Scrash: A Tool for Generating Secure Crash Information

Pete Broadwell Matt Harren Naveen Sastry
{pbwell, matth, nks}@cs.berkeley.edu

December 16, 2002

Abstract

A growing number of contemporary applications and operating systems include provisions for sending debugging information back to the developer after a crash. While this practice is of great help to the developer, it can pose a privacy vulnerability to the end user of the software. Crash reports may contain sensitive user data such as passwords and credit card numbers, which are exposed to misuse or interception when the report is sent over the network and later stored in the developer's crash data repository.

This paper presents Scrash, a tool that safeguards user information by removing sensitive data from crash reports. Scrash operates by modifying the source code of C programs to ensure that data labeled "sensitive" does not appear in a crash report. In evaluation tests, Scrash added only a small amount of run-time overhead and required minimal involvement on the part of the developer.

1 Introduction

Examining a process' state is one of the primary resources that developers use to diagnose and fix errors in computer programs. For this reason, operating systems and programming suites have for decades included tools to capture a process' state at crash time. The file that holds this state is known as a *core file*. Recently, however, the advent of ubiquitous network connectivity for the personal computer has given developers the ability to receive information about bugs in their programs *after* they have been distributed to users.

The remote crash reporting tools incorporated into modern operating systems and applications typically send the developer a subset of the data present in a core file: part or all of the call stack, register contents, or system configuration information. While it is rare for a crash report to be more detailed than this, the largest reports may also contain the contents of the heap and any files that the program may have been accessing during the crash. We will call the information that is sent back *crash data*. Note that what comprises crash data varies from vendor to vendor. Each may decide to send a different subset of the core file.

Remote crash reporting technology grants the developer access to potentially vast amounts of crash data, speeding the diagnosis and repair of software vulnerabilities. Fingerprinting the call stack after a crash can help the developer to focus on fixing the bugs that appear most often. Alternatively, the developer can suggest fixes or patches to the user based on the call stack fingerprint. Indeed, post-deployment crash reporting and bug fix releases are quickly becoming a necessity for modern consumer software, given the increasing complexity and hurried development cycle of applications and operating systems.

There are many forms of "sensitive" information worthy of protection. Clearly a password should be considered private, and hence sensitive. But is the list of recently used file names sensitive? What about the action the user is currently executing? For this paper, we will largely sidestep this issue. We will only allow the sensitive tag to be applied to memory addresses, so that the list of function names called will *not* be considered sensitive. Furthermore, we will defer judgment of what should be considered sensitive memory addresses to the software developer. We will provide a discussion of information flow in Section 4 which impacts what should be considered sensitive.

The benefits of mobile crash data do not come without risks, however. Because they contain some or all of the memory contents of the program at the time it failed, crash reports may include sensitive user information. A recent security flaw in the Dr. Watson crash report tool for Windows NT and 2000 was due to the fact that the program would write comprehensive crash reports, including the memory contents of a program, to a world-readable directory on the computer [14]. This practice raised security and privacy concerns, because a malicious party on a multi-user system could examine the crash report with a text editor and extract possibly confidential information, such as the credit card numbers or web browser cookies of targeted user.

In addition, there are inherent security risks associ-

ated with sending crash data to a remote party over a network. For instance, it is possible for the data to be intercepted en route, although most bug-report programs attempt to guard against this by encrypting the data stream. Another, possibly greater concern deals with the fate of the data after it reaches the developer. It is likely that the data will be stored for some time, perhaps indefinitely, in a crash data repository maintained by the code developer. If it becomes known that a certain fatal error results in sensitive, valuable user data being stored in these repositories, they could quickly become attractive targets for both recreational and professional information thieves.

Finally, it is important to consider that the intentions of the application maintainer may not be entirely honorable. There is always a risk that the personal data contained in crash reports may be used to the advantage of the person or corporation controlling it, but to the detriment of the end user's privacy and security. For example, the developer may sell the private information to a third party for profit, or may use it to deliver unsolicited, targeted advertising to the user. Our system does not address the problem of deliberate privacy invasion by the developer, in part because even if we blocked crash reports completely there would still be no way to prevent other kinds of covert channels. Instead, we assume that the developer cooperates with us and examine how secure we can make crash reports in this case.

A simple solution would be for the user to disable all crash reporting, but this action would deprive the developer of much information that ultimately results in more secure and reliable software. Instead, we try to balance the security concerns of the user with the debugging needs of the developer. We have developed Scrash, a transformation on C code that removes sensitive information from crash reports, while still retaining significant non-user-specific debugging information to help the developer find and fix bugs.

Scrash works by requiring the developer to annotate data fields that will hold sensitive user data. It then performs a static analysis of the program's source code to identify any other variables that may contain this data at some point in the program's execution. Finally, Scrash inserts code into the program to ensure that sensitive user information is stored separately from other data and can easily be removed from a crash report.

2 Implementation

There are a number of way to prevent transmitting the sensitive information from the core file to the developer. Among these methods, there is often a tradeoff between the amount of utility that the core file presents and the privacy that the user achieves. The most secure method, of course, is to prevent the transmission of the core file at all. This guarantees that no information, whether sensitive or not, is leaked via the core file but deprives the developer of any useful crash information. At the other end of the tradeoff, the crash reporting tool transmits the entire core file. The developer gains all of the utility of the core file, but without any privacy protection for the user.

The current tools do not transmit the heap, so sensitive information which resides there will not be vulnerable; this leaves sensitive information in globals and on the stack vulnerable to attacks. In addition, the heap is not available for the developer's use. It should be noted that the heap is omitted for other reasons as well, as it is often significantly larger than the other segments of the core file and less helpful than the stack in the developer's task of figuring out what went wrong at the crash site.

The approach that we take seeks to eliminate sensitive information from the heap, stack, and global variables while still providing useful information to the developer. We place the contents of any sensitive variables in a separate region which is not transmitted on a crash. Thus, the stack, globals and main heap will *only* contain insensitive information, so that the crash reporting tool is free to transmit any of them. The key difficulty is identifying the sensitive data, which we will outline below.

We have implemented Scrash using very little new code. We have written 600 lines of Objective Caml code to perform the transformations, and 200 lines of C code to create a modified allocator.

2.1 Merging of source files

We use CIL (a C Intermediate Language implemented in OCaml)[7] as the infrastructure for our source-to-source translation. CIL translates C code into a clean, easy to manipulate subset of C. It includes drivers that act as drop-in replacements for `gcc`, `ar`, and `ld` so that CIL can be used with existing makefiles. CIL uses these drivers to collect all of the source files for a program, preprocess them, and merge them into a single C file to facilitate whole-program analysis.

2.2 Analyzing the sensitivity of variables

Our system extends each type in the program with a *type qualifier* to indicate whether or not it may hold sensitive information. We use CQual, a type qualifier inference program, to determine which types should have this "$sensitive" qualifier [3]. The implementation of

CQual used by our system performs a whole program, flow-insensitive analysis with a limited form of polymorphism which only works for library calls. [1] to determine where sensitive data might spread from an initial set of sensitive variables annotated by the programmer. The question of whether data may be sensitive is analogous to the question of whether it may be tainted, so we can use the same analysis as in [10].

As an alternative to annotating specific data at the point it enters the program, the programmer may choose to use a pre-annotated header file that marks as sensitive all data returned by functions like `read` and `recv`. At the cost of unnecessarily marking some values as sensitive, this option makes it easy to denote user data as sensitive without the need to enter program-specific annotations. This is the approach taken in our experiments.

The CQual stage outputs the original program and applies attributes to each variable describing its sensitivity. This allows later stages to quickly determine whether a variable should reside in the secure or insecure region of memory.

2.3 Smalloc & Shadow Stacks

After identifying the sensitive memory addresses, it becomes possible to erase their contents before shipping the core file. A difficulty arises in determining where the information resides in the core file. In general, the sensitive variables will be scattered throughout the entire core file. One method to find such variables would be to append each sensitive variable with an immutable tag identifying the sensitivity status. A post process cleaning process could then iterate over the core file and remove or overwrite all sensitive variables by looking at the tag. An alternative, which we utilize, groups sensitive memory together using a separate region.

We have written Smalloc, an allocator which is region aware, to manage this "secure" region. It is based on the Vmalloc package which provides an ideal platform to create allocators[11]. The interface to Smalloc is similar to `malloc`. We add an extra parameter to the allocation function to identify which region the new memory should come from; the reallocate and free functions remain unchanged. See Figure 3 for the complete Smalloc interface.

We use `smalloc` for all heap allocated variables as well as globals. While in principle stack allocated variables could also use this sensitive heap, we found that the performance penalty of this is significant. Instead, we use a shadow stack which resides within the secure region to hold the sensitive variables.

[1] We expect to be able to use a fully polymorphic inference soon.

```
#include <crypt.h>
int $sensitive private[2] = {0, 1};

void getPassword(char cryptpw[14]) {
  char $sensitive * password = malloc (255);
  memcpy (cryptpw,
          crypt (password, "00"), 14);
}

void check() {
  char $sensitive cryptpw[14];
  getPassword(cryptpw);
}
```

Figure 1: The original, annotated program. It contains a sensitive global, a pointer to sensitive data, and a sensitive stack variable.

```
struct check_shadow {
   char cryptpw[14] ;
};
struct __smalloc_globals {
   int private[2] ;
};
struct __smalloc_globals *__smalloc_global_var ;
void ( __attribute__((__constructor__))
       __smalloc_global_init)() ;
void __smalloc_global_init(void) {
  {
    __smalloc_global_var = (struct __smalloc_globals *)
       smalloc(sizeof(struct __smalloc_globals ), 1);
    __smalloc_global_var->private[0] = (int )0;
    __smalloc_global_var->private[1] = (int )1;
  }
}
char *stackPointer  =    0;
void getPassword(char *cryptpw ) {
  char *password ;
  char *tmp ;
  char *tmp___0 ;
  {
    tmp = (char *)smalloc(255, 1);
    password = tmp;
    tmp___0 = crypt(password, (char const *)"00");
    memcpy(cryptpw, tmp___0, 14);
    return;
  }
}
void check(void) {
  struct check_shadow *check_shadow ;
  {
    check_shadow =
      (struct check_shadow *)stackPointer;
    stackPointer += sizeof(struct check_shadow );
    getPassword((char *)(check_shadow->cryptpw));
    {
      stackPointer -= sizeof(struct check_shadow );
      return;
    }
  }
}
```

Figure 2: The results of the program transformations on Figure 1

```
void * smalloc (size_t size, char secure);
void * scmalloc (size_t nmemb, size_t size, char secure);
void sfree (void * ptr);
void * srealloc (void * ptr, size_t size);
```

Figure 3: The Smalloc allocator interface. The allocation functions take an extra parameter which specifies whether the data should be allocated in the sensitive region or on the insecure heap.

2.4 Transformations

We use the Smalloc library and the qualifiers derived by CQual to drive the program transformations. CIL provides an easy platform to perform each of these transformations, which we outline below. The results of applying the complete set of transformations to the program in Figure 1 can be seen in Figure 2.

2.4.1 Sensitive Heap Variables

A sensitive heap variable is easily identified since it contains the $sensitive qualifier and is allocated with a `malloc` call. Unsensitive heap variables also are allocated with a `malloc` call, but do not contain the $sensitive qualifier. We change each of the calls to use the Smalloc allocator, using the presence of the $sensitive attribute to control which region the `smalloc` uses. We similarly replace `calloc` with `scalloc`.

In addition to replacing the allocation functions, we also need to replace any instances of `free` and `realloc` with the `smalloc` equivalents: `sfree` and `srealloc`. These functions have the same signatures as the functions they replace, so we can perform a simple substitution.

2.4.2 Sensitive Stack Variables

There are two transformations which can be applied to move sensitive stack variables. We will outline both, reserving the discussion of each until the evaluation section.

The first transformation moves the sensitive stack variables into the secure heap. The variables are allocated at function entry and then deallocated before the return. We then rewrite all lvalues that refer to the reallocated stack variable. However, this requires adding a `smalloc` and a `sfree` call to many function bodies.

An alternative to the previous is to use a shadow stack. This is a separate stack that parallels the normal stack and holds sensitive variables. This shadow stack resides within the secure region, so that we maintain the invariant that all sensitive information is contained within that region. We adjust the shadow stack pointer at the entry and exit for the function.

2.4.3 Sensitive Global Variables

Finally, for sensitive global variables, we define a new struct to contain each of the sensitive global variables, instantiating it as `__smalloc_global_var`. We allocate it on the heap with a special initialization function. By adding the gcc-specific attribute "constructor", we can ensure that this function runs before `main`. In this function we also perform any initialization that is needed for each global variable by expanding its initializer clause into regular C statements.

2.5 Postprocessing: Cleaning

Using the above transformations, all of the sensitive information is fully contained within the secure memory region. Note that if the program crashes, the core file will still contain the sensitive information. We use a cleaning process which overwrites the secure memory region. It searches for a magic tag that identifies the metadata for the secure region. The metadata encodes the type and size of the region, allowing the cleaning process to overwrite it.

One could imagine incorporating this functionality into the operating system where the core file is produced. This would ensure that the cleaning process is always run before the crash report is written to disk and prevent problems such as the Dr. Watson bug mentioned in the introduction.

3 Evaluation

We tested our system using the OpenSSH secure shell client [8]. The program consists of about 59,000 lines of preprocessed C code. In this application it is necessary to treat all data typed by the user as sensitive. The password used to set up the connection is the most obvious security risk, but even after the connection is established the user may send passwords and other sensitive information to the server. Therefore, we use the prespecified annotations mentioned in Section 2.2, which marks all data returned by `read` (among other functions) as sensitive. This marks approximately 16% of the variables as $sensitive.

3.1 Security evaluation

We ran our modified version of `ssh` to verify that sensitive information was placed only in the secure region and that the cleaning process can properly eliminate the data. Figure 4 shows the excerpts from three core files

```
core.normal.dirty:

000732e0: 6d80 0608 7fd0 0708 0cf3 ffbf 0004 0000  m...............
000732f0: 0200 0000 34f7 ffbf e854 0908 98f8 ffbf  ....4....T......
00073300: 6842 0908 1800 0000 a066 2440 6162 7261  hB.......f$@abra
00073310: 6361 6461 6272 6100 5842 0908 f058 0908  cadabra.XB...X..
00073320: c830 0840 c4ef 0f40 7c3b 0908 28f4 ffbf  .0.@...@|;..(...
00073330: 28f4 ffbf 5842 0908 0000 0000 8855 0908  (...XB.......U..

core.smalloc.dirty:

0006a330: 70d0 2340 0000 0000 0000 0000 0000 0000  p.#@............
0006a340: 70d0 2340 0904 0000 0100 0000 0df0 edfe  p.#@............
0006a350: 6162 7261 6361 6461 6272 6100 0000 0000  abracadabra.....
0006a360: 80d3 2340 0000 0000 70d0 2340 0000 0000  ..#@....p.#@....
0006a370: 70d0 2340 0000 0000 90d3 2340 0000 0000  p.#@......#@....

core.smalloc.clean:

0006a330: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a340: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a350: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a360: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
0006a370: 5858 5858 5858 5858 5858 5858 5858 5858  XXXXXXXXXXXXXXXX
```

Figure 4: Excerpts from the core file of an induced crash in the `ssh` client. The top core file excerpt shows the stack with the password present – "abracadabra" from an unmodified `ssh` client. The middle core file is from a version of `ssh` which has been modified using the Scrash transformations and annotations. The password now resides in the secure region, but since the cleaning process has not yet been executed on the core file, the password is again present. The bottom core file shows that the cleaner overwrites the secure region, and all occurrences of the password have been removed.

where the program was induced to crash. The top core file is the original version of `ssh` where the password lives on the stack. The middle core files is the result of running `ssh` after applying the Scrash transformations, in which the password resides in the secure heap. The final excerpt shows the result after running the cleaner.

It is thus simple to validate that the password was properly moved off of the stack and into the secure region.

3.2 Performance

Figure 5 shows the performance of `ssh` after being instrumented with our system. The tests consist of connecting to a server on `localhost` and sending 2000 commands to that server. We recorded the user mode portion of the output of the `time` command on a Linux machine.

Version	Running time (s)
Baseline (without Scrash)	0.420
Sensitive locals moved to heap	0.568
Sensitive locals moved to shadow stack	0.440

Figure 5: Time needed for the OpenSSH client to read 2000 lines of commands from stdin and transmit them to the server. Numbers shown are the mean of 10 runs.

Our first strategy for moving sensitive stack variables to the heap – a call to `smalloc` at the beginning of each applicable function, as described in Section 2.4.2 – resulted in too large a performance penalty. An implementation using a shadow stack, however, added only about 5% overhead on this test due to the cost of maintaining the second stack pointer.

4 Discussion

In addition to the runtime overhead for Scrash, the system requires some effort from the programmer. This includes annotating an initial set of sensitive variables (or deciding to use the pre-annotated file). A small handful of code changes were required for `ssh` before CIL would accept it, such as fixing missing or mismatched variable declarations. This is because CIL is more restrictive in typechecking that `gcc`. It takes roughly three minutes to run the entire Scrash transformations on `ssh` from preprocessing through program modification using a 1.5 Ghz Pentium.

We must be a bit careful in evaluating the success of a technique like Scrash. For example, the absence of the password from the core file does not mean that there is no sensitive information related to the password in the core file. For example, the length of the password may be stored in a separate variable. For complete security, the taint analysis must also mark the length field as being tainted. Failure to do so would reveal the password's length.

Alternatively, it may be possible to ascertain the size of a sensitive buffer by comparing pointers. For example, if we let p be a pointer to a sensitive data field, we can bound the size of the sensitive data by comparing all heap-allocated pointers t to its pointer:

$$\min_{t>p}(t-p)$$

Thus, with a simple analysis, it may be possible to reveal the length of the sensitive buffer. This suggests that p is also sensitive and should be placed on the sensitive heap, adding an extra level of indirection to all accesses to p.

In addition, the instruction pointer and call stack can also leak information in subtle ways. They can reveal the state of conditional expressions that may depend on a sensitive value, such as whether the password length is zero or not. Similarly, the contents of the CPU registers may leak security-related information. The register

contents form an integral component of a program's debugging information, however, and removing them from a core file will greatly reduce the utility of the remaining crash data.

As we have demonstrated, the tradeoffs between user privacy and utility to the developer become difficult to manage when dealing with information flow and covert channels. By its nature, our taint analysis must be conservative in order to be effective, yet an overly conservative set of flow analysis rules quickly results in all data being forced into the secure region. This will not leak any data, but obviously is not very useful to the developer. Thus, the precision and sophistication of the taint analysis directly affects the amount of useful information the developer receives.

A final problem involves the use of precompiled and dynamic (shared) libraries. Current libraries, such as `glibc`, are written without consideration of the concept of sensitive data. CQual understands the semantics of many `glibc` functions and will correctly propagate qualifiers across, for example, calls to `memcpy`. However, there is no way for a source-level translation like Scrash to modify the storage of variables in precompiled libraries. For example, `strcpy` may keep a char temporarily on the stack or, more likely, in a register; `strlen` may keep a running string length count as a stack variable. In the event of a crash, these variables will remain on the insecure stack, where they can leak pieces of sensitive information. This leads us to conclude that sensitive data should never be passed to a precompiled library function, in order to preserve the sensitivity semantics.

There are a number of possible solutions to this problem. One could choose from a set of precompiled libraries with different versions of the given function: one in which the first argument is considered sensitive but not the second, one in which only the second argument is considered sensitive but not the others, and so on. For an n argument function, this could require up to 2^n different versions. Alternatively, the function could be written assuming that all the arguments are considered sensitive. A third solution involves abandoning precompiled libraries altogether and compiling shared functions directly into the program, using CQual to propagate taint analysis qualifiers. This last method, unfortunately, deprives end users of the benefits offered by shared libraries.

5 Related Work

To the best of our knowledge, there has been no previous research published on the topic of limiting crash data to ensure privacy. Microsoft's Dr. Watson [2] and the Bug-Buddy bug reporting tool for Gnome [1] represent the current state of the art in remote crash reporting software. Both tools can be used to transmit the function call stack as part of a bug report. Dr. Watson encrypts this data for transmission using SSL, and also keeps much more detailed crash information, such as the program heap, on the local machine.

Liblit and Aiken look to automate part of the debugging process with partial information[6]. They describe a technique to automatically reconstruct program paths given only a limited set of information, for example just the instruction pointer and stack backtrace. Their work is mainly focused on a better analysis and does not address security.

There is a large body of work which describes techniques for efficient allocators [13] and garbage collectors [12]. Region based memory allocators in which multiple heaps are exposed have also been studied [4, 5]. While they present a richer set of semantics than we need, they helped to inspire our implementation. Finally, the Vmalloc software release provides an alternative allocator to `malloc`. It exposes many different allocation fit strategies and exposes rich internal interfaces.

We use CQual, a static analysis tool, to track the possible spread of sensitive information [3]. Sabelfeld and Myers[9] survey language-based systems for statically tracking information flow in a secure manner. This typically involves removing all covert channels within a program which can require extensive code modifications. We do not address the issue of convert channels in this work.

6 Future Work

Changes to Scrash in the short term mostly involve improvements to the analysis phase. We hope to use an improved CQual which reduces the number of false positive sensitive type qualifier tags. This new polymorphic version of CQual should easily integrate into the Scrash tool without any effort. Modifying Scrash to work with C++ is another area of active interest; CQual has recently been extended to work with C++ code.

In addition, we hope that support for Scrash will be incorporated into some of the standard bug reporting tools. For the greatest degree of security, however, it will be necessary to integrate Scrash with the routines in the operating system that actually produce core files, so that sensitive data will already be zeroed out of a core file before it is written to the file system.

7 Conclusion

We have described Scrash, a C code modification tool for generating secure crash information. The tool is developer friendly and introduces very low run-time overhead. It is our hope that the tool will soon become widely adopted, so that developers may continue to take advantage of the opportunities offered by remote crash reports, while at the same time safeguarding the end user's security and privacy.

8 Acknowledgments

Many people have contributed to this project. Dan Wilkerson and Rob Johnson implemented many last minute CQual features for us, while John Kodumal, Jeff Foster, and the rest of the CQual team provided advice on using CQual. We thank Ben Liblit and David Gay for their insightful comments and suggestions. The CIL project has proved instrumental for implementing our transformation. Finally, David Wagner provided helpful guidance along the way.

References

[1] Jacob Berkman. Project Info for Bug-Buddy. `http://www.advogato.org/proj/bug-buddy/`, 2002.

[2] Microsoft Corporation. Dr. Watson Overview. `http://www.microsoft.com/TechNet/prodtechnol/winxppro/proddocs/drwatson%_overview.asp`, 2002.

[3] Jeffrey S. Foster et al. CQual: A tool for adding type qualifiers to C. `http://www.cs.berkeley.edu/~jfoster/cqual/`.

[4] David Gay and Alexander Aiken. Memory Management with Explicit Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

[5] David Gay and Alexander Aiken. Language Support for Regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[6] Ben Liblit and Alex Aiken. Building a Better Backtrace: Techniques for Postmortem Program Analysis. *UC Berkeley Computer Science Technical Report UCB——CSD-02-1203*, October 2002.

[7] George C. Necula, Scott McPeak, Westley Weimer, Raymond To, and Aman Bhargava. CIL: Infrastructure for C Program Analysis and Transformation. `http://www.cs.berkeley.edu/~necula/cil`, 2002.

[8] OpenSSH. `http://www.openssh.com/`.

[9] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information Flow Security. *IEEE Journal on Selected Areas in Communications*, January 2003.

[10] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *10th USENIX Security Symposium*, pages 201–220, August 2001.

[11] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice & Experience*, 26:1–18, 1996.

[12] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[13] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[14] Brandon Wirtz. Dr. Watson's a Big-Mouth. `http://www.griffin-digital.com/dr__watson.htm`, 2002.