Using Execution Transactions To Recover From Buffer Overflow Attacks

Stelios Sidiroglou
Columbia University
stelios@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract

We examine the problem of containing buffer overflow attacks in a **safe** *and* **efficient** *manner. Briefly, we dynamically augment source code to catch stack and heap-based buffer overflow and underflow attacks, and recover from them by allowing the program to continue execution. Our hypothesis is that we can treat each code function as a transaction that can be aborted when an attack is detected, without affecting the application's ability to correctly execute. Furthermore, our approach allows us to selectively enable or disable components of this defensive mechanism in response to external events, allowing for a direct tradeoff between security and performance. We apply our system to the problem of containing worms, combining this defensive mechanism with a honeypot-like configuration. This approach allows us to detect previously unknown attacks and automatically adapt an application's defensive posture at a negligible performance cost.*

The primary benefits of our scheme are its low impact on application performance, its ability to respond to attacks without human intervention, its capacity to handle previously unknown vulnerabilities, and the preservation of service availability. We implemented the scheme as a stand-alone tool, DYBOC, which we use to instrument a number of vulnerable applications. Our performance benchmarks indicate a slow-down of 20.1% for Apache in full-protection mode. We validate our transactional hypothesis via two experiments: first, by applying our scheme to 17 vulnerable applications, successfully fixing 14 of them; second, by examining the behavior of Apache when each of 154 potentially vulnerable routines are made to fail, resulting in correct behavior in 139 of cases.

1 Introduction

The prevalence of buffer overflow attacks as a preferred intrusion mechanism, accounting for approximately half the CERT advisories in the past few years [54], has elevated them into a first-order security concern. Such attacks exploit software vulnerabilities related to input (and in particular input length) validation, and allow attackers to inject code of their choice into an already running program. The ability to launch such attacks over a network has resulted in their use by a number of highly publicized computer worms [51, 1, 5, 7, 39, 57, 6, 49].

In their original form [8], such attacks seek to overflow a buffer in the program stack and cause control to be transfered to the injected code. Similar attacks overflow buffers in the program heap [36, 2], virtual functions and handlers [43], or use other injection vectors (*e.g.,* format strings [3]). Due to the impact of these attacks, a variety of techniques for detecting, removing, containing, or mitigating buffer overflows have been developed over the years. Although detection is perhaps the most desirable solution, this is a very difficult problem, for which only partial solutions exist (*e.g.,* [15, 31]). Each of these techniques suffers from at least one of the following problems (this is not an exhaustive list):

- Most importantly, there is a poor trade-off between security and availability: once an attack has been detected, the only option available is to terminate program execution [19], since the stack has already been overwritten. Although this is arguably better than allowing arbitrary code to execute, program termination is not always a desirable alternative (especially for critical services). *Automated, high-volume attacks, e.g., a worm outbreak,* *can exacerbate the problem by suppressing a server that is safe from infection but is being continuously probed and thus crashes.*

- Severe impact in the performance of the protected application: especially with dynamic techniques that seek to detect and avoid buffer overflow attacks during program execution by instrumenting memory accesses, the performance degradation can be significant.

- Ease of use: especially as it applies to translating applications to a safe language such as Java or using a new library that implements safe versions of commonly abused routines.

An ideal solution uses a comprehensive, perhaps "expensive" protection mechanism only where needed and allow applications to gracefully recover from such attacks, in conjunction with a low-impact protection mechanism that prevents intrusions at the expense of service disruption. We have developed such a mechanism that automatically instruments all statically and dynamically allocated buffers in an application so that any buffer overflow or underflow attack will cause transfer of the execution flow to a specified location in the code, from which the application can resume execution. *Our hypothesis is that function calls can be treated as transactions that can be aborted when a buffer overflow is detected, without impacting the application's ability to execute correctly.* Nested function calls are treated as sub-transactions, whose failure is handled independently. Our mechanism takes advantage of standard memory-protection features available in all modern operating systems and is highly portable. We implemented our scheme as a stand-alone tool, named DYBOC (DYnamic Buffer Overflow Containment), which simply needs to be run against the source code of the target application. Previous research [45, 46] has applied a similar idea in the context of a safe language runtime (Java); we extend and modify that approach for use with unsafe languages.

We apply DYBOC to 17 open-source applications for which buffer overflow exploits are known, correctly mitigating the effects of these attacks (allowing the program to continue execution without any harmful side effects) for 14 of the applications. In the remaining 3 cases, the program terminated; in no case was the attack successful. Although a contrived micro-benchmark shows a performance degradation of up to 440%, measuring the ability of an instrumented instance of the Apache web server indicates a performance penalty of only 20.1%. We validate our hypothesis on the recovery of execution transactions by evaluating the effects on program execution on the Apache web server. Our results show that when each of the 154 potentially vulnerable routines are forced to fail, 139 result in correct behavior. Albeit we focus on stack-based attacks, our approach can be used against heap overflows.

Although we believe this performance penalty (as the price for security and service availability) to be generally acceptable, we extend our scheme to protect only against specific exploits that are dynamically detected. This approach lends itself particularly well to defending against network worms. Briefly, we use an instrumented version of the application (*e.g.,* web server) in a sandboxed environment, with all protection checks enabled. This environment is operated *in parallel with* the production servers, but is not used to serve actual requests nor are requests delayed. Rather, it is used to detect "blind" attacks, such as when worm or an attacker is randomly scanning and attacking IP addresses. We use this environment as a "clean room" environment to test the effects of "suspicious" requests, such as potential worm infection vectors. A request that causes a buffer overflow on the production server will have the same effect on the sandboxed version of the application. The instrumentation allows us to determine the buffers and functions involved in a buffer overflow attack. This information is then passed on to the production server, which enables that subset of the defenses that is necessary to protect against the detected exploit. In contrast with our previous work, where patches were dynamically generated "on the fly" [50], DYBOC allows administrators to test the functionality and performance of the software with all protection components enabled. Even by itself, however, the honeypot-mode of operation can significantly accelerate the identification of new attacks and the generation of patches or the invocation of other remediation mechanisms.

The remainder of this paper is organized as follows. We describe our approach and the prototype implementation in Section 2. We then evaluate its performance and effectiveness in Section 3, give an overview of related work in Section 4, and conclude the paper in Section 5.

2 Our Approach

The core of our approach is to automatically instrument parts of the application source code[1] that may be vulnerable to buffer overflow attacks (*i.e.,* buffers declared in the stack or the heap) such that overflow or underflow attacks cause an exception. Our goal then is to catch these exceptions and recover the program execution from a suitable location. This description raises several questions:

- Which buffers are instrumented?
- What is the nature of the instrumentation?
- How can we recover from an attack, once it has been detected?
- Can all this be done efficiently and effectively?

In the following subsections we answer these questions, also describing the main components of our system. The question of efficiency and effectiveness is mostly addressed in the next section.

2.1 Instrumentation

Since our goal is to contain buffer overflow attacks, our system instruments all statically and dynamically allocated buffers, and all uses of these buffers. In principle, we can combine our system with a static analysis tool such as [24, 54, 31, 48, 21, 15] to identify those buffers (and uses of buffers) that are provably safe from exploitation. Although such an approach would be an integral part of a complete system, we do not examine it further here; we focus on the details of the dynamic protection mechanism. Likewise, we expect that our system would be used in conjunction with a mechanism like StackGuard [19] or ProPolice to prevent successful intrusions against attacks we are not yet aware of; following such an attack, we can enable the dynamic protection mechanism to prevent service disruption. We should also note the "prove and check" approach has been used in the context of software security in the past, most notably in CCured [42]. In the remainder of this paper, we will focus on stack-based attacks, although our technique can equally easily defend against heap-based ones.

For the code transformations we use TXL [37], a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors. The grammar responsible for parsing the source input is specified in a notation similar to Extended Backus-Naur (BNF). Several parsing strategies are supported by TXL making it comfortable with ambiguous grammars allowing for more "natural" user-oriented grammars, circumventing the need for strict compiler-style "implementation" grammars. In our system, we use TXL for C-to-C transformations using the GCC C front-end.

The instrumentation itself is conceptually straightforward: we move static buffers to the heap, by dynamically allocating the buffer upon entering the function in which it was previously declared; we de-allocate these buffers upon exiting the function, whether implicitly (by reaching the end of the function body) or explicitly (through a *return* statement). Care must be taken to handle the *sizeof* construct, a fairly straightforward task with TXL.

For memory allocation, we use a version of *malloc()* that allocates two additional zero-filled, write-protected pages that bracket the requested buffer, as shown in Figure 1. *pmalloc()* uses *mprotect()* to mark the surrounding pages as read-only. As *mprotect()* operates at memory page granularity, every memory request is rounded up to the nearest page. The pointer that is returned by *pmalloc()* can be adjusted to immediately catch any buffer overflow or underflow depending on where attention is focused. This functionality is similar to that offered by the *ElectricFence* memory-debugging library, the difference being that *pmalloc()* catches both buffer overflow and underflow attacks.

Figure 2 shows an example of such a translation. Buffers that are already allocated via *malloc()* are simply switched to *pmalloc()*. This is achieved by examining declarations in the source and transforming them to pointers where the size is allocated with a *malloc()* function call. Furthermore, we adjust the C grammar to free the variables before the function returns. After making changes to the standard ANSI C grammar that allow entries such as *malloc()* to be inserted between declarations and statements, the transformation step is trivial.

[1]Although binary rewriting techniques such as those used by RAD [44] may be applicable here, we do not further consider them here due to their significant complexity.

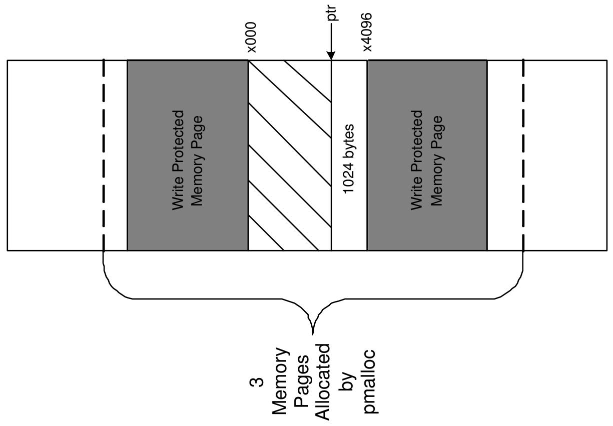
x000
ptr
x4096
Write Protected Memory Page
1024 bytes
Write Protected Memory Page
3 Memory Pages Allocated by pmalloc

Figure 1. **Example of *pmalloc()*-based memory allocation: the trailer and edge regions (above and below the write-protected pages) indicate "waste" memory allocated by *malloc()*. This is needed to ensure that *mprotect()* is applied on complete memory pages.**

Original code	*Modified code*
`int func()`	`int func()`
`{`	`{`
`    char buf[100];`	`    char *buf = pmalloc(100);`
`...`	`...`
`    other_func(buf);`	`    other_func(buf);`
`...`	`...`
`    return 0;`	`    pfree(buf); return 0;`
`}`	`}`

Figure 2. **First-stage code transformation, moving buffers from the stack to the heap using** *pmalloc()*.

For single-threaded, non-reentrant code, it is possible to only use *pmalloc()* once for each previously-static buffer. Generally, however, this allocation needs to be done each time the function is invoked. We shall see how to minimize this cost in Section 2.3.

Any overflow (or underflow) on a buffer allocated via *pmalloc()* will cause the process to receive a Segmentation Violation (SEGV) signal, which is caught by a signal handler we have added to the source code. It is then the responsibility of the signal handler to recover from such failures.

2.2 Recovery: Execution Transactions

In determining how to recover from such exception, we introduce the hypothesis of an **execution transaction.** Very simply, we posit that for the majority of code (and for the purposes of defending against buffer overflow attacks), we can treat each function execution as a transaction (in a manner similar to a sequence of operations in a database) that can be aborted without adversely affecting the graceful termination of the computation. Each function call from inside that function can itself be treated as a transaction, whose success (or failure) does not contribute to the success or failure of its enclosing transaction. Under this hypothesis, it is sufficient to snapshot the state of the program execution when a new transaction begins, detect a failure per our previous discussion, and recover by aborting this transaction and continuing the execution of its enclosing transaction. Currently, we focus our efforts inside the process address space, not dealing with rolling back I/O. For this purpose, a virtual file system approach can be employed to roll back any I/O that is associated with a process. We plan to address this further in future work.

Note that our hypothesis does not imply anything about the correctness of the resulting computation, when a failure occurs. Rather, it merely states that if a function is prevented from overflowing a buffer, it is sufficient to continue execution at its enclosing function, "pretending" the aborted function returned. Depending on the return type of the function, a set of heuristics is employed so as to determine an appropriate error return value that is, in turn, used by the program to handle error conditions. Details of this approach are described in Section 2.2. Our underlying assumption is that the remainder of the program can handle truncated data in a buffer in a graceful manner. For example, consider the case of a buffer overflow vulnerability in a web server, whereby extremely long URLs cause the server to be subverted: by catching such an overflow, the web server will simply try to process the truncated URL (which may simply be garbage, or may point to a legitimate page).

A secondary assumption is that functions that are thusly aborted do not have other side effects (*e.g.,* touch global state), or that such side effects can be ignored. *Obviously, neither of these two conditions can be proven, and examples where they do not hold can trivially be constructed.* Since we are interested in the actual behavior of real software, we experimentally evaluate our hypothesis in Section 3. Note that, in principle, we could checkpoint and recover from each instruction (line of code) that "touches" a buffer; doing so, however, would likely prove prohibitively expensive.

The mechanism for implementing this recovery is straightforward: we use *sigsetjmp()* to snapshot the location to which we want to return once an attack has been detected. The effect of this operation is to save the stack pointers, registers, and program counter, such that the program can later restore their state. We also inject a signal handler (initialized early in *main()*) that catches the SIGSEGV[2] and calls *siglongjmp()*, restoring the stack pointers and registers (including the program counter) to their values prior to the call of the offending function (in fact, they are restored to their values as of the call to *sigsetjmp()*):

```
void sigsegv_handler() {  siglongjmp(global_env, 1);  }
```

The program will then re-evaluate the injected conditional statement that includes the *sigsetjmp()* call. This time, however, the return value will cause the conditional to evaluate to false, thereby skipping execution of the offending function. Note that the targeted buffer will contain exactly the amount of data (infection vector) it would if the offending function performed correct data-truncation.

In our example, after a fault, execution will return to the conditional statement just prior to the call to *other_func()*, which will cause execution to skip another invocation of *other_func()*. If *other_func()* is a function such as *read(), strcpy(), or sprintf()* (*i.e.,* code with no side effects), the result will be equivalent to a situation where these functions correctly handled array bounds checking.

There are two benefits in this approach. First, objects in the heap are protected from being overwritten by an attack on the specified variable, since there is a signal violation when data is written beyond the allocated space. Second, we can recover gracefully from an overflow attempt, since we can recover the stack context environment prior to the offending function's call, and effectively *siglongjmp()* to the code immediately following the routine that caused the overflow or underflow. While the contents of the stack can be recovered by restoring the stack pointer, special care must be placed in handling the state of the heap. In order to deal with data corruption in the heap, we can employ data structure consistency constraints, as described in[20], to detect and recover from such errors.

Thus, the code in our example from Figure 2 will be transformed as shown in Figure 3 (grayed lines indicate changes from the previous example).

To accommodate multiple functions checkpointing different locations during program execution, a globally defined *sigjmp_buf* structure always points to the latest snapshot to recover from. Each function is responsible for saving and restoring this information before and after invoking a subroutine respectively, as shown in Figure 4.

Finally, functions may also refer to global variables; ideally, we would like to unroll any changes made to them by an aborted transaction. The use of such variables can be determined fairly easily via lexical analysis of the instrumented function: any *l-values* not defined in the function are assumed to be global variables (globals used as *r-values* do not cause any changes to their values, and can thus be safely ignored). Once the name of the global variable has been determined, we scan the code to determine its type. If it is a basic type (*e.g.,* integer, float, character), a fixed-size array of a basic type, or a statically allocated structure, we define a temporary variable of the same type in the enclosing function and save/restore its original value as needed. In the example shown in Figure 5, variable

[2]Care must be taken when handling this, to avoid an endless loop on the signal handler if another such signal is raised while in the handler. We validated our approach on OpenBSD and Linux RedHat.

```
int func()
{
  char *buf;
  buf = pmalloc(100);
...
  if (sigsetjmp(global_env, 1) == 0) {
     other_func(buf); /* Indented */
  }
...
  pfree(buf);
  return 0;
}

/* Global definitions */
sigjmp_buf global_env;
```

Figure 3. **Saving state for recovery.**

```
int func()
{
  char *buf;
  sigjmp_buf curr_env;
  sigjmp_buf *prev_env;
  buf = pmalloc(100);
...
  if (sigsetjmp(curr_env, 1) == 0) {
    prev_env = global_env;
    global_env = &curr_env;
    other_func(buf); /* Indented */
    global_env = prev_env;
  }
...
  pfree(buf);
  return 0;
}
```

Figure 4. **Saving previous recovery context.**

"global" is used in *other_func()*.

Unfortunately, dynamically allocated global data structures (such as hash tables or linked lists) are not as straightforward to handle in this manner, since their size may be determined at run time and thus be indeterminate to a static lexical analyzer. One possibility is to modify the dynamic memory allocator to record additional information as to the size of each memory chunk, and combine this with lexical analysis of the data structure definition to determine which fields are pointers. This would likely have a significant impact on performance, however, especially for large data structures such as hash tables. Another possible solution is to provide a framework for programmer assisted code annotation. Through this mechanism, developers could identify functions that "touch" global or otherwise sensitive data structures so that they can be handled with care in the transformation process. Special handling of these data structures could translate to skipping the enclosing function completely or adding data structure constraints that can be used to effectively repair inconsistent data structures and enable the program to continue to operate successfully[**?**]. We plan to address this issue in future work.

2.3 Dynamic Defensive Postures

'Eternal vigilance is the price of liberty.'
- Wendell Phillips, 1852

Unfortunately, when it comes to security mechanisms, vigilance takes a back seat to performance. Thus, although our mechanism can defend against all buffer overflow attacks and (as we shall see in Section 3) maintains service availability in the majority of cases, this comes at the cost of performance degradation. Although such degradation seems to be modest at least for some applications (about 20% for Apache, see Section 3), it is conceivable that other applications may suffer a significant performance penalty, if all buffers are instrumented with our system (for example, a worst-case micro-benchmark measurement indicates a 440% slowdown). One possibility we already mentioned is the use of static analysis tools to reduce the number of buffers that need to be instrumented; however, it is very likely that a significant number of these will be remain unresolved, requiring further protection.

However, our scheme makes it possible to *selectively* enable or disable protecting specific buffers in specific functions, in response to external events (*e.g.,* an administrator command, or an automated intrusion detection system). In the simplest case, an application may execute with all protection disabled, only to assume a more defensive posture as a result of increased network scanning and probing activity. This allows us to avoid paying the cost of instrumentation most of the time, while retaining the ability to protect against attacks at a fast pace. Although this strategy entails some risks (specifically, its exposure to a successful directed attack without prior warning or other indications), it may be the only alternative when we wish to simultaneously achieve security, availability, **and**

```
/* Global variables */
int global;



int func()
{
    char *buf;
    sigjmp_buf curr_env;
    sigjmp_buf *prev_env;
    buf = pmalloc(100);
    int temp_dyboc_global;
...
    if (sigsetjmp(curr_env, 1) == 0) {
      temp_dyboc_global = global;
      prev_env = global_env;
      global_env = &curr_env;
      other_func(buf); /* Indented */
      global_env = prev_env;
    } else {
      global = temp_dyboc_global;
    }
...
    pfree(buf);
    return 0;
}
```

Figure 5. **Saving global variable.**

```
int func()
{
    char *buf;
    sigjmp_buf curr_env, *prev_env;
    char _buf[100];
    if (dyboc_flag(827))
      buf = pmalloc(100); /* Indented */
    else
      buf = _buf;
...
    if (dyboc_flag(1821)) {
      if (sigsetjmp(curr_env, 1) == 0) {
        prev_env = global_env;
        global_env = &curr_env;
        other_func(buf);
        global_env = prev_env;
      }
    } else {
      other_func(buf);
    }
...
    if (dyboc_flag(827)) {
      pfree(buf); /* Indented */
    }
    return 0;
}
```

Figure 6. **Enabling DYBOC conditionally.**

performance.

The basic idea is to only use *pmalloc()* and *pfree()* if a flag instructs the application to do so; otherwise, the transformed buffer is made to point to a statically allocated buffer. Similarly, the *sigsetjmp()* operation is performed only when the relevant flag indicates so. This flagging mechanism is implemented through the *dyboc_flag()* macro, which takes as argument an identifier for the current allocation or checkpoint, and returns true if the appropriate action needs to be taken. Continuing with our previous example, the code will be transformed as shown in Figure 6.

Note that there are three invocations of *dyboc_flag()*, using two different identifiers: the first and last invocation use the same identifier, which indicates whether a particular buffer should be *pmalloc()*'ed or be statically allocated; the second invocation, with a different identifier, indicates whether a particular transaction (function call) should be checkpointed. We use different identifiers to allow for the mixing of allocation and use of buffers across the program code: different invocations of the same function from different locations in the code will get assigned different identifiers, allowing them to be protected independently, at a very high level of granularity.

To implement the signaling mechanism, we use a shared memory segment of sufficient size to hold all identifiers (1 bit per flag). *dyboc_flag()* then simply tests the appropriate flag. A second process, acting as the *notification monitor* is responsible for setting the appropriate flag, when notified through a command-line tool or an automated mechanism. Turning off of a flag requires manual intervention by the administrator. We are not concerned about possible memory leaks due to the obvious race condition (turning off the flag while a buffer is already allocated), since we expect the administrator to restart the service under such rare circumstances, although these can be addressed with additional checking code. Another mechanism that can be used to address memory leaks and inconsistent data structures is that of recursive restartability [13]. We plan to investigate the effects of this recovery mechanism in future work.

2.4 Worm Containment

Recent incidents have demonstrated the ability of self-propagating code, also known as “network worms,” to infect large numbers of hosts, exploiting vulnerabilities in the largely homogeneous deployed software base [7, 57, 39] (or even a small homogeneous base [49]), often affecting the offline world in the process [33]. Even when a worm carries no malicious payload, the direct cost of recovering from the side effects of an infection epidemic can be tremendous. Thus, countering worms has recently become the focus of increased research, generally focusing on content-filtering mechanisms combined with large-scale coordination strategies [40, 52].

Despite some promising early results, we believe that this approach will be insufficient by itself in the future. We base this primarily on two observations. First, to achieve coverage, such mechanisms are intended for use by routers (*e.g.,* Cisco’s NBAR); given the routers’ limited budget in terms of processing cycles per packet, even mildly polymorphic worms (mirroring the evolution of polymorphic viruses, more than a decade ago) are likely to evade such filtering, as demonstrated recently in [16]. Network-based intrusion detection systems (NIDS) have encountered similar problems, requiring fairly invasive packet processing and queuing at the router or firewall. When placed in the application’s critical path, as such filtering mechanisms must, they will have an adverse impact on performance. Second, end-to-end “opportunistic” encryption in the form of TLS/SSL or IPsec is being used by an increasing number of hosts and applications. We believe that it is only a matter of time until worms start using such encrypted channels to cover their tracks. Similar to the case for distributed firewalls, these trends argue for an end-point worm-countering mechanism.

The mechanism we have described allows us to create an autonomous mechanism for combating a scanning worm that does not require snooping the network. We utilize two instances of the application to be protected (*e.g., a web server*), both instrumented as described above. The production server (which handles actual requests) is operating with all security disabled; the second server, which runs in honeypot mode, is listening on an un-advertised address. A scanning worm such as Blaster, CodeRed, or Slammer (or an automated exploit toolkit that scans and attacks any vulnerable services without human intervention) will trigger an exploit on the honeypot server; our instrumentation will allow us to determine which buffer and function are being exploited by the particular worm or attack. This information will then be conveyed to the production server notification monitor, which will set the appropriate flags. A service restart may be necessary, to ensure that no instance of the production server has been infected while the honeypot was detecting the attack.

As a result, targeted services can automatically enable those parts of their defenses that are necessary to defend against a particular attack, without incurring the performance penalty at other times, and cause the worm to slow down. Note that there is no dependency on some critical mass of collaborating entities, as is the case with several other schemes: defenses are engaged in a completely decentralized manner, independent of other organizations’ actions. Wide-spread deployment would cause worm outbreaks to subside relatively quickly, as vulnerable services become immune as they are being exploited. This system can protect against zero-day attacks [34], for which no patch or signature is available.

3 Experimental Evaluation

To test the capabilities of our system, we conducted a series of experiments and performance measurements. In the first, rather contrived scenario, we constructed a simple file-serving application that had a buffer overflow vulnerability and contained superfluous services where we could test against stack-smashing attacks. The server, written in C, used a simple two-phase protocol where a service is requested (different functions) and then the application waits for network input.

A buffer overflow attack was constructed that overwrites the return address and attempts to get access to a root shell. Specific to the set of actions that we have implemented thus far, we recompile the TXL-transformed code, and run a simple functionality test on the application (whether it can correctly serve a given file). The test is a simple script that attempts to access the available service. This application was an initial proof-of-concept for our system, and did not prove the correctness of our approach. More substantial results were acquired through the examination of the applications provided by the Code Security Analysis Kit (CoSAK) project.

3.1 CoSAK Data

To determine the validity of our execution transactions hypothesis, we examined a number of vulnerable open-source software products. This data was made available through the Code Security Analysis Kit (CoSAK) project from the software engineering research group at Drexel university. CoSAK is a DARPA-funded project that is developing a toolkit for software auditors to assist with the development of high-assurance and secure software systems. They have compiled a database of thirty open source products along with their known vulnerabilities and respective patches. This database is comprised of general vulnerabilities, with a large number listed as susceptible to buffer overflow attacks. We applied DYBOC against this data set; the results are illustrated in Appendix A. Note that some of these applications are not in fact network services, and would thus probably not be susceptible to a worm. However, they should serve as a representative sample of buffer overflow vulnerabilities.

As illustrated in the appendix, our tests resulted in fixing 14 out of 17 ”fixable” buffer overflow vulnerabilities, with 82% success rate. The remaining 14 packages were not tested because their vulnerabilities were unrelated (non buffer-overflow). In the remaining 3 cases (those for which our hypothesis appeared not to hold), we manually inspected the vulnerabilities and determined that what would be required to provide an appropriate fix are adjustments to the DYBOC tool to cover special cases, such as handling multi-dimensional buffers and pre-initialized arrays; although these are important in a complete system, we felt that our transaction execution hypothesis was validated with the initial experiment.

It is interesting to note that the majority of the vulnerabilities provided by the CoSAK dataset were caused by calls to the *strcpy()* routine. Examination of the respective security patches showed that for most cases the buffer overflow susceptibility could be repaired by a respective *strncpy()*. Furthermore, most routines did not check for return values and did not include routines within the routines, thus providing fertile ground for use of our *pmalloc()* approach.

3.2 Execution Transactions

In order to validate our hypothesis on the recovery of execution transactions, we experimentally evaluate its effects on program execution on the Apache web server. We run a profiled version of Apache against a set a concurrent requests generated by ApacheBench and examine the subsequent call-graph generated by these requests with gprof.

The call tree is analyzed in order to extract leaf functions. The leaf functions are, in turn, employed as potentially vulnerable transactions. As mentioned previously, we treat each function execution as a transaction that can be aborted without incongruously affecting the normal termination of computation. Armed with the information provided by the call-graph, we run a TXL script that inserts an early return in all the leaf functions, simulating an aborted transaction.

This TXL script operates on a set of heuristics that were devised for the purpose of this experiment. Briefly, depending on the return type of the leaf function, an appropriate value is returned. For example, if the return type is an *int,* a -1 is returned; if the value is *unsigned int,* we return 0, *etc.* A special case is used when the function returns a pointer. Specifically, instead of blindly returning a *NULL*, we examine if the pointer returned is dereferenced further down by the calling function. In this case, we issue an early return immediately before the terminal function is called. For each simulated aborted transaction, we monitor the program execution of Apache by running httperf [41], a web server performance measurement tool. Specifically, we examined 154 leaf functions.

The results from these tests were very encouraging; 139 of the 154 functions completed the httperf tests successfully: program execution was not interrupted. What we found to be surprising, was that not only did the program not crash but in some cases all the pages were served correctly. This is probably due to the fact a large number of the functions are used for statistical and logging purposes. Furthermore, out of the 15 functions that produced segmentation faults, 4 did so at start up.

3.3 Performance

To understand the performance implications of our patch-generation engine and protection mechanism, we ran a set of performance benchmarks. We first measure the worst-case performance impact of DYBOC in a contrived program; we then ran DYBOC against the Apache web server and measure the overhead of our system with full protection enabled.

Micro Benchmark The first benchmark is aimed at helping us understand the performance implications of our DYBOC engine. For this purpose, we use an austere *C* program that makes an *strcpy()* call using a statically allocated buffer as the basis of our experiment.

After patching the program with DYBOC, we compare the performance of the patched version to that of the original version by examining the difference in processor cycles using the Read Time Stamp Counter (RDTSC), found in Pentium class processors. The results illustrated by Figure 7 indicate the mean time, in microseconds (adjusted from the processor cycles), for 100,000 iterations. The performance overhead for the patched, protected version is 440%, a value that is expected given the complexity of the *pmalloc()* routine relative to the simplicity of calling *strcpy()* for small strings.

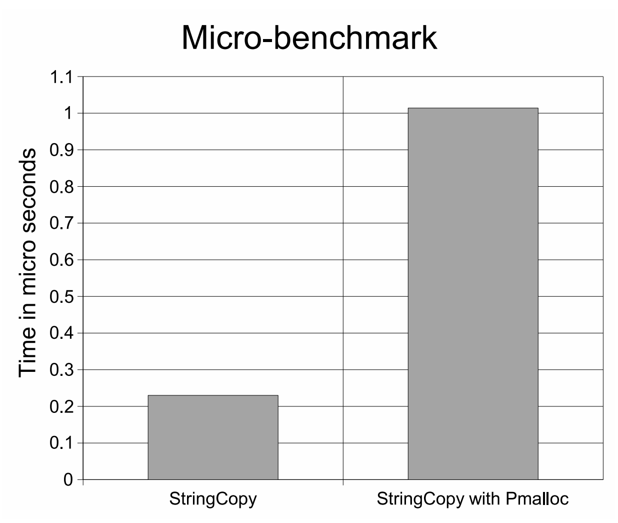

Figure 7. **Micro-benchmark results.**

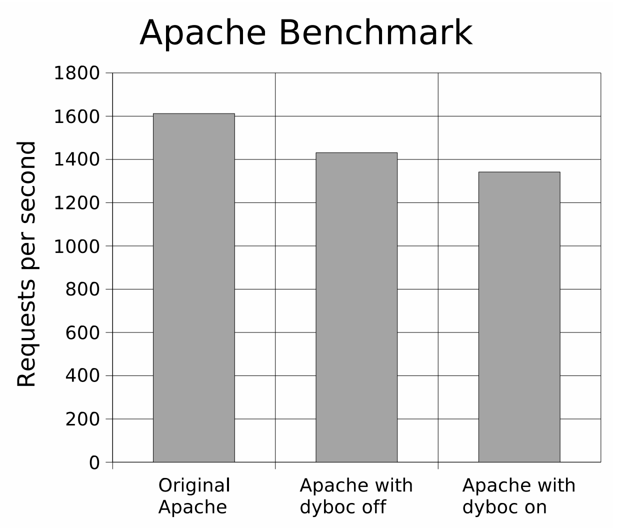

Figure 8. **Apache benchmark results.**

Macro Benchmark We also used DYBOC on the Apache web server, version 2.0.49. Apache was chosen due to its popularity and source-code availability. Basic Apache functionality was tested, omitting additional modules. The purpose of the evaluation was to examine the overhead of preemptive patching of a software system. The tests were conducted on a PC with a 2GHz Intel P4 processor and 1GB of RAM, running Debian Linux (2.6.5-1 kernel).

We used ApacheBench [4], a complete benchmarking and regression testing suite. Examination of application response is preferable to explicit measurements in the case of complex systems, as we seek to understand the effect on overall system performance.

Figure 8 illustrates the requests per second that Apache can handle for 6000 concurrent requests. There is an 20.1% overhead for the patched version of Apache over the original, which is expected since the majority of the patched buffers belong to utility functions that are not heavily used. This result is very encouraging, as it validates the assumption that a software system can be preemptively patched without incurring a prohibitive performance hit. Furthermore, this result is an indication of the worst-case analysis, since all the protection flags were enabled.

Space Overhead Of further interest are the increases in the number of lines and binary size of the patched version. The line count for the server files in the original version of Apache is 226,647, while the patched version is 258,061 lines long, representing an increase of 13.86%. Note that buffers that are already being allocated with *malloc()* (and de-allocated with *free()*) are simply translated to *pmalloc()* and *pfree()* respectively, and thus do not contribute to an increase in the line count. The binary size of the original version was 2,231,922 bytes, while the patched version of the binary was 2,259,243, an increase in the order of 1.22%. Similar results are obtained with OpenSSH 3.7.1. Thus, the impact of our approach in terms of additional required memory or disk storage is minimal.

4 Related Work

Modeling executing software as a transaction that can be aborted has been examined in the context of language-based runtime systems (specifically, Java) in [46, 45]. That work focused on safely terminating misbehaving threads, introducing the concept of "soft termination". Soft termination allows threads to be terminated while preserving the stability of the language runtime, without imposing unreasonable performance overheads. In that approach, threads (or *codelets*) are each executed in their own transaction, applying standard ACID semantics. This allows changes to the runtime's (and other threads') state made by the terminated codelet to be rolled back. The performance overhead of their system can range from 200% up to 2,300%. Relative to that work, our contribution is twofold. First, we apply the transactional model to an unsafe language such as C, addressing several (but not all) challenges presented by that environment. Second, by selectively applying transactional processing, we substantially reduce the performance overhead of the application. However, there is no free lunch: this reduction comes at the cost of allowing failures to occur. Our system aims to automatically evolve a piece of code such that it *eventually* (*i.e.,* once an attack has been observed, possibly more than once) does not succumb to attacks.

Safe Languages and Compilers Safe languages eliminate various software vulnerabilities altogether by introducing constructs that programmers cannot misuse (or abuse). Unfortunately, programmers do not seem eager to port older software to these languages. Java has arguably overcome this barrier, and other safe languages that are more C-like (*e.g.,* Cyclone [27]) may result in wider use of safe languages. In the short and medium term however, "unsafe" languages (in the form of C and C++) are unlikely to disappear, and they will in any case remain popular in certain specialized domains, such as embedded systems.

One step toward the use of safe constructs in unsafe languages is the use of "safe" APIs (*e.g.,* the strl*() API [38]) and libraries (*e.g., libsafe* [9]). While these are, in theory, easier for programmers to use than a completely new language (in the case of libsafe, they are completely transparent to the programmer), they only help protect specific functions that are commonly abused (*e.g.,* the str*() family of string-manipulation function in the standard C library). Vulnerabilities elsewhere in the program remain open to exploitation.

Source Code Analysis Increasingly, source code analysis techniques are brought to bear on the problem of detecting potential code vulnerabilities. The simplest approach has been that of the compiler warning on the use of certain unsafe functions, *e.g., gets()*. More recent approaches [24, 54, 31, 48, 21] have focused on detecting specific types of problems, rather than try to solve the general "bad code" issue, with considerable success. While such tools can greatly help programmers ensure the safety of their code, especially when used in conjunction with other protection techniques, they (as well as dynamic analysis tools such as [35, 32]) offer incomplete protection, as they can only protect against and detect *known* classes of vulnerabilities.

MOPS [15, 14] is an automated formal-methods framework for finding bugs in security-relevant software, or verifying their absence. They model programs as pushdown automata, represent security properties as finite state automata, and use model-checking techniques to identify whether any state violating the desired security goal is reachable in the program. While this is a powerful and scalable (in terms of performance and size of program to be verified) technique, it does not help against buffer overflow or other code-injection attacks. RacerX [22] uses flow-sensitive, inter-procedural analysis to detect race conditions and deadlocks, geared towards debugging of large multi-threaded systems.

CCured [42] combines type inference and run-time checking to make C programs type safe, by classifying pointers according to their usage. Those pointers that cannot be verified statically to be type safe are protected by compiler-injected run-time checks. Depending on the particular application, the overhead of the approach can be up to 150%. MECA [56] allows a programmer to annotate code such that specific security properties of the program can be verified automatically. The focus is to allow for easy annotation of large amounts of code with little effort, by inferring missing annotations from existing ones. In [26], the authors model C string manipulations, which account for many (although not all) buffer overrun vulnerabilities, as linear programs. They then use model solvers based on linear programming that are efficient and accurate. ARCHER [55] symbolically executes the code and checks, using a database about program variables and their current state, whether accesses to buffers are within bounds.

Compiler Tricks Perhaps the best-known approach to countering buffer overflows is StackGuard [19]. This is a patch to the popular *gcc* compiler that inserts a *canary* word right before the return address in a function's activation record on the stack. The canary is checked just before the function returns, and execution is halted if it is not the correct value, which would be the case if a stack-smashing attack had overwritten it. This protects against simple stack-based attacks, although some attacks were demonstrated against the original approach [12], which has since been amended to address the problem.

A similar approach [28], also implemented as a *gcc* patch, adds bounds-checking for pointers and arrays without changing the memory model used for representing pointers. This helps to prevent buffer overflow exploits, but at a high performance cost, since all indirect memory accesses are checked, greatly slowing program execution. A somewhat more efficient version is described in [47]. Stack Shield [53] is another *gcc* extension with an activation record-based approach. Their technique involves saving the return address to a write-protected memory area, which is impervious to buffer overflows, when the function is entered. Before returning from the function, the system restores the proper return address value. Return Address Defense (RAD) [44] uses a redundant copy of the return address to detect stack-overflow attacks. Its innovation lies in the ability to work on pre-compiled binaries using disassembly techniques, which makes it usable for protecting legacy libraries and applications without requiring access to the original source code. These methods are very good at ensuring that the flow of control is never altered via a function-return. However, they cannot detect the presence of any data memory corruption, and hence are susceptible to attacks that do not rely on the return address.

ProPolice [23], another patch for *gcc*, is also similar to Stack Guard in its use of a canary value to detect attacks on the stack. The novelty is the protection of stack-allocated variables by rearranging the local variables so that *char* buffers are always allocated at the bottom of the record. Thus, overflowing these buffers cannot harm other local variables, especially function-pointer variables. This avoids attacks that overflow part of the record and modify the values of local variables without overwriting the canary and the return-address pointer.

MemGuard [19] makes the location of the return address in the function prologue read-only and restores it upon function return, effectively disallowing any writes to the whole section of memory containing the return address. It permits writes to other locations in the same virtual memory page, but slows them down considerably because they must be handled by kernel code. StackGhost [25] is a kernel patch for OpenBSD for the Sun SPARC architecture, which has many general-purpose registers. These registers are used by the OpenBSD kernel for function invocations as register windows. The return address for a function is stored in a register instead of on the stack. As a result, applications compiled for this architecture are more resilient against normal input-string exploits. For deeply-nested function calls, the kernel will have to perform a register window switch, which involves saving some of the registers onto the stack. StackGhost removes the possibility of malicious data overwriting the stored register values by using techniques like write-protecting or encrypting the saved state on the stack. FormatGuard [17] is a library patch for eliminating format string vulnerabilities. It provides wrappers for the *printf()* family of functions that count the number of arguments and match them to the specifiers.

Another related approach is that of program shepherding [30], where an interpreter is used to verify the source and target of any branch instruction, according to some security policy. To avoid the performance penalty of interpretation, their system caches verified code segments and reuses them as needed. Despite this, there is a considerable performance penalty for some applications. A somewhat similar approach is used by *libverify* [9], which dynamically re-writes executed binaries to add run-time return-address checks.

PointGuard [18] encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. This is implemented as an extension to the GCC compiler, which injects the necessary instructions at compilation time, allowing a pure-software implementation of the scheme. Another approach, address obfuscation [11], randomizes the absolute locations of all code and data, as well as the distances between different data items. Several transformations are used, such as randomizing the base addresses of memory regions (stack, heap, dynamically-linked libraries, routines, static data, *etc.*), permuting the order of variables/routines, and introducing random gaps between objects (*e.g.,* randomly pad stack frames or *malloc()*'ed regions). Although very effective against *jump-into-libc* attacks, it is less so against other common attacks, due to the fact that the amount of possible randomization is relatively small (especially when compared to our key sizes). However, address obfuscation can protect against attacks that aim to corrupt variables or other data.

RISE [10] and Instruction Randomization [29] introduce a virtual machine layer that uses a randomized instruction set to deny an attacker the ability to exploit a vulnerability.

5 Conclusion

The main contribution of this paper is the introduction and validation of the *execution transaction* hypothesis, which states that every function execution can be treated as a transaction (in a manner similar to a sequence of database operations) that can be allowed to fail, or forced to abort, without affecting the graceful termination of the computation. We validate this hypothesis by examining a number of open source software packages with known vulnerabilities.

For that purpose, we developed DYBOC, a tool for instrumenting C source code such that buffer overflow attacks can be caught, and program execution continue without any adverse side effects (such as forced program termination). DYBOC allows a system to dynamically enable or disable specific protection checks in running software, potentially as a result of input from external sources (*e.g.,* an IDS engine), at an very high level of granularity. This enables the system to implement policies that trade off between performance and risk, retaining the capability to re-evaluate this trade-off very quickly. This makes DYBOC-enhanced services highly responsive to automated indiscriminate attacks, such as scanning worms. Finally, our preliminary performance experiments indicate that: (a) the performance impact of DYBOC in contrived examples can be significant, but (b) the impact in performance is significantly lessened (less than 10%) in real applications, and (c) this performance impact is further lessened by utilizing the dynamic nature of our scheme.

Our plans for future work include enhancing the capabilities of DYBOC by combining it with a static source-code analysis tool, extending the performance evaluation, and further validating our hypothesis by examining a larger number of open source applications.

References

[1] CERT Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. `http://www.cert.org/advisories/CA-2001-19.html`, July 2001.

[2] CERT Advisory CA-2001-33: Multiple Vulnerabilities in WU-FTPD. `http://www.cert.org/advisories/CA-2001-33.html`, November 2001.

[3] CERT Advisory CA-2002-12: Format String Vulnerability in ISC DHCPD. `http://www.cert.org/advisories/CA-2002-12.html`, May 2002.

[4] ApacheBench: a complete benchmarking and regression testing suite. `http://freshmeat.net/projects/apachebench/`, July 2003.

[5] Cert Advisory CA-2003-04: MS-SQL Server Worm. `http://www.cert.org/advisories/CA-2003-04.html`, January 2003.

[6] CERT Advisory CA-2003-21: W32/Blaster Worm. `http://www.cert.org/advisories/CA-2003-20.html`, August 2003.

[7] The Spread of the Sapphire/Slammer Worm. `http://www.silicondefense.com/research/worms/slammer.php`, February 2003.

[8] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[9] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[10] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, October 2003.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[12] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[13] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[14] H. Chen, D. Dean, and D. Wagner. Model Cehcking One Million Lines of C Code. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 171–185, February 2004.

[15] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.

[16] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, August 2003.

[17] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 191–199, August 2001.

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, August 2003.

[19] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[20] B. Demsky and M. C.Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application*, October 2003.

[21] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[22] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of ACM SOSP*, October 2003.

[23] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, June 2000.

[24] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[25] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66, August 2001.

[26] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 345–364, October 2003.

[27] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, June 2002.

[28] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1997.

[29] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.

[30] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–205, August 2002.

[31] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, August 2001.

[32] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of the 12th USENIX Security Symposium*, pages 121–136, August 2003.

[33] E. Levy. Crossover: Online Pests Plaguing the Offline World. *IEEE Security & Privacy*, 1(6):71–73, November/December 2003.

[34] E. Levy. Approaching Zero. *IEEE Security & Privacy*, 2(4):65–66, July/August 2004.

[35] K. Lhee and S. J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, August 2002.

[36] M. Conover and w00w00 Security Team. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, January 1999.

[37] A. J. Malton. The Denotational Semantics of a Functional Tree-Manipulation Language. *Computer Languages*, 19(3):157–168, 1993.

[38] T. C. Miller and T. de Raadt. strlcpy and strlcat: Consistent, Safe, String Copy and Concatentation. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, June 1999.

[39] D. Moore, C. Shanning, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the 2nd Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

[40] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference*, April 2003.

[41] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[42] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the Principles of Programming Languages (PoPL)*, January 2002.

[43] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. *IEEE Security & Privacy*, 2(4):20–27, July/August 2004.

[44] M. Prasad and T. Chiueh. A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, June 2003.

[45] A. Rudys and D. S. Wallach. Transactional Rollback for Language-Based Systems. In *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2001.

[46] A. Rudys and D. S. Wallach. Termination in Language-based Systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.

[47] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, February 2004.

[48] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–216, August 2001.

[49] C. Shannon and D. Moore. The Spread of the Witty Worm. *IEEE Security & Privacy*, 2(4):46–50, July/August 2004.

[50] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.

[51] E. H. Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Purdue University, 1988.

[52] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[53] Vendicator. Stack shield. `http://www.angelfire.com/sk/stackshield/`.

[54] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, February 2000.

[55] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference and the 11th ACM Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 2003.

[56] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 321–334, October 2003.

[57] C. C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, November 2002.

Appendix A. CoSAK Data

System Name	System Call	Functions within functions	Works?	Return value?
bash	strcpy()	None	Yes	No
crond	strcpy()	None	Yes	No
elm	strcpy()	None	Yes	Yes
lukemftp	None(pointers)	None	No	No
lynx	sprintf()	Yes	Yes	No
mailx	strcpy()	Yes	Yes	No
netkit-ftp	None(pointers)	-	No	No
netkit-ping	Memcpy()	None	Yes	No
nmh	sprintf(), strcpy()	None	Yes	No
screen	Format String	None	No	No
sharutils	sscanf()	None	Yes	Yes
stunnel	fdprint()	None	Yes	Yes
sysklogd	read()	None	Yes	Yes
telnetd	sprintf()	None	Yes	No
wu-ftpd	strcat()	None	Yes	No
wu-ftpd	sprintf()	None	Yes	No
zgv-1	strcpy()	None	Yes	No

The column ”Functions within functions” indicates whether the vulnerable system call used in the application invoked another function as part of the parameters to the call. The column ”Return value” indicates whether the vulnerable system call’s return value was checked upon returning from the call. The significance of these columns is pertinent to the application of our *pmalloc()* heuristic.