CASPER: Compiler-Assisted Securing of Programs at Runtime

Gaurav S. Kc, Stephen A. Edwards, Gail E. Kaiser, Angelos Keromytis
Columbia University, Department of Computer Science
500 West 120th Street, New York, NY 10027
{gskc, sedwards, kaiser, angelos}@cs.columbia.edu

Abstract

Ensuring the security and integrity of computer systems deployed on the Internet is growing harder. This is especially true in the case of server systems based on open source projects like Linux, Apache, Sendmail, etc. since it is easier for a hacker to get access to the binary format of deployed applications if the source code to the software is publicly accessible. Often, having a binary copy of an application program is enough to help locate security vulnerabilities in the program. In the case of legacy systems where the source code is not available, advanced reverse-engineering and decompilation techniques can be used to construct attacks.

This paper focuses on measures that reduce the effectiveness of hackers at conducting large-scale, distributed attacks. The first line of defense involves additional runtime checks that are able to counteract the majority of hacking attacks. Introducing diversity in deployed systems to severely diminish the probability of propagation to other systems helps to prevent effective attacks like the DDOS attack against the DNS root servers in October 21, 2002.

1 Introduction

Internet servers are increasingly at risk of being broken into. Worms comprise the majority of attack methods against such systems [5]. This is different from virus and Trojan horse attacks that are commonly propagated via automatically-executing code sent as email attachments. In the general case, a successful attack on a server system is more dangerous than a break-in on a personal computer simply because of the potential for there being more confidential data. For instance, breaking into an Amazon.com server might provide access to credit card data for a large number of customers. Hackers have a wide array of tools such as disassemblers, decompilers, and network monitors that enable them to constantly probe software applications to detect security vulnerabilities, e.g., by reverse-engineering the binary images of compiled application programs. It is much easier to construct attacks that can compromise these vulnerable applications if one has access to the source code of the application.

These vulnerabilities are generally the result of programming bugs, misconfiguration, library bugs, operating system changes, and especially the absence of array bounds checking on the size of input being stored in a buffer array for unsafe languages such as C or C++. This is an unfortunate byproduct of programming languages that produces fast system-level code. This means that poorly-written code or non-robust code leave a vulnerability that can be exploited with a well-constructed input data string or sequence of events, yielding administrative-level access to the attacker.

We are considering two key issues: preventing the compromise of server software by detecting an attempt to circumvent the normal control flow of program execution [1] and limiting the applicability of a successful attack on other servers running the same software. Introducing randomized structure and instruction ordering as part of software compilation is sufficient to help achieve enough diversity in generated binaries so that a single successful attack on a given instance cannot be replicated elsewhere.

We describe the popular stack smashing attack in Section 2, mention some already-proposed techniques for circumventing it in Section 3, present a new technique in Section 4, propose some additional ways to further modify executables in Section 5, and conclude with a discussion of future work in Section 6.

2 Attack techniques: Stack smashing

Although code pages are read-only by default on the Intel x86 architecture, code placed in data memory can be executed freely, making it difficult to determine if the processor is executing foreign code injected by an attack.

The stack smashing technique, the most widely used form of hacking, exploits the fact that the x86 does not distinguish between read and execute accesses of data memory pages. This technique involves overflowing a stack buffer, resulting in the injection of arbitrary code in the buffer memory on the stack. The processor is then made to execute this code, which often ends up yielding root access to the hacker.

2.1 *Aleph One's Smashing the Stack for Fun and Profit*

Aleph One [1] provides a detailed walkthrough of how to exploit a stack buffer vulnerability to both inject attack code and to overwrite the return address of the function to point to the starting address of this injected code. He suggests ways to increase the chances of successfully hacking a system by approximating the return address of the injected code by padding the beginning of the injected code with no-op instructions and approximating the actual position of the return address relative to that of the vulnerable buffer by copying the address of the injected code over a range of locations so that the return address location is covered.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **2002**	2. REPORT TYPE	3. DATES COVERED **00-00-2002 to 00-00-2002**

4. TITLE AND SUBTITLE **CASPER: Compiler-Assisted Securing of Programs at Runtime**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Columbia University,Department of Computer Science,500 West 120th Street,New York,NY,10027**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**6**	

The hacker can have the injected code carry out various tasks, e.g., steal confidential information, mutilate data stores, deface websites, etc. Often, a hacker will leave a back door that will enable him to return to the system at a future point, e.g., by inserting a root-uid entry in /etc/passwd. Usually, the injected code is used to spawn a shell that gives the attacker administrative access.

3 Related Work

3.1 Software diversity

Forrest et al. [6] discuss the advantages of bringing diversity into computer systems, and likens the effects to that which diversity helps cause in biological systems. They observed how the lack of diversity among computer systems can facilitate large-scale replication of exploits due to the identical weakness being present in all instances of the system. Some ideas are presented regarding using randomized compilation to introduce sufficient diversity among software systems to severely hamper large-scale spreading of exploits. Among these, the ones most relevant to this paper involve transformations to memory layout.

3.2 Cyclone and Bounds Checking for C Arrays

In the Cyclone project, Jim et al. [9] propose a dialect of the C programming language that has built-in support for a safer pointer model, more static type checking, etc. This programming language is designed to generate an executable as efficient as that from C while reducing the likelihood of producing unsafe programs. The fact that Cyclone uses a different memory model for its internal data structures renders it incompatible with existing systems written in C or C++.

Jones and Kellys [10] patch to the GCC compiler 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 an unacceptable cost—all indirect memory accesses are checked, greatly slowing program execution.

3.3 StackGuard and MemGuard

Implemented as a compiler patch to GCC, Stackguard [4] inserts a *canary* word right before the return address in a function's activation record on the stack. While trying to overwrite the return address, a stack smashing attack would also overwrite the canary word. The value of the canary word is checked just before the function returns, and the program prints an error message and halts if it has changed. This circumvents simple-minded stack-smashing techniques, although bypass techniques have already been developed [3].

MemGuard [4] 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 over locations in the same virtual memory page, but greatly slows them because they must be handled by kernel code. The large overhead so incurred makes this technique impractical.

3.4 Program shepherding

Kiriansky et al. [11] propose a policy-driven mechanism for closely monitoring and dynamically controlling the flow of program execution. They define different default and customizable security policies for code based on the nature of its origin, whether it was loaded from the local file system, generated by the running program itself, or if it self-mutated. Their system is integrated into an interpreter, which enables the sandboxed checking of running applications and monitoring of their control-flow. While the functionality of this approach is attractive, the fact that it is interpreted makes for significant overhead.

3.5 Libsafe and libverify

The libsafe project [2] replaces potentially dangerous standard library functions with safer implementations that bounds-check parameters. Through clever choices of algorithms, running libsafe can actually decrease program run times. This approach removes all weaknesses due to the absence of bounds checking in standard library functions such as printf, etc. The libsafe system is installed as a dynamically loaded library that intercepts calls to potentially unsafe standard library functions. The main disadvantage of libsafe is that it will fail to protect against vulnerabilities in user-defined or non-standard library code. Also, if an application is compiled with the standard library function statically linked in, the libsafe mechanism will be completely ineffective.

The related libverify project [2] works by maintaining a copy of all object code on the heap, and executing that instead of having the processor fetch and execute code from the text segment. This permits the libverify system to run checks at load time, and also insert mechanisms for runtime checking of the execution, primarily dealing with verifying a functions return address before passing control to the function caller.

3.6 StackGhost

Frantzen and Shuey's StackGhost [7] is implemented as 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. However, 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.

3.7 Stack Shield

Stack Shield [12] 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. This method is very good at ensuring that the flow of control is never altered via a function-return. However, it cannot detect the presence of any data memory corruption, and hence is susceptible to attacks that do not rely solely on the return address.

4 Our Approach

Open source software is very widely deployed on the Internet. Since the same version of the software is running on millions of computers, a single attack that can exploit a given vulnerability will be able to overcome all instances of the software without requiring any extra work on the part of the hacker. This is one of the reasons that hackers have been so effective at breaking into software systems and conducting large-scale DDOS attacks [8]. The more determined hacker can always use reverse-engineering tools on the binary images of compiled application programs to hunt for vulnerabilities. Having access to the source makes the job easier, since one can now search for strings like `printf` or `syslog`, which often reveal possible weaknesses.

We plan to make each binary executable of a particular server program change dynamically so that a successful attack on one copy is not effective on any other, or even on the same one a few hours later. This avoids the problematic monoculture of broadly-deployed applications based on identical binaries, each containing the same vulnerabilities. By eliminating this wide-scale mono-culture, we can significantly minimize the impact and success of large-scale security attacks. There are numerous ways of modifying an application at both the source and binary levels. In either case, the program must continue to behave identically at a high level.

Ensuring that a successful attack cannot exploit a given vulnerability in multiple installations of the same version of an application is helpful for reducing the number of collaborative, distributed attacks. However, it is important to fortify server applications from singular attacks as well. Many authors [4, 7, 12] observe most successful attacks exploit automatic buffers that are allocated in a function's activation record. It is therefore possible to fend off the majority of security attacks by securing the activation record during the execution of the function.

4.1 XOR: A Simple Masking Technique

We have implemented a simple patch to GCC, similar to StackGuard's approach to using the function activation record to detect and handle a security breach. However, instead of introducing a canary word entry in the activation record, we obfuscate the return address at the top of the stack frame by XORing it with a randomly-chosen 32-bit value. Just before the function returns, we XOR the return address with the same 32-bit value to return it to its original value. If an attacker has successfully overrun a buffer and modified the return value, this final XOR will corrupt it, most likely sending the program counter to an illegal address and causing a segmentation fault that halts the program. Figure 1 illustrates the code this patch inserts in a function's prologue and epilogue.

```
foo:
 pushl %ebp
 movl %esp,%ebp
 subl $24,%esp

... body

 movl %ebp,%esp
 popl %ebp
 ret
```

(a)

```
foo:
 pushl %ebp
 movl %esp,%ebp
 subl $24,%esp
 xorl $14351054,4(%ebp)
... body
 xorl $14351054,4(%ebp)
 movl %ebp,%esp
 popl %ebp
 ret
```

(b)

Figure 1: (a) Original and (b) XOR-patched function entry and exit code. The value XORed with the return address is chosen randomly each time the program is compiled. If an attack managed to change the return address at 4(%ebp), the second `xorl` instruction would corrupt it, most likely causing the processor to jump to an illegal location and terminate the program.

For stack smashing attacks alone, both the XOR and StackGuard methods are equally effective at ensuring the integrity of the return address. Both methods guarantee (somewhat) a valid return address before returning control to the calling function. However, it is possible to construct an attack string that can successfully 'tiptoe' over the current canary word and leave it untouched, e.g., by incorporating the 32-bit canary value in the attack string, thus completely circumventing StackGuard's defense mechanism [3]. A major advantage of using the XORed return address over the StackGuard technique is that it is harder to achieve the same since one would need to read the 32-bit value used for the XOR—this value exists only in the code memory, as opposed to the canary word which does exist in data memory.

A sophisticated attack on an XOR-fortified system would need to analyze the binary layout of the running program and extract enough information about the XOR value used to successfully construct an attack on the system. The actual input data sequence that is used to overrun the buffer needs to be modified so that the function epilogue will end up XORing it to make it point to the injected code. This task can be made harder by using custom 32-bit values for each function, and possibly each invocation of each function. This way, the attacker cannot use static analyses to construct an input sequence.

Our approach is less favorable than the StackGuard technique with regards to guaranteed halting or the printing of an informational message upon detection of an attack. However, our technique is a much simpler, smaller way to detect stack smashing attacks. Only two machine instructions are added to each function body: one for the XOR at the function prologue, and one at the epilogue. This results in a significantly faster execution for function invocations. In fact, a program that invokes a small function in a tight loop can easily end up running significantly slower when compiled with the StackGuard patch than with our approach, as we show in Section 4.2.

It might be argued that we could attempt to salvage the program state or even let the application shut down cleanly. However, there are very limited guarantees that can be made about the memory and internal data structures after a successful stack smashing attack. In most cases, it is acceptable to have the program crash, and perhaps generate a core file in the process of

```
#define LIMIT 500000000

static int i = 0;

void inc_global()    { i++;        }
void inc_ptr(int *i) { (*i)++;     }
int inc_r_val(int i) { return i+1; }

int main(int argc, char *argv[]) {
  switch (atoi(argv[1])) {
    case 0:
      while (i < LIMIT) i++;
      break;
    case 1:
      while (i < LIMIT) inc_global();
      break;
    case 2:
      while (i < LIMIT) inc_ptr(&i);
      break;
    case 3:
      while (i < LIMIT) i = inc_r_val(i);
      break;
  }
  return 0;
}
```

Figure 2: Program used in micro-benchmarks.

doing so. This core file can be used at a post-execution analysis phase to determine the exact location and nature of the vulnerability and the attempted exploit. However, in certain cases, it would be better to let the compromised program continue execution so that the attackers intentions can be monitored. Obviously, the corrupted application should be allowed to continue only after dynamically sandboxing the program. If the application is halted, it can be recompiled immediately with a different binary signature, and be restarted. This would be useful when dealing with applications with high availability requirements, e.g. web servers, database servers.

4.2 *Experimental results*

Experiments with the XOR technique produced very promising results. We were able to detect and thwart a simple stack smashing attack with negligible effect on the execution time of the generated code.

We ran some micro-benchmark tests identical to those of StackGuard [4] involving different ways of incrementing an integer a large number of times (Figure 2 shows the program we used). Table 1 shows that our XOR-technique fared much better against the unmodified installation of `gcc-2.95` than StackGuard did against unmodified `gcc-2.7.2.3` (although the two patches were applied to different revisions of gcc, we believe this comparison is valid). The only significant slowdown (a factor of ×2) in our approach was seen with the second test where a zero-argument function is used to increment a global variable. However, this is the worst-case scenario caused by the minuscule-sized function body. The overhead approaches zero as the complexity and size of the functions increase.

We also ran some macro-benchmarks to compare the effects of our XOR-technique on the execution of real-world applications. Table 2 shows the execution time for using both unmodified `gcc-2.95` and our XOR-technique to compile our patched version of `gcc-2.95`. It also shows the results running the `ctags` program on the source code for `vim6.1` text editor. These results show our XOR-based approach requires little overhead.

code	**XOR**	**StackGuard**
i++	0%	0%
void inc()	101%	125%
void int(int *)	13.1%	69%
int inc(int)	11.5%	80%

Table 1: Comparison of execution time overhead between the XOR and StackGuard techniques, showing the execution time overhead of XOR is noticibly lower. (StackGuard numbers from Cowan et al. [4])

Program	**Execution time**	**Exec. time with XOR**
gcc	45m 2.0s	45m 3.5s
ctags	12.902s	15.555s

Table 2: Comparison of execution speeds. The XOR-protected gcc executable took almost the same time; the XOR-protected ctags executable (run on 227k lines of input) took 20% longer.

4.3 *Provisions for debugging*

Being able to debug such “fortified” applications is equally important as securing them in the first place. Most of these techniques that attempt to put off an attacker also make it harder for a debugger like GDB to properly extract runtime information about the execution of a program; for example StackGuards introduction of the canary word changes the stack layout. When using a pre-defined single XOR value, it is possible to hardcode this value into gdb directly and have it compute the return address for an activation record during the execution of a function. This 32-bit value is also needed to help reconstruct the calling sequence by following the dynamic chain of function invocations. However, using different 32-bit values for different functions to achieve increased security means that it becomes increasingly harder to be able to use GDB with the produced executable.

Using individual XOR values for each file (or function) generated by the compiler makes it was necessary to enable a program like gdb to access this information at runtime. We encapsulate this set of XOR values in a static function in the output file, which gdb can dynamically use to interpret the perceived return-address for a given function, yielding the true return address for the function. A different approach involves performing transformations on the functions so that each encapsulates its own XOR information. Obviously, all these transformations have to appear transparent to gdb, while still keeping the XOR information encapsulated, and safely hidden from the program runtime.

5 Other Techniques

In this section, we describe a few additional defense mechanisms against stack smashing techniques. Each of these may not seem that powerful in isolation. However we believe that by using different combinations of these individual technologies, we will be able to achieve a high degree of binary diversity between different installations of the same software.

5.1 Automatic garbage collection of the stack frame

Stack smashing attacks make use of automatic buffers that are allocated on the stack frame for the duration of the function. This section of memory need not contain valid data after the function returns. Hence, the compiler can easily insert a call to `memset` at the end of the function epilogue. This will have the effect of erasing the contents of all local variables or buffers that could possibly contain malicious code.

5.2 Heap-based activation records

Nested function invocations result in the activation records for the functions being adjacent to each other in the data stack. This makes it possible for a buffer overflow in a deeply-nested function to overwrite the return address, or carry out other memory-corrupting operations for a different function, making it harder to have detection and preventative security mechanisms. Allocating each activation record in dynamically allocated heap memory instead of the stack should help avoid this problem. The above-mentioned notion of automatic garbage collection of a used stack frame ties in very well with having activation records on the heap.

5.3 Randomizing memory layout to achieve diversity

The ideas presented in this subsection are very similar in nature to the memory layout transformations proposed by Forrest et al. [6]. Randomly-spaced activation records can be used to counteract a stack smashing attempt by reducing the chance that the modified return-address value will successfully point to the injected code. Another major advantage of this approach is that it makes it even harder for an exploit in one function frame to overwrite data in a calling function's stack frame further up the dynamic chain.

5.4 Diversification within a single system

These security techniques can be repeatedly used to achieve diversity within the same installation of server software. Carrying out recompilation of the applications fairly frequently should help reduce the risk of being attacked. This is because there is very little likelihood of an attacker being able to locate a weakness and successfully devise an attack scheme to exploit that vulnerability before the next scheduled recompilation. Service downtime can be avoided by using a failover strategy to incrementally migrate from the old version of the server binary to the new one every time one needs to switch servers. However, there are disadvantages to this approach since it requires additional time and processor resources for the extra recompilations.

6 Future Work

Being able to secure a deployed system against the majority of existing (and future) attacks might be sufficient for a server system in the general case. However, in order to give any near-total guarantees against possible break-ins, it is necessary to consider other infrequent forms of security attacks and their related exploits. We need to consider the patterns of various other attack techniques like heap-smashing techniques and function pointer overwriting. A language like C++ with an accessible table of function pointers (virtual table) also opens up compiled code to attacks that attempt to corrupt the pointer tables. Strongly-typed languages that heavily depend on the type-safety properties of the language are vulnerable to attacks that can circumvent the type-checking mechanisms such as illegal access to private encapsulated data via type-spoofing. Almost all programming languages have some form of dynamic memory management schemes (manual or automatic, with a garbage collector). A hacker might be able to overwrite header information like the meta-data used by `malloc` in the dynamically allocated blocks to corrupt the data structures related to the memory management subsystems. Our current techniques will not be able to detect attacks like these since the return address for the function will be left intact, with the damage being done elsewhere.

All of the defense mechanisms discussed here involve some speed and size overhead. It might be possible to have the compiler selectively insert precautionary measures when generating code for functions. Simpler functions that obviously cannot be attacked may not need to be fortified. However, it is possible that a vulnerability in one part of the program execution can be used to corrupt memory data in a different location. For instance, a hacker could replace the return address of a safe frame from an inner vulnerable frame without affecting the inner frame itself. Randomly-spaced frames can be used to so that the return address for any given adjacent or ancestor frame will not be at a statically determinable location. A local user could, however, bypass this by inspecting the binary image of the program being executed.

The XOR approach is the main technique we have experimented with to date, and the other techniques discussed in Section 5 are only the first in a series of extensions being considered that would result in diversifying binaries and thwarting various classes of attacks. There are open issues regarding the security of the whole system: using our current technique to compile application code while using original libraries would still leave us with a vulnerable system. We plan to study loader- and linker-based approaches that might address the securing of non-open source libraries and legacy applications. Another advantage of introducing late-binding of the security mechanism is that the compiled units would not expose security details. It is also important to examine how our current and proposed techniques would interact with optimizations, with different kinds of hardware architectures, such as PowerPC, SPARC, MIPS, and embedded processors like ARM. Some of the techniques that we will implement may not be portable across all architectures, so machine-specific methods will also be considered. Finally, we are interested in examining how various source-level and peephole optimization technologies can be applied to improving security as opposed to just optimization.

7 Conclusion

Computer system hacking is both an art form and a science as much as virtually any other field of computer science. Identifying vulnerabilities in a program application is a fairly disciplined field that requires high ability and lots of ingenuity. No absolute guarantees can be made about whether a software system will be completely hack-proof It is hoped that this research will derive a technology that is a economically and practically feasible solution to maintaining the security and integrity of an application system. There are expected to be a number of tradeoffs that one would need to consider such as the effect of tighter security on program execution and the likelihood of generating more false positives for attacks detected. The ultimate goal is to provide a set of customizable security mechanisms that a user can choose to best fit her needs, thus helping make the system secure while still leaving it usable. Widespread adoption of a given technique requires a number of features: simplicity in incorporation, backwards-compatibility with existing systems, imperceptible effect on compilation and execution speed, and so on.

GCC is a stable, well-constructed, well-known and widely-accepted compiler technology. Implementing our research ideas on top of this open source compiler system has obviated the need to invest heavily in building a working, comprehensive compiler system from scratch. Use of such tools will enable further research on our results.

Acknowledgements

We would like to thank Alfred Aho for his insightful comments on the approach described in this paper, and Spiros Mancoridis and Vassilis Prevelakis from Drexel University for useful discussions on vulnerabilities and other security issues related to open source projects. Kaiser's Programming Systems Laboratory is funded in part by Defense Advanced Research Project Agency under DARPA Order K503 monitored by Air Force Research Laboratory F30602-00-2-0611, by National Science Foundation grants CCR-02-03876, EIA-00-71954, and CCR-99-70790, and by Microsoft Research. Keromytis is funded in part by DARPA and the Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-01-2-0537. Edwards is funded in part by the National Science Foundation under grant CCR-0133348, under the New York State NYSTAR program, and by Intel Corporation.

References

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

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

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

[4] 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*, Jan. 1998.

[5] M. Eichin and J. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of IEEE Computer Society Symposium on Security and Privacy*, 1989.

[6] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *HotOS-VI*, 1997.

[7] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the USENIX Security Symposium*, 2001.

[8] K. J. Houle, G. M. Weaver, N. Long, and R. Thomas. Trends in denial of service attack technology. Technical report, CERT, 2001.
http://www.cert.org/archive/pdf/DoS_trends.pdf.

[9] 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*, Monterey, California, June 2002.

[10] 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.

[11] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.

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