programmers who spend inordinate amounts of effort fixing memory leaks in programs they themselves devised.) We will argue that even though one can disassemble the client code, eavesdrop on client-server message traffic, and even tamper with that traffic, in the end, a would-be pirate faces the problem of building a memory-reclamation algorithm for an application whose dynamic memory structure is not just unknown, but possibly designed to frustrate "conservative" collectors.

We now give a schematic description of how virtual leashing makes `free()` calls hidden from the client but made known to the server (a more detailed description appears below). The key idea is that as we augment and replace native memory management calls with message transmissions, we construct a table on the side that records the meaning of each message. This table is then encrypted off-line using a key known only to the server[2]. When a leashed application starts up, it sends the encrypted leashing table to the server. This encrypted table is the unique key protecting the application's security. This technique is secure because the client never sees the decrypted table. It is also scalable, because the table is small (a few thousand bytes), and because the server does not need to maintain a database of leashing tables.

Every call of the form

```
p = malloc(size)
```

is replaced by

```
p = malloc(size);
...
send(m);
```

Sometime after the `malloc()` call, the client sends a message `m` containing only the current line number and a randomly-permuted list of local variable values that includes `p` and `size`. (If the size is constant, then it can be included directly in the table, and need not be sent in the message.) When the server receives the message, its leashing table indicates that the message number reports a `malloc()` call, and indicates which arguments are relevant.

Next, every `free()` call is *replaced* by a `send(m)` call, where `m` is a message, indistinguishable from the others, containing only the current line number and a permuted list of local variable values. When the server receives the message, its leashing table indicates that the message number reports a `free()` call, and indicates which arguments are relevant.

[1] We focus here on applications that manage memory explicitly (such C and $C++$ programs). We consider garbage-collected languages such as Java in the conclusion.

[2] Either symmetric or asymmetric (public-key) encryption can be used, although asymmetric encryption protects the server against corrupt leashing programmers who might leak a symmetric key.

We can identify an important and useful class of memory allocation calls, which we call *static allocations*. A `free()` call is static if it always frees the address most recently allocated by a particular `malloc()` call. A `malloc()` call is static if all its corresponding `free()` calls are static. (As discussed in the Appendix, static calls can be detected by profiling.) The leashing table indicates which `malloc()` calls are static, and which `free()` calls correspond to that `malloc()` call. Static `free()` calls are attractive because the address being freed does not need to appear anywhere in the message.

Finally, we add *decoy* message transmissions to the program. These messages are indistinguishable from the malloc and free messages: each has a line number and a list of local variable values. Their leashing table entries instruct the server to ignore them. Decoy messages provide *steganographic* protection for the free messages, making it difficult for a pirate intent on traffic analysis to distinguish between real and decoy messages.

The application's `malloc()` calls are still present in the leashed executable, but its `free()` calls have been removed and replaced with message transmission calls. The leashed client is unable to free memory by itself, so instead, it listens for messages from the server that instruct the client which blocks of memory to free. The server tracks the client's memory usage, and releases enough memory to keep the server running. We discuss the security aspects of this arrangement in the sequel.

We have found virtual leashing a good match for electronic games (having leashed Quake I, Quake II, and others). These games tend to be large, relatively unstructured C programs, that make extensive use of dynamic memory management, and that must react quickly to user input (so more conventional kinds of software splitting are unlikely to perform well).

2 Client and Server Prototypes

Both the client and the server represent the leashed heap as a *skiplist* [7], where each list element contains a block's size and starting address. The client-side interface provides the client an allocation call (which takes a size and returns an address), and it provides the server a *release* call (which takes a starting address and a size). Not all memory management calls must be leashed.

The skiplist representation makes it easy for the client to locate the block containing an arbitrary address. For example, the client might allocate 1000 bytes starting at address 100, and the server might later instruct the client to release 900 blocks starting at location 200. In a similar way, when the client sends a *free* message, it can send any address within the block being freed, and the server knows to free the block containing that address.

To ensure that the interactions between the application and the server are asynchronous, the client runs in a thread parallel to the application's main thread (all applications we have leashed are single-threaded). The client thread and server communicate over a TCP connection. When a client creates a new connection, the server creates a new thread to handle it. The client and the server threads execute a simple handshake in which the client sends the encrypted leashing table to the client. All malloc, free, and decoy messages are 32 bytes. The first word is the offset of that message's entry in the leashing table, and the other seven are the permuted message arguments. (This size was chosen arbitrarily; longer messages with more arguments enhance security while consuming more bandwidth.)

The server keeps track of memory that the client has implicitly freed, but that has not yet been explicitly released to the client. When that amount exceeds a threshold, the server sends the client an 8-byte *release* message containing the addresses and size of the memory to be released. To conserve bandwidth, the server merges adjacent free blocks.

3 Security

Our goal is to ensure that that an adversary cannot reconstruct the missing memory management code by observing the program's run-time behavior.

3.1 Direct Attacks

Perhaps the most direct attack is simply to avoid ever having to free allocated memory. For example, run the application with enough physical memory so that it never needs to free anything. (Note that this attack is limited by the size of the physical memory, not the virtual memory, since the application will start to thrash once its working set substantially exceeds physical memory.)

In the short term, we can *churn* the application's memory usage. One simple way is to overallocate memory for short-lived objects. Another is to move stack objects into the heap, allocating them when a procedure is called, and freeing them asynchronously after the procedure returns (these allocations are static, so the corresponding messages contain no meaningful addresses).

We nevertheless discovered that with simple profiling tools, we can tune the application to consume memory at a rate that guarantees that the application will exhaust the resources on an ordinary machine quickly enough to render the adversary's experience unsatisfactory, but not so quickly that the leashing server cannot keep up. Of course, a pirate willing to pay for an extraordinary amount of memory will be able to run longer, but such piracy is expensive, and does not affect the security of the application on standard machines. Eventually, of course, falling memory prices will lower the barrier to this kind of attack. Nevertheless, the value of the protected software is falling as well. By the time memory prices have fallen enough that a pirate can afford to run the application long enough to be usable, the software itself may well no longer be worth protecting.

Another attack is to add a memory management system based on a conservative garbage collector (for example, [2]). In a leashed application one would first rip out the native memory management and then add the conservative collector. Conservative collectors, however, assume that both the programmer and the program behave themselves. The programmer should not "hide" pointers, and the program should leave around few pointers to "dead" memory. The open-source programs we leashed (discussed below) did pointer arithmetic and had pointers to the middle of data structures, and other practices incompatible with conservative collectors.

It is also a simple matter to overallocate long-lived memory blocks, and have the server later return the excess. Such a technique makes it difficult for a collector to be sure how much of a large block is actually in use.

Finally, when allocating a leashed object, it is easy to add a few pointer fields to other recently-allocated objects, ensuring that every object ever allocated remains reachable. It is also easy to displace pointers, to XOR pointers with other values, and so on. In the end, we think that any such attack would be prohibitively expensive, because each small incremental defense on the part of the leasher will require a much larger incremental response on the part of the pirate.

3.2 Attacks on Leashing

We now review ways a pirate might analyze and tamper with message traffic and contents of a leashed application. Recall that a malloc message contains the address and size of the block being allocated (although a constant-size malloc can store the size in the leashing table). A dynamic free message contains the address of the block being freed, or at least a pointer into that block. (Note however, that any such pointer will most likely also appear as a decoy argument in other messages) A static free message does not typically include the address of the block being freed (the server reconstructs that address from the encrypted leashing table and the history of malloc messages). Note that it would not be difficult to combine two or more logical messages into a single physical message.

3.2.1 Program and Traffic Analysis

The first class of attack tries to identify when memory becomes free by using a debugger to single-step through the application, eavesdropping on client/server message traffic.

Can we exploit server-to-client traffic? The server "ages" memory before releasing it, making it difficult to correlate the server-to-client release message with any prior client-to-server message. The server never releases more than a fixed percent of its free memory, to guard against an attack where the pirate pretends to be low on memory. Finally, the order in which the server releases memory is unrelated to the order in which that memory was either allocated or freed.

Recording and *replaying* the server's messages will not help a pirate because any application complex enough to be valuable will behave differently each time it is run, especially an interactive application, and there will easily be an exponential number of such possible combinations.

One naïve attack is to trace when an address is allocated, trace when it is released by the server, and to insert a `free()` call in the line of code being executed at the time the release message is received. Since the message is received asynchronously, however, this choice is no better than a guess, and there is no guarantee that the same address will actually be free the next time that particular statement is executed.

Here is the most effective attack we have been able to devise. Given two statements:

```
p = malloc(size);
...
send(m);
```

we can test the hypothesis that the message transmission replaces the call `free(p)` by inserting a `free(p)` call immediately after the message transmission, and then exhaustively testing the application. If it ever crashes, the hypothesis is wrong. If it does not crash, the hypothesis is probably correct (assuming the effectiveness of your test suite).

Naturally, testing any single message transmission is not enough. For each `malloc()`, the would-be pirate must identify *all* matching `free()` calls, because allocated memory must be freed in all possible executions along all control paths. Given m `malloc()` calls and n message transmission calls, this attack requires $m \cdot n$ exhaustive tests, a formidable barrier. For example, given an application with 100 `malloc()` calls and 300 message transmission calls, the pirates will have to run 30,000 exhaustive tests. If each test takes a half-hour, then running the tests will take about two years.

This attack, expensive as it is, does not detect *dynamic* `malloc()` calls in which the address being freed is a function of the execution. For these messages, we can insert a `malloc()` call for each of the message arguments, requiring another $O(n)$ exhaustive tests.

3.2.2 Tampering Attacks

A pirate might attempt to gain information by tampering with the client/server message traffic. For example, a pirate might omit a message containing a particular address, and then watch to see if that address is freed. If that address is not freed, then the missing message may be a free message. This kind of attack faces the same kind of computational barrier as the attacks we have already considered. In fact, tampering attacks are weaker yet, because they can often be detected. Once tampering is detected, the server can mislead the pirate. For example, the server could ignore a later `free()` message, misleading the pirate into thinking that the omitted message was one of the ignored `free()` messages. Even if the server can detect only some tampering, the pirate can never be sure whether the server's reaction to a message is real or misleading.

Some tampering can be detected easily. For example, if the client tries to free an address that was never allocated, then we can deduce that the client omitted the malloc message. We have devised other ways of introducing dependencies among messages in a way that ensures probabilistically that the server is likely to detect omitted or spurious messages. The same arguments apply to attacks in which message contents are altered.

3.3 Summary

Leashing is secure even if the leashing protocol is completely public. All that matters is the correspondence between messages and `free()` calls, a correspondence that appears only in the leashing table. The leashing table itself is encrypted with a key known only to the server, and compiled into the application . In this way, virtual leashing is scalable because servers do not need a database of leashing tables. Moreover, an application can be leashed by any server, either across the Internet or on a secure coprocessor.

Virtual leashing provides "defense in depth". Even if a pirate learns somehow that a particular message corresponds to a free statement, that knowledge does not make it any easier to locate other missing free statements in that application. Even if a pirate is able to crack one application (say, by stealing the leashing table from the developer), that knowledge does not make it any easier to crack other applications.

4 Leashed Applications

To evaluate the performance implications of virtual leashing, we leashed three sample applications from different domains: Quake II[3], a popular game, Abiword[4], a word-processing program similar to Microsoft Word, and Mozilla[5], a browser. We chose Quake II because it is an ideal target for leashing: it has complex memory usage patterns, and requires quick reactions. We included the other two (which are unlikely targets of piracy) simply to explore the performance aspects of leashing other kinds of software. All three are written in C or C++, and are available in open-source releases. Our discussion here focuses on performance and resource use. Security itself is hard to test empirically, especially when the programs at hand are open-source.

Figure 1 shows the numbers of calls expressed in terms of source code lines, and Figure 2 shows how frequently they were executed (some columns fail to add up to 100% because of rounding errors). We leashed almost all the memory calls in Abiword and Quake, but only the Javascript engine of Mozilla (a much larger program). The number and frequency of decoy messages was chosen more-or-less arbitrarily to illustrate the effects of small, medium, and large frequencies.

We tested each leashed application against a server running on the same machine (at 127.0.0.1), and against a server running on a remote workstation accessed over the Internet. The remote server was located on a 1GHz machine at Brown University, accessed through an institutional firewall. Abiword and Mozilla were tested on a 660Mhz machine at Tel-Aviv University (ping 170ms) while Quake was tested from a 400Mhz home workstation in Boston, about fifty miles away from the server in Providence (DSL line, ping 28ms). None of the applications is compute-bound. Leashing itself is not computationally demanding: profiling shows that when Abiword is actively being leashed, the leashing client consumes no more than 5% of the CPU cycles, some of which would have been consumed anyway by native memory management.

4.1 Bandwidth

	Mean	Max
Abiword local	36.5	418.3
Abiword remote	36.2	294.4
Mozilla local	30.9	262.7
Mozilla remote	30.1	260.1
Quake II local	43.3	445.2
Quake II remote	36.5	181.6

Figure 4. Bandwidth in Kbits/second

Testing each application against a local server reveals how leashing works when available bandwidth is maximized, while testing against a remote server reveals behavior when bandwidth is limited. Figure 3 contrasts the memory use for Quake II running against the local and remote servers. Figure 4 shows maximum and mean band-

[3] http://www.idsoftware.com
[4] http://www.abisource.com
[5] http://www.mozilla.org

	Static malloc	Static free	Dynamic malloc	Dynamic free	Decoys
Abiword	35	36	41	61	24
Mozilla	13	12	20	20	9
Quake II	13	17	12	11	8

Figure 1. Numbers of source statements

	Static malloc	Static free	Dynamic malloc	Dynamic free	Decoys
Abiword	1	1	33	48	14
Mozilla	3	3	44	41	9
Quake II	3	3	3	0	89

Figure 2. Percentage of run-time traffic

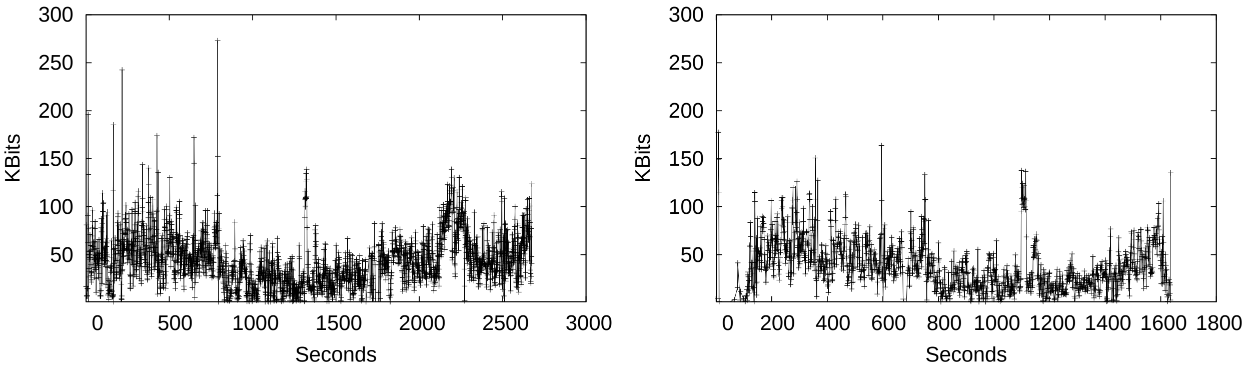

Figure 3. Bandwidth consumed by Quake II, local (left) and remote (right) servers

width consumption for local and remote servers as evaluated through the trace files. For each application, the mean bandwidth consumed is essentially the same for both the local and remote servers, while the maximum bandwidth differs substantially for Abiword and Quake. These observations suggest that the asynchronous nature of leashing allows peak bandwidth demand to be smoothed out over time. The application is not delayed when the bandwidth demanded exceeds the bandwidth available because the client runs in its own parallel thread. The application's pool of unused memory provides a cushion against the effects of message latency. In all cases, the average bandwidth demands could be met by a dial-up connection.

4.2 Memory

Clearly, leashed applications will require more memory than their unleashed counterparts. Leashing introduces a delay between when memory becomes free and when that memory becomes available for reuse. This delay shows up as increased memory use. This phenomenon becomes particularly acute during an “allocation storm”, such as loading a complex web page, or entering a new game level. An unleashed application frees a number of blocks and then allocates new blocks, resulting in a burst of activity, but little or no additional memory consumption. A leashed application, by contrast, sends a number of messages, and allocates the new memory before it can reuse the old memory, resulting in short-lived spikes in memory consumption. While bandwidth demand spikes can be smoothed over by asynchronous communication, memory consumption spikes simply require more memory.

	mean	max
Abiword local	1.39	1.42
Abiword remote	1.42	1.44
Mozilla local	1.76	2.07
Mozilla remote	1.67	2.12
Quake II local	1.97	2.09
Quake II remote	1.74	2.00

Figure 6. Memory Use Ratios

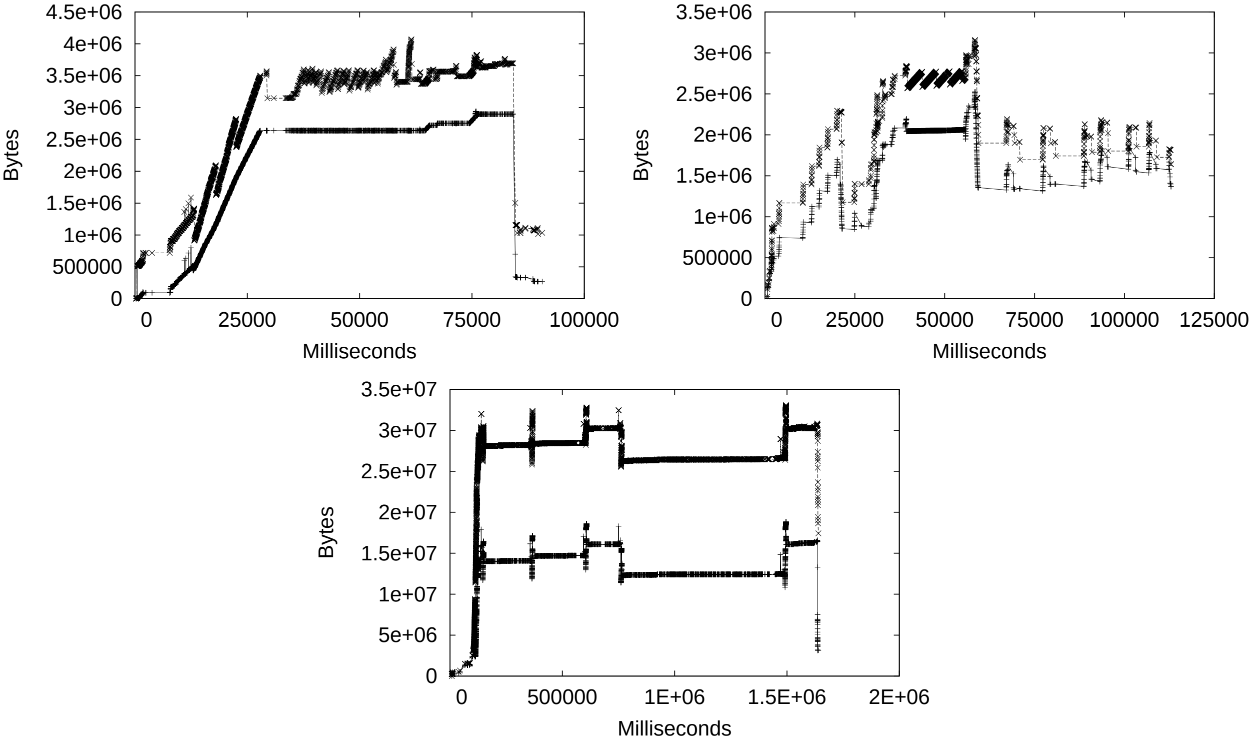

Figure 5. Memory use comparison for Abiword (top left), Mozilla (top right) , and Quake (bottom), using a remote server.

To evaluate the extra memory consumption induced by leashing, we analyzed traces of leashed applications to compute how much memory that trace *would* have allocated had it not been leashed. Specifically, we keep running leashed and unleashed memory use totals. The unleashed total is decreased immediately when a free message is sent, while the leashed total is decreased only when the server releases that memory.

Figure 5 shows the memory use curves for the three applications running against a remote server. In each case, the leashed and unleashed curves start out the same, but the server quickly establishes a distance between them.

More generally, Figure 6 displays the ratio of the maximum leashed memory allocation over the maximum unleashed memory allocation, and the mean leashed memory allocation over the mean unleashed memory allocation. Leashed Abiword requires about one and a half times as much memory, and the others need about twice as much.

Finally, Figure 7 shows the rate at which the applications allocate memory, giving a rough idea how long they would run disconnected from the server. These rates represent a modest effort to churn the memory; a more aggressive effort could drive the rates higher.

Abiword	Mozilla	Quake II
29.75MB	15.38MB	13.61MB

Figure 7. Allocation rate per minute

5 Related Work

Direct hardware support for copy protection include XOM [5] and AEGIS [10].

The most popular industrial software protection schemes are software wrappers [4, 8] and hardware-based dongles such as HASP [3]. A number of techniques to break the protection of any secured application in an automatic way, are widely available on the Web.

Zhang and Gupta [11] describe how to use compiler technology to remove short segments of code from programs. We are aware of two commercial software-splitting schemes [6, 9]. Both algorithms explicitly remove code from the application and emulate the missing instructions on the secure server. Sospita uses a specialized OS sitting on a smartcard [9], and Netquartz uses a server across the Internet [6]. All these techniques, however, are blocking: the main program must wait for a response each time it calls a remote code fragment.

6 Conclusions

The principal contribution of this paper is simply the observation that one can exploit the distinction between lazy and eager tasks to provide piracy protection in asynchronous networked environments. Soon, network connectivity will be almost ubiquitous, and smart cards and USB devices can fill in the few remaining gaps (such as airplanes and space stations).

This work raises a number of open questions. Can we exploit other resource management asymmetries, such as buffer space, file ids, and so on? How can we leash applications written in garbage-collected languages such as Java? (Perhaps by leashing the underlying virtual machine?) Can we use leashing to protect copyrighted content, such as films or songs?

References

[1] B. S. Alliance. Seventh annual BSA global software piracy study. In `http://www.bsa.org/usa/policyres-/admin/2002-06-10.130.pdf`, 2001.

[2] D. Detlefs. Garbage collection and run-time typing as a c++ library. In *C++ Conference*, pages 37–56, 1992.

[3] A. Inc. HASP3 to HASP4 – whitepaper. In `ftp://ftp.ealaddin.com/pub/hasp/-new_releases/docs/hasp-3tohasp4.pdf`, 2003.

[4] M. Kaplan. IBM cryptolopes, superdistribution and digital rights management. In `http://www.research.ibm.com/people/k-/kaplan/cryptolope-docs/crypap.html`, 2003.

[5] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[6] NetQuartz. Easyplatform 2.0 technical overview. In `http://www.netquartz.com/`, 2003.

[7] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[8] O. Sibert, D. Bernstein, and D. V. Wie. The DigiBox: A self-protecting container for information commerce. In *In Proc. 1st USENIX workshop on Electronic Commerce*, pages 171–183, 1995.

[9] Sospita. Schlumbergersema-sospita software protection. In `http://www.sospita.com/files-/SchlumbergerSema_Sospita_WP.pdf`, 2003.

[10] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. aegis: Architecture for tamper-evident and tamper-resistant processing, 2003.

[11] X. Zhang and R. Gupta. Hiding program slices for software security. In *Proceedings of the international symposium on Code generation and optimization*, pages 325–336. IEEE Computer Society, 2003.

A Appendix: How to Leash an Application

In this appendix we describe how to leash an application. We focus on the problem of retrofitting virtual leashing to an existing application for which we have the source code. Our goal is to minimize the degree to which the leashing programmer must understand the application's structure. This problem is harder than leashing an application under development (where the developers understand the application's structure in detail).

Our experience can be summarized as follows: a combination of extensive testing, tracing, and trace analysis can alleviate the need for a deep understanding of the application being leashed. Our methodology can be summarized as follows.

- Profile the application to discover how it manages memory. Identify the application's allocate and free calls, and wrap them with macros that record their activities in a trace file. This step requires little expertise, but much patience.

- Analyze the trace files (we use Perl scripts) and generate reports. These reports identify which memory management calls to leash, as well as the most effective way to leash them. This step requires expertise in leashing, but little or no specialized knowledge of the application itself.

- Add *decoy* messages to the leashed application to frustrate traffic analysis. This step, too, requires expertise in leashing (to maximize obfuscation while conserving bandwidth), but no specialized knowledge of the application.

- Before deploying, debug and optimize the results of the previous steps. This step, if needed, does require some understanding of the application. As recounted in Section 4, our experience leashing three unfamiliar applications yielded no non-trivial debugging problems, but did yield a few interesting technical and performance problems. Solving these problems did require at most one person-day to "drill down" on specific aspects of the application source code, but did not require any generalized understanding.

In our prototype, all trace file analysis, source preprocessing, and source postprocessing were accomplished by surprisingly uncomplicated Perl scripts.

A.1 Profile the Application

The first step is to understand how the application manages free storage. Replace each of the application's native memory calls, and "wrap" each one in a macro that logs each call in a trace file. Each log entry includes (1) the actual native call, (2) a timestamp, (3) the call's source file and line number, and (4) all arguments and results. Once the native memory calls are traced, run the application long enough to generate sufficiently complete trace files. This step is similar to quality control testing: care should be taken to exercise as many features and paths as possible.

Next, analyze the trace files to identify static and constant-size allocations. It is important for the trace files to be comprehensive enough to avoid identifying an allocation as static when it is not. (We found no such "false positives" in our examples.)

The Quake II application has a common special form of static allocation called *tagged* allocation. The caller provides an integer *tag* argument when allocating a block, and all blocks allocated with the same tag can be freed by a single tagged free call. Tagged allocation is useful for loading and unloading DLLs, and for entering and leaving game levels.

The trace file analysis guides leashing the application. Some memory calls should not be leashed. For example, Mozilla sometimes goes into a frenzy of allocating lots of very small strings with very short lifetimes. Clearly, there is little value to leashing such calls.

At each step, the profiling report identifies matching allocation and free calls. It is worth emphasizing that it is not uncommon for a `malloc()` call to match multiple `free()` calls, and vice-versa. When leashing an application incrementally, it is, of course, necessary to replace all matching calls in a single step. (The existence of multiple matchings also adds complexity to the analysis task facing a would-be pirate.)

For example, we can replace the following matching calls:

```
p = malloc(size);
...
free(p);
```

by something like

```
p = malloc(size);
VL_SEND_MALLOC(p, size);
...
VL_SEND_FREE(p);
```

The `VL_SEND_MALLOC` expression accompanies the allocation call, while the `VL_SEND_FREE` expression replaces the free call. These expressions are *not* function calls; instead they are expanded by a preprocessor into message transmission calls, as described below. The programmer in charge of leashing the code may provide optional *decoy* message arguments to help disguise which arguments are real. (Otherwise, decoy arguments are chosen by the preprocessor.)

As a common-sense measure, common idioms such as:

```
if (p) {
 free(p);
 p = NULL;
}
```

should be transformed to

```
VL_SEND_FREE(p);
...
p = NULL;
```

Such a transformation, while not essential, is a nuisance for an aspiring pirate. It works because the server sensibly ignores requests to free `NULL` pointers. Note that setting `p` to `NULL` does not need to happen immediately after the message transmission.

Static allocations are important, and deserve special treatment.

```
p = malloc(size);
...
free(p);
```

becomes

```
p = malloc(size);
...
VL_STATIC_MALLOC(p, size, TAG);
...
VL_STATIC_FREE(TAG);
```

Here, `TAG` is a unique string recognized by the preprocessor, not a program variable.

Static allocations have two properties that are essential for understanding the security of virtual leashing as a whole. First, at run-time, static allocations are indistinguishable from regular allocations. Second, the message that reports a static free need not (and should not) contain the address being freed. (Tagged allocations are treated essentially the same, with some minor technical differences.)

A.2 Debugging Build

In the next step, the file is preprocessed to yield two outputs: a C (or C++)-language file in which the virtual leashing expressions are replaced by tracing and message-transmission calls, and a table fragment that identifies the meaning of each call. If we compile and link the preprocessed files, the result is a running program in which some free calls have been replaced with message transmissions to the server. As part of the compilation process, the table fragments are combined, encrypted, and compiled into the application.

Although the essential leashing functionality is present, the modified application is not ready for deployment, as it lacks decoy messages and the memory consumption rate has not yet been fine-tuned. The application also generates detailed trace information describing message traffic and memory usage.

Client-to-server messages are 8 words (32 bytes) long. The first word is a message id, which is actually the offset of its entry in the application's leashing table. In our prototype, each message has seven argument slots, containing both actual and decoy arguments. The arguments are randomly permuted. Some or all of the decoy arguments may be provided explicitly by the leashing programmer, and the remaining decoy arguments are chosen at random by the preprocessor from a pool of recently-allocated addresses. Each message's type and permutation are recorded only in the encrypted leashing table, and appear nowhere in the application code itself.

A.3 Tuning

The next phase is to add decoy messages and to fine-tune the rate at which memory is allocated. This phase requires some expertise in leashing, but does not require any detailed understanding of the application.

Decoy messages make it harder to guess which messages correspond to `free()` calls. They provide a kind of "bodyguard of lies" making it difficult for pirates to separate signal from noise. Here, one can apply common-sense rules: to maximize protection, the number of decoy message calls should be at least as much as the number of free and malloc calls, and similarly for their frequency. We think that there are intriguing research problems figuring out how best to seed an application with decoy messages. For our prototypes, we just did the best we could.

A.4 Production Build

Once we are satisfied that the leashed application has a good ratio of decoy-to-free messages, that the bandwidth consumption is not too high, and the memory consumption rate is not too low, then we can build the production version simply by disabling tracing.