A Dynamic Mechanism for Recovering from Buffer Overflow Attacks

Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis

Department of Computer Science, Columbia University, USA
{*stelios,ig2111,angelos*}*@cs.columbia.edu*

Abstract. We examine the problem of containing buffer overflow attacks in a **safe** and **efficient** manner. Briefly, we automatically augment source code to dynamically 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. Our approach allows us to enable selectively or disable components of this defensive mechanism in response to external events, allowing for a direct tradeoff between security and performance. We combine our defensive mechanism with a honeypot-like configuration to detect previously unknown attacks, automatically adapt an application's defensive posture at a negligible performance cost, and help determine worm signatures.

Our scheme provides low impact on application performance, the ability to respond to attacks without human intervention, the capacity to handle previously unknown vulnerabilities, and the preservation of service availability. We implement a stand-alone tool, DYBOC, which we use to instrument a number of vulnerable applications. Our performance benchmarks indicate a slow-down of 20% for Apache in full-protection mode, and 1.2% with selective protection. We provide preliminary evidence towards the validity of 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 cases (90%), with similar results for *sshd* (89%) and Bind (88%).

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 [1], has elevated them into a first-order security concern. Such attacks exploit software vulnerabilities related to input (and 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.

In their original form [2], 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, virtual functions and handlers [3, 4], or use other injection vectors such as format strings. Due to the impact of these attacks, a variety of techniques

for removing, containing, or mitigating buffer overflows have been developed over the years. Although bug elimination during development is the most desirable solution, this is a difficult problem with only partial solutions. These techniques suffer from at least one of the following problems:

- 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 [5, 6], 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 (particularly 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 constantly probed and thus crashes.*
- Severe impact in the performance of the protected application: dynamic techniques that seek to detect and avoid buffer overflow attacks during program execution by instrumenting memory accesses, the performance degradation can be significant. Hardware features such as the NoExecute (NX) flag in recent Pentium-class processors [6] address the performance issue, but cover a subset of exploitation methods (*e.g.,* jump-into-libc attacks remain possible).
- 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 allows applications to gracefully recover from such attacks, in conjunction with a low-impact protection mechanism that prevents intrusions at the expense of service disruption.

Our Contribution 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 implement our scheme as a standalone tool, named DYBOC (DYnamic Buffer Overflow Containment), which simply needs to be run against the source code of the target application. Previous research [7, 8] 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, focusing on single-threaded applications. Because we instrument memory regions and not accesses to these, our approach does not run into any problems with pointer aliasing, as is common with static analysis and some dynamic code instrumentation techniques.

We apply DYBOC to 17 open-source applications with known buffer overflow exploits, 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 did the attack succeed. 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%. We provide some preliminary experimental validation of our hypothesis on the recovery of execution transactions by examining its effects on program execution on the Apache web server. We show that when each of the 154 potentially vulnerable routines are forced to fail, 139 result in correct behavior, with similar results for *sshd* and Bind. Our approach can also protect against heap overflows.

Although we believe this performance penalty (as the price for security and service availability) to be generally acceptable, we provide further extensions to our scheme to protect only against specific exploits that are detected dynamically. This approach lends itself well to defending against scanning 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 operates *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 a worm or an attacker is randomly scanning and attacking IP addresses. We use this environment as a "clean room" 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" [9, 10], DYBOC allows administrators to test the functionality and performance of the software with all protection components enabled. Even by itself, the honeypot mode of operation can significantly accelerate the identification of new attacks and the generation of patches or the invocation of other protection mechanisms, improving on the current state-of-the-art in attack detection [11, 12].

We describe our approach and the prototype implementation in Section 2. We then evaluate its performance and effectiveness in Section 3, and give a brief overview of related work in Section 4.

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. We then 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 and describe the main components of our system. The question of efficiency and effectiveness is addressed in the next section.

[1] Binary rewriting techniques may be applicable, but we do not further consider them due to their significant complexity.

2.1 Instrumentation

Since our goal is to contain buffer overflow attacks, our system instruments all statically and dynamically allocated buffers, and all read and writes to these buffers. In principle, we could combine our system with a static analysis tool 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 [5] 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 [13]. 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 [14], 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.

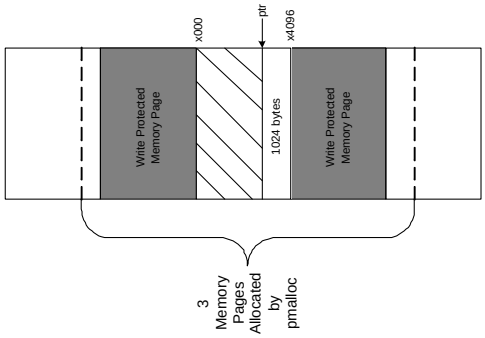

Fig. 1. Example of *pmalloc()*-based memory allocation: the trailer and edge regions (above and below the write-protected pages) indicate "waste" memory allocated by *malloc()*.

The instrumentation is fairly 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).

Original code

```
int func()
{
    char buf[100];
...
    other_func(buf);
...
    return 0;
}
```

Modified code

```
int func()
{
    char *buf = pmalloc(100);
...
    other_func(buf);
...
    pfree(buf); return 0;
}
```

Fig. 2. First-stage transformation, moving buffers from the stack to the heap with *pmalloc()*.

For memory allocation we use *pmalloc()*, our own version of *malloc()*, which allocates two zero-filled, write-protected pages surrounding the requested buffer (Figure 1).

The guard pages are *mmap()*'ed from */dev/zero* as read-only. As *mmap()* 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. Because we *mmap()* pages from */dev/zero*, we do not waste physical memory for the guards (just page-table entries). Some memory is wasted, however, for each allocated buffer, since we round to the next closest page. While this could lead to considerable memory waste, we note that in our experiments the overhead has proven manageable.

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

Any overflow or underflow attack to a *pmalloc()*-allocated buffer 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, and do not deal 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, by adopting the techniques described in [15]. However, there are limitations to what can be done, *e.g.*, network traffic.

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 an error. Depending on the return type of the function, a set of heuristics are 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 2.3. 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: when DYBOC catches 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 most 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 be trivially constructed, e.g., an mmap()'ed file shared with other applications.* 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 be prohibitively expensive.

To implement recovery 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 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() {
    /* transaction(TRANS_ABORT); */
    siglongjmp(global_env, 1);
}
```

(We explain the meaning of the *transaction()* call later in this section.) 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 *strcpy(), or sprintf()* (*i.e.*, code with no side effects), the result is similar to a situation where these functions correctly handled array-bounds checking.

There are two benefits to 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

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

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. To deal with data corruption in the heap, we can employ data structure consistency constraints, as described in [16], 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).

```
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;
```

Fig. 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;
}
```

Fig. 4. Saving previous recovery context.

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.

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 "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. Thus, when we cannot determine the side-effects of a function, we use a different mechanism, assisted by the operating system: we added a new system call, named *transaction()*. This is conditionally invoked (as directed by the *dyboc_flag()* macro) at three locations in the code, as shown in Figure 5.

First, prior to invoking a function that may be aborted, to indicate to the operating system that a new transaction has begun. The OS makes a backup of all memory page permissions, and marks all heap memory pages as read-only. As the process executes and modifies these pages, the OS maintains a copy of the original page and allocates a

```
/* 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;
  /* OR: transaction(TRANS_START); */
      prev_env = global_env;
      global_env = &curr_env;
      other_func(buf); /* Indented */
      global_env = prev_env;
    } else {
      global = temp_dyboc_global;
  /* OR: transaction(TRANS_END); */
    }
...
    pfree(buf);
    return 0;
}
```

Fig. 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;
}
```

Fig. 6. Enabling DYBOC conditionally.

new page (which is given the permissions the original page had, from the backup) for the process to use, in exactly the same way copy-on-write works in modern operating systems. Both copies of the page are kept until *transaction()* is called again. Second, after the end of a transaction (execution of a vulnerable function), to indicate to the operating system that a transaction has successfully completed. The OS then discards all original copies of memory pages that have been modified during processing this request. Third, in the signal handler, to indicate to the OS that an exception (attack) has been detected. The OS then discards all dirty pages by restoring the original pages.

A similar mechanism could be built around the filesystem by using a private copy of the buffer cache for the process executing in shadow mode, although we have not implemented it. The only difficulty arises when the process must itself communicate with another process while servicing a request; unless the second process is also included in the transaction definition (which may be impossible, if it is a remote process on another system), overall system state may change without the ability to roll it back. For example, this may happen when a web server communicates with a back-end database. Our system does not currently address this, *i.e.,* we assume that any such state changes are benign or irrelevant (*e.g.,* a DNS query). Back-end databases inherently support the concept of a transaction rollback, so it is (in theory) possible to undo any changes.

The signal handler may also notify external logic to indicate that an attack associated with a particular input from a specific source has been detected. The external logic may then instantiate a filter, either based on the network source of the request or the contents of the payload.

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 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 remain unresolved, requiring further protection.

Our scheme makes it possible to *selectively* enable or disable protection for specific buffers in 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 quickly. Although this strategy entails some risks (exposure to a successful directed attack with no prior warning), it may be the only alternative when we wish to achieve security, availability, **and** 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 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.

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 a flag requires manual intervention by the administrator. We not address memory leaks due to the obvious race condition (turning off the flag while a buffer is already allocated), since we currently only examine single threaded cases and we expect the administrator to restart the service under such rare circumstances, although these can be addressed with additional checking code. Other mechanisms that can be used to address memory leaks and inconsistent data structures are recursive restartability [17] and micro-rebooting [18]. We intend to examine these 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 (or even a small homogeneous base [19]), often affecting the offline world in the process [20]. Even when a worm carries no malicious payload, the direct cost of recovering from the side effects of an infection

epidemic can be tremendous. Countering worms has recently become the focus of increased research, generally focusing on content-filtering mechanisms.

Despite some promising early results, we believe that in the future this approach will be insufficient. 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 [21]) are likely to evade such filtering, as demonstrated recently in [22]. 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, as well as cause a large number of false positive alerts [23]. 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. These trends argue for an end-point worm-countering mechanism. Mechanisms detecting misbehavior [24] are more promising in that respect.

The mechanism we have described allows us to create an autonomous mechanism for combating a scanning (but not hit-list) worm that does not require snooping the network. We use 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 [11], 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) 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. The payload that triggered the buffer overflow, the first part of which can be found on the instrumented buffer, may also be used for content-based filtering at the border router (with the caveats described above). Thus, our system can be used in quickly deriving content-filter rules for use by other mechanisms. Active honeypot techniques such as those proposed in [25] can make it more difficult for attackers to discriminate between the honeypot and the production server.

Thus, 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. There is no dependency on some critical mass of collaborating entities, as with some 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 after being exploited. This system can protect against zero-day attacks, 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. Results were acquired through the examination of the applications provided by the Code Security Analysis Kit (CoSAK) project.

Security Analysis 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.

Our tests resulted in fixing 14 out of 17 "fixable" buffer overflow vulnerabilities, a 82% success rate. The remaining 14 packages in the CoSAK suite 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 feel that our initial results were encouraging.

Execution Transaction Validation In order to evaluate the validity of 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 determine which functions are used. These 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 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 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 later 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*, a web server performance measurement tool. Specifically, we examined 154 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. Out of the 15 functions that produced segmentation faults, 4 did so at startup.

Similarly for *sshd*, we iterate through each aborted function while examining program execution during an *scp* transfer. In the case of *sshd*, we examined 81 functions. Again, the results were encouraging: 72 of the 81 functions maintained program execution. Furthermore, only 4 functions caused segmentation faults; the rest simply did not allow the program to start.

For Bind, we examined the program execution of *named* during the execution of a set of queries; 67 functions were tested. In this case, 59 of the 67 functions maintained the proper execution state. Similar to sshd, only 4 functions caused segmentation faults.

Naturally, it is possible that Apache, Bind, and *sshd* will exhibit long-term side effects, *e.g.*, through data structure corruption. Our experimental evaluation through a benchmark suite, which issues many thousand requests to the same application, gives us some confidence that their internal state does not "decay" quickly. To address longer-term deterioration, we can use either micro-rebooting (software rejuvenation) [18] or automated data-structure repair [16]. We intend to examine the combination of our approach with either of these techniques in future work.

Performance Overheads To understand the performance implications of our protection mechanism, we run a set of performance benchmarks. We first measure the worst-case performance impact of DYBOC in a contrived program; we then run DYBOC against the Apache web server and measure the overhead of full protection.

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.

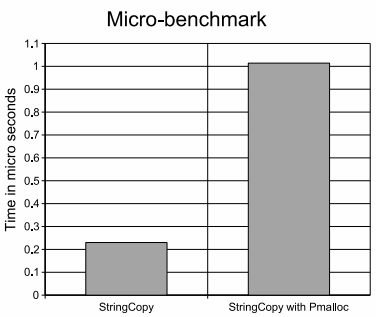

Fig. 7. Micro-benchmark results.

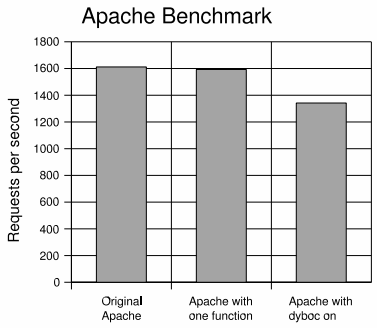

Fig. 8. Apache benchmark results.

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%, which is expected given the complexity of the *pmalloc()* routine relative to the simplicity of calling *strcpy()* for small strings.

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. Our goal 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, 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. There is a 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 an indication of the worst-case analysis, since all the protection flags were enabled; although the performance penalty is high, it is not outright prohibitive for some applications. For the instrumentation of a single buffer and a vulnerable function that is invoked once per HTTP transaction, the overhead is 1.18%.

Space Overheads The line count for the server files in 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 bytes, an increase 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 [8, 7]. 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 code such that it *eventually* (*i.e.*, after an attack has been observed) does not succumb to attacks.

Some interesting work has been done to deal with memory errors at runtime. For example, Rinard *et al.* [26] have developed a compiler that inserts code to deal with writes to unallocated memory by virtually expanding the target buffer. Such a capability aims toward the same goal our system does: provide a more robust fault response rather than simply crashing. The technique presented in [26] is modified in [27] and introduced as *failure-oblivious computing*. Because the program code is extensively re-written to

include the necessary check for *every* memory access, their system incurs overheads ranging from 80% up to 500% for a variety of different applications.

For a more comprehensive treatise on related work, see [28].

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 (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 provide some preliminary evidence on the validity of 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 2%) 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 applications.

References

1. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In: Network and Distributed System Security Symposium. (2000) 3–17
2. Aleph One: Smashing the stack for fun and profit. Phrack **7** (1996)
3. Pincus, J., Baker, B.: Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. IEEE Security & Privacy **2** (2004) 20–27
4. Arce, I.: The Shellcode Generation. IEEE Security & Privacy **2** (2004) 72–76
5. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the 7^{th} USENIX Security Symposium. (1998)
6. Garber, L.: New Chips Stop Buffer Overflow Attacks. IEEE Computer **37** (2004) 28
7. Rudys, A., Wallach, D.S.: Transactional Rollback for Language-Based Systems. In: ISOC Symposium on Network and Distributed Systems Security (SNDSS). (2001)
8. Rudys, A., Wallach, D.S.: Termination in Language-based Systems. ACM Transactions on Information and System Security **5** (2002)

9. Sidiroglou, S., Keromytis, A.D.: A Network Worm Vaccine Architecture. In: Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security. (2003) 220–225
10. Sidiroglou, S., Keromytis, A.D.: Countering Network Worms Through Automatic Patch Generation. IEEE Security & Privacy (2005) (to appear).
11. Provos, N.: A Virtual Honeypot Framework. In: Proceedings of the 13^{th} USENIX Security Symposium. (2004) 1–14
12. Hernacki, B., Bennett, J., Lofgren, T.: Symantec Deception Server Experience with a Commercial Deception System. In: Proceedings of the 7^{th} International Symposiun on Recent Advanced in Intrusion Detection (RAID). (2004) 188–202
13. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code. In: Proceedings of the Principles of Programming Languages (PoPL). (2002)
14. J.R. Cordy, T.R. Dean, A.M., Schneider, K.: Source Transformation in Software Engineering using the TXL Transformation System. Journal of Information and Software Technology **44** (2002) 827–837
15. Sun, W., Liang, Z., Sekar, R., Venkatakrishnan, V.N.: One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In: Proceedings of the 12^{th} ISOC Symposium on Network and Distributed Systems Security (SNDSS). (2005) 265–278
16. Demsky, B., Rinard, M.C.: 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 (OOPSLA). (2003)
17. Candea, G., Fox, A.: Recursive restartability: Turning the reboot sledgehammer into a scalpel. In: Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, IEEE Computer Society (2001) 110–115
18. Candea, G., Fox, A.: Crash-only software. In: Proceedings of the 9th Workshop on Hot Topics in Operating Systems. (2003)
19. Shannon, C., Moore, D.: The Spread of the Witty Worm. IEEE Security & Privacy **2** (2004) 46–50
20. Levy, E.: Crossover: Online Pests Plaguing the Offline World. IEEE Security & Privacy **1** (2003) 71–73
21. Ször, P., Ferrie, P.: Hunting for Metamorphic. Technical report, Symantec Corporation (2003)
22. Christodorescu, M., Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. In: Proceedings of the 12th USENIX Security Symposium. (2003) 169–186
23. Pasupulati, A., Coit, J., Levitt, K., Wu, S., Li, S., Kuo, J., Fan, K.: Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In: Proceedings of the Network Operations and Management Symposium (NOMS). (2004) 235–248, vol. 1
24. Weaver, N., Staniford, S., Paxson, V.: Very Fast Containment of Scanning Worms. In: Proceedings of the 13^{th} USENIX Security Symposium. (2004) 29–44
25. Yegneswaran, V., Barford, P., Plonka, D.: On the Design and Use of Internet Sinks for Network Abuse Monitoring. In: Proceedings of the 7^{th} International Symposium on Recent Advances in Intrusion Detection (RAID). (2004) 146–165
26. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T.: A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In: Proceedings 20^{th} Annual Computer Security Applications Conference (ACSAC). (2004)
27. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., W Beebee, J.: Enhancing Server Availability and Security Through Failure-Oblivious Computing. In: Proceedings 6^{th} Symposium on Operating Systems Design and Implementation (OSDI). (2004)
28. Sidiroglou, S., Keromytis, A.: Countering network worms through automatic patch generation. Technical Report CUCS-029-03, Columbia University (2003)