Available online at www.sciencedirect.com
ScienceDirect

J. Parallel Distrib. Comput. 66 (2006) 1233–1240

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Virtual Leashing: Creating a computational foundation for software protection[☆]

Ori Dvir[a], Maurice Herlihy[b,*], Nir N. Shavit[c]

[a]Computer Science Department, Tel Aviv University, Ramat Aviv 69978, Israel
[b]Computer Science Department, Brown University, Providence, RI 02912, USA
[c]Computer Science Department, Tel Aviv University, Ramat Aviv 69978, Israel

Received 17 December 2005; received in revised form 31 March 2006; accepted 10 April 2006
Communicated by Weisong Shi, Cheng-Zhong Xu, Xiabo Zhou
Available online 14 June 2006

Abstract

We introduce *Virtual Leashing*,[1] a new technique for software protection and control. The leashing process removes small fragments of code, pervasive throughout the application, and places them on a secure server. The secure server provides the missing functionality, but never the missing code. Reverse engineering the missing code, even with full tracing of the program's execution and its communication with the server, is computationally hard. Moreover, the server provides the missing functionality asynchronously: the application's performance is independent (within reason) of the secure server's speed. For example, the server might reside on a slow inexpensive chip or a remote Internet server. Leashing makes only modest demands on communication bandwidth, space, and computation.

Keywords: Digital rights management; Virtual leashing

1. Introduction

Software piracy is an enormous economic problem, reaching by some estimates a worldwide level of 40% in 2001 [2]. 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. These technologies offer protection against only casual piracy. Software must be unwrapped before it can be executed, and can then be captured, and cryptographic tests can be removed using widely available tools. These technologies fail to provide software with the same degree of protection (security based on computational hardness), that cryptography provides for communication.

A conceptually simple and appealing way to introduce "computational hardness" into software protection is *software-splitting*: 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. One obstacle is how to identify functionality that cannot easily be reverse-engineered. Another potential show-stopper is *communication latency*. Suppose the server is running on a secure coprocessor. Any coprocessor is 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. Moreover,

[☆] A preliminary version of this paper appeared in the 25th International Conference on Distributed Computing Systems (ICDCS, 2005), 6–10 June 2005, Columbus, OH, USA.

[*] Corresponding author.

E-mail addresses: oridvir@hotmail.com (Ori Dvir), herlihy@cs.brown.edu (M. Herlihy), shanir@cs.tau.ac.il (N.N. Shavit).

[1] The techniques described in this paper are protected by U.S. patents, both granted and pending.

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.04.013

for the foreseeable future secure coprocessors will have limited memory, so programs and data will have to be swapped in and out of memory, further increasing communication delay and uncertainty. Similar problems arise if the server is running across a network, since network delays can be long and unpredictable. Either way, it is 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, where any perceptible delay will be unacceptable to users.

In this paper, we propose the first technique that makes software-splitting a practical and effective defense against software piracy. We claim that this technique, called *virtual leashing*, is both efficient and secure in the computational sense.

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 the 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 platform, either off-site or on a secure coprocessor.

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 asynchrony 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.

How do we propose to solve this problem? Two words: `malloc()` and `free()`. 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.

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 actively 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 technique is secure because the client never sees the decrypted table. It is also scalable, because the server does not need to maintain a database of all leashing tables, only the ones in use at that moment.

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 the current line number and a randomly permuted list of local variable values that includes `p` and `size`. A leashing table entry instructs the server that each message with the given line number reports a `malloc()` call, and also indicates for certain types of frees which permuted arguments correspond to the allocated address and size. Next, every `free()` call is *replaced* by a `send(m)` call, where `m` is a message indistinguishable from the one before, containing the current line number and a permuted list of local variable values. A Leashing Table entry instructs the server that each message with the given line number replaces a `free()` call, and also indicates which permuted argument is the freed address.

Next, we add lots of *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 computationally

[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.

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 client running. We discuss the security aspects of this arrangement in the sequel.

2. How to leash an application

In this section 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 suggests that a combination of extensive testing, tracing, and trace analysis can alleviate the need for a deep understanding of the application being leashed.

First, 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.

Second, 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.

Third, 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.

Finally, before deploying, debug and optimize the results of the previous steps. This step, if needed, does require some understanding of the application. As recounted below in Section 5, our experience leashing three unfamiliar applications yielded no non-trivial debugging problems, but did yield a few interesting technical and performance problems. In our prototype, all trace file analysis, source preprocessing, and source postprocessing were accomplished by surprisingly uncomplicated Perl scripts.

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. Run the application long enough to generate sufficiently complete trace files.

Next, analyze the trace files to identify matching `malloc()` and `free()` calls. It is common for a `malloc()` call to match multiple `free()` calls, and vice-versa. 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.)

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. Static allocations are indistinguishable at run-time from regular allocations, but the message that reports a static free need not contain the address being freed.

```
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.

The file is then 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.

In our prototype, 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. The remaining seven argument slots contain both actual and decoy arguments, randomly permuted. Each message's type and permutation are recorded only in the encrypted leashing table, and appear nowhere in the application code itself.

The next phase is to add decoy messages and to fine-tune the rate at which memory is allocated. 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.

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.

3. Client and server prototypes

Both the client and the server represent the leashed heap as a *skiplist* [8], 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 calls are leashed. The client can allocate and free non-leashed memory in the usual way.

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. (While a multithreaded server works well for our prototype, an industrial-scale server might have to be single-threaded.)

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.

4. Computational basis for security

We cannot provide a mathematical proof that Virtual Leashing is secure. Instead, we make our case by inexact analogy to the RSA [9] encryption protocol. RSA is based on a Hard Problem (factoring) widely believed to be intractable. It requires hard instances, choosing "good" keys and avoiding "bad" keys. The algorithm is public: security depends on a particular secret key, not the algorithm, which is known to all. Finally, no algorithm, however clever, is immune to a *protocol attack* that exploits an environment that "leaks" information [5].

The heart of Virtual Leashing is the following *memory reclamation problem* (MRP).

> Given a program that allocates but never frees memory and the ability to run, trace, and modify it on a machine with unbounded memory, devise an equivalent program (in terms of output and performance) that runs on a machine with only bounded memory.

We summarize our security claims for Virtual Leashing as follows.

First, we believe that the MRP is inherently hard (at the very least in the practical sense) although we do not know how to prove this claim. Second, it requires hard instances, choosing a program for which the MRP does not have a trivial solution (for example, programs for which "conservative" garbage-collectors are ineffective). Third, the leashing algorithm itself is public (to anyone who reads this paper), but a leashed application's security relies entirely on keeping the application-specific leashing table private. Finally, we offer pragmatic arguments why the client/server message protocol does not leak information to pirates.

To solve MRP, one could simply avoid ever having to free allocated memory. For example, run the application with enough physical memory so that it never needs to free anything. (This attack is limited by the size of the physical memory, not the virtual memory, since the application will 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).

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, 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 way to solve MRP is to add a memory management system based on a conservative garbage collector (for example, [4]). 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. It is easy to violate both assumptions, frequently and deviously, even without understanding the application in detail. 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.

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 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.

One 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. 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 is thus no better off than with MRP.

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. 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. Both attacks are complicated by the observation that it is not uncommon for a `malloc()` call to match multiple `free()` calls, and vice-versa, so the pirate cannot rest after finding one putative match.

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.

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.

5. Leashed applications

To evaluate the performance implications of Virtual Leashing, we leashed three sample applications from different domains:[3] Quake II, a popular game, Abiword, a word-processing program similar to Microsoft Word, and Mozilla, a browser. 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.

Fig. 1 shows the numbers of calls expressed in terms of source code lines. We leashed almost all the memory calls in Abiword and Quake, but only the Javascript engine of Mozilla (a much larger program).

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 1 GHz machine at Brown University in the Eastern United States. Abiword and Mozilla were tested on a 660 MHz machine in Israel (ping 170 ms) while Quake was tested from a 400 MHz home workstation in Boston (ping 28 ms). None of the applications is compute-bound. Leashing

[3] www.idsoftware.com, www.abisource.com, and www.mozilla.org.

	Static malloc	Static free	Dynamic malloc	Dynamic free	Decoys
Abiword	35 (1%)	36 (1%)	41 (33%)	61 (48%)	24 (14%)
Mozilla	13 (3%)	12 (3%)	20 (44%)	20 (41%)	9 (9%)
Quake II	13 (3%)	17 (3%)	12 (3%)	11 (0%)	8 (89%)

Fig. 1. Numbers of source statements and percentage of run-time traffic.

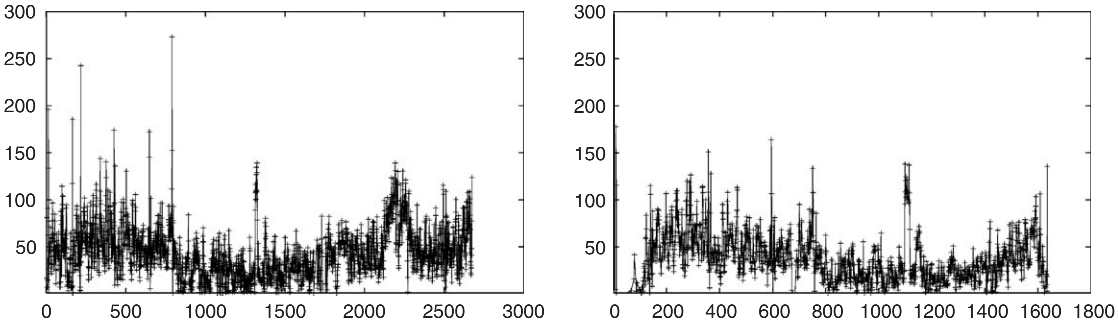

Fig. 2. Bandwidth (Kbits) consumed by Quake II, local (left) and remote (right) servers plotted against time in seconds.

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

Fig. 3. Bandwidth in Kbits/s.

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.

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. Fig. 2 contrasts the memory use for Quake II running against the local and remote servers. Fig. 3 shows maximum and mean bandwidth 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.

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, which becomes particularly visible 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.

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 4 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, Fig. 5 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, Fig. 6 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.

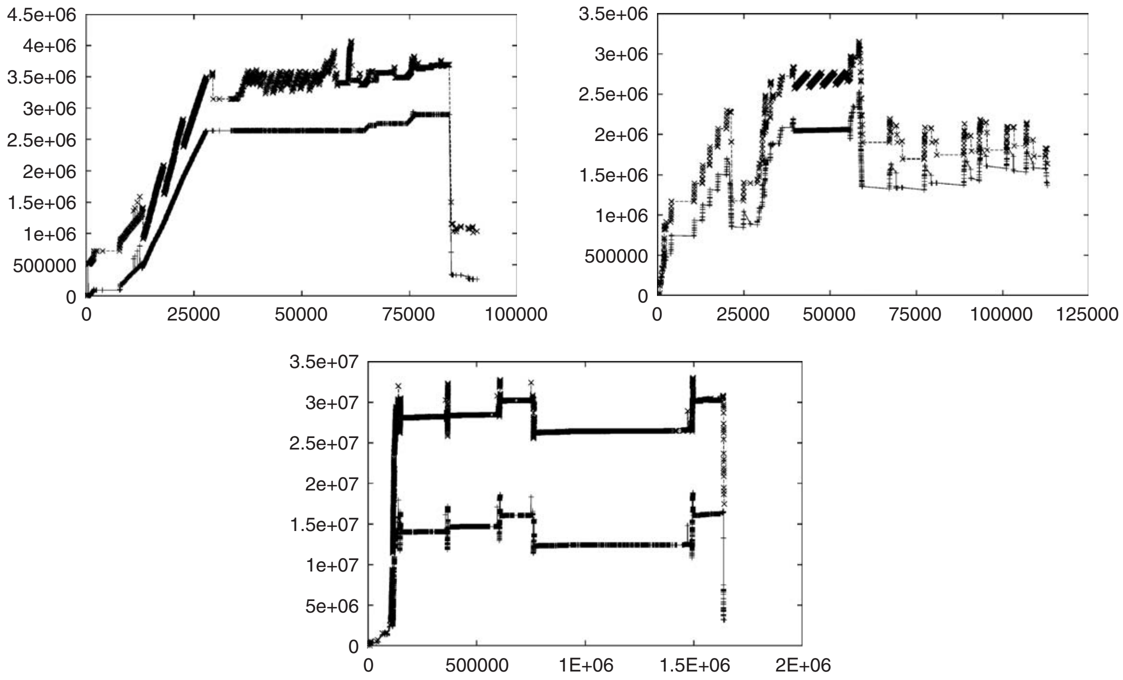

Fig. 4. Memory use comparison for Abiword, Mozilla, and Quake, remote server (higher curve is leashed, lower unleashed).

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

Fig. 5. Memory use ratios (leashed/unleashed).

Abiword	Mozilla	Quake II
29.75MB	15.38MB	13.61MB

Fig. 6. Allocation rate per minute.

6. Related work

The most popular industrial software protection schemes are software wrappers [6,10] and hardware-based dongles such as HASP [1]. Techniques to break this protection are widely available on the web (for example, see [3]).

We are aware of two prior software-splitting schemes [7,11] that explicitly remove code from the application and emulate the missing instructions on the secure server. These approaches are not asynchronous: the client application blocks waiting for a response from the server. The removed code is not pervasive, nor is there any evidence it is inherently difficult to reconstruct. The remote emulator is platform-dependent.

References

[1] Aladdin Inc., HASP3 to HASP4—whitepaper, in: ⟨ftp://ftp.ealaddin.com/pub/hasp/new_releases/docs/hasp-3tohasp4.pdf⟩, 2003.

[2] Business Software Alliance, Seventh annual BSA global software piracy study, in: ⟨http://www.bsa.org/usa/policyres/admin/2002-06-10.130.pdf⟩, 2001.

[3] Crackmain, Dongle reverse engineering, in: ⟨http://www.geocities.com/SiliconValley/Ridge/1237/crackmain.html⟩, 2003.

[4] D. Detlefs, Garbage collection and run-time typing as a c++ library, in: C++ Conference, 1992, pp. 37–56.

[5] M. Joye, J.-J. Quisquater, Cryptanalysis of rsa-type cryptosystems: a visit, DIMACS Series in Discrete Mathemematics and Thoretical Computer Science, 1998.

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

[7] NetQuartz, Easyplatform 2.0 technical overview, in: ⟨http://www.netquartz.com/⟩, 2003.

[8] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, in: Workshop on Algorithms and Data Structures, 1989, pp. 437–449.

[9] R.L. Rivest, A. Shamir, L.M. Adelman, A method for obtaining digital signatures and public-key cryptosystems, Technical Report MIT/LCS/TM-82, MIT Press, Cambridge, MA, 1977.

[10] O. Sibert, D. Bernstein, D. Van Wie, The DigiBox: a self-protecting container for information commerce, in: Proceedings of the first USENIX Workshop on Electronic Commerce, 1995, pp. 171–183.

[11] Sospita, Schlumbergersema-sospita software protection, in: ⟨http://www.sospita.com/files/SchlumbergerSema_Sospita_WP.pdf⟩, 2003.

Ori Dvir received his B.A. from the Ben Gurion University and his M.Sc. from the Tel Aviv University, both in Computer Science. He has worked for various software companies in the security area. He is now leading a development team in Biosense Webster Inc. His areas of interest are: software protection, compilation, and cryptography.

Maurice Herlihy's research interests focus around practical and theoretical aspects of designing, implementing, and reasoning about concurrent and distributed systems. He has his A.B. in Mathematics from the Harvard University and his Ph.D. in Computer Science from the MIT. He has been a faculty member in the Computer Science Department at the Carnegie Mellon University, and a member of research staff at Digital Equipment Corporation's Cambridge Research Lab. In 1994, he joined the Computer Science Deptartment at the Brown University. He has been awarded the 2004 Goedel Prize for outstanding journal articles in theoretical Computer Science, and the 2003 Dijkstra Prize in Distributed Computing. He is a Fellow of the ACM.

Nir Shavit received his B.A. and M.Sc. from the Technion and his Ph.D. from the Hebrew University, all in Computer Science. He was a Postdoctoral Researcher at the IBM Almaden Research Center, Stanford University, and MIT, and a Visiting Professor at the MIT. He joined the Computer Science Department at the Tel-Aviv University in 1992 and became a Member of Technical Staff at Sun Microsystems Laboratories in 2003. Dr. Shavit is the recipient of the Israeli Industry Research Prize in 1993 and the ACM/EATCS Godel Prize in Theoretical Computer Science in 2004. His research interests include software aspects of Multiprocessor Synchronization, the design and implementation of Concurrent Data-Structures, and the Theoretical Foundations of Asynchronous Computability.