Countering Code-Injection Attacks With Instruction-Set Randomization

Gaurav S. Kc
Computer Science Dept.
Columbia University
gskc@cs.columbia.edu

Angelos D. Keromytis
Computer Science Dept.
Columbia University
angelos@cs.columbia.edu

Vassilis Prevelakis
Computer Science Dept.
Drexel University
vp@drexel.edu

ABSTRACT

We describe a new, general approach for safeguarding systems against *any* type of code-injection attack. We apply Kerckhoff's principle, by creating process-specific randomized instruction sets (*e.g.,* machine instructions) of the system executing potentially vulnerable software. An attacker who does not know the key to the randomization algorithm will inject code that is invalid for that randomized processor, causing a runtime exception. To determine the difficulty of integrating support for the proposed mechanism in the operating system, we modified the Linux kernel, the GNU *binutils* tools, and the *bochs-x86* emulator. Although the performance penalty is significant, our prototype demonstrates the feasibility of the approach, and should be directly usable on a suitable-modified processor (*e.g.,* the Transmeta Crusoe).

Our approach is equally applicable against code-injecting attacks in scripting and interpreted languages, *e.g.,* web-based SQL injection. We demonstrate this by modifying the Perl interpreter to permit randomized script execution. The performance penalty in this case is minimal. Where our proposed approach is feasible (*i.e.,* in an emulated environment, in the presence of programmable or specialized hardware, or in interpreted languages), it can serve as a low-overhead protection mechanism, and can easily complement other mechanisms.

Categories and Subject Descriptors

D.2.0 [**Protection Mechanisms**]: Software Randomization

General Terms

Security, Performance.

Keywords

Interpreters, Emulators, Buffer Overflows.

1. INTRODUCTION

Software vulnerabilities have been the cause of many computer security incidents. Among these, buffer overflows are perhaps the most widely exploited type of vulnerability, accounting for approximately half the CERT advisories in the past few years [59]. Buffer overflow attacks exploit weaknesses in software that allow them to alter the execution flow of a program and cause arbitrary code to execute. This code is usually inserted in the targeted program, as part of the attack, and allows the attacker to subsume the privileges of the program under attack. Because such attacks can be launched over a network, they are regularly used to break into hosts or as an infection vector for computer worms [55, 4, 9, 10, 46, 64].

In their original form [12], 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 [43, 5] or use other injection vectors (*e.g.,* format strings [6]). Such *code-injection* attacks are by no means restricted to languages like *C*; attackers have exploited failures in input validation of web CGI scripts to permit them to execute arbitrary SQL [7] and unix command line [8] instructions respectively on the target system. There has been some speculation on similar attacks against *Perl* scripts (that is, causing Perl scripts to execute arbitrary code that is injected as part of input arguments). Although the specific techniques used in each attack differ, they all result in the attacker executing code of their choice, whether machine code, shell commands, SQL queries, *etc.* The natural implication is that the attacker knows what "type" of code (*e.g., x86* machine code, SQL queries, unix shell commands) can be injected.

This observation has led us to consider a new, general approach for preventing code-injection attacks, *instruction-set randomization.* By randomizing the underlying system's instructions, "foreign" code introduced by an attack would fail to execute correctly, regardless of the injection approach. *Thus, our approach addresses not only stack- and heap-based buffer overflow attacks, but* **any** *type of code-injection attack.* What constitutes the instruction set to be randomized depends on the system under consideration: common stack or heap-based buffer overflow attacks typically inject machine code that corresponds to the underlying processor (*e.g.,* Intel *x86* instructions). For Perl injection attacks, the "instruction set" is the Perl language, since any injected code will be executed by the Perl interpreter. To simplify the discussion, we will focus on machine code randomization for the remainder of this paper, although we discuss our prototype randomized Perl in Section 3.2.

Randomizing an arbitrary instruction set, *e.g.,* the *x86* machine code, involves three components: the randomizing element, the execution environment (*e.g.,* an appropriately modified processor), and the loader. Where the processor supports such functionality (*e.g.,* the TransMeta Crusoe, some ARM-based systems [56], or other programmable processors), our approach can be implemented without noticeable loss of performance, since the randomization process can be fairly straightforward, as we report in Section 2. We

CCS'03, October 27–30, 2003, Washington, DC, USA.

describe the necessary modifications to the operating system and the randomizing element. We use a modified version of the *bochs-x86 Pentium* emulator to validate our design. Generally, the loss of performance associated with an emulator is unacceptable for most (but not all [62]) applications: we present a small but concrete example of this in Section 3.1.3. Our prototype demonstrates the simplicity of the necessary software support. In the case of interpreted languages, our approach does not lead to any measurable loss in performance. Compared to previous techniques, we offer greater transparency to languages, applications and compilers, as well as a smaller impact on performance.

Paper Organization. The remainder of this paper is organized as follows. Section 2 presents the details of our approach. Section 3 describes two prototype implementations, for *x86* executables and Perl scripts respectively. We discuss some details of the approach, limitations, and future work in Section 4. Section 5 gives an overview of related work aimed at protecting against code-injection attacks or their effects. We conclude the paper in Section 6.

2. INSTRUCTION-SET RANDOMIZATION

Code-injection attacks attempt to deposit executable code (typically machine code, but there are cases where intermediate or interpreted code has been used) within the address space of the victim process, and then pass control to this code. These attacks can only succeed if the injected code is compatible with the execution environment. For example, injecting *x86* machine code to a process running on a SUN/SPARC system may crash the process (either by causing the CPU to execute an illegal op-code, or through an illegal memory reference), but will not cause a security breach. Notice that in this example, there may well exist sequences of bytes that will crash on neither processor.

Our approach leverages this observation: we create an execution environment that is unique to the running process, so that the attacker does not know the "language" used and hence cannot "speak" to the machine. We achieve this by applying a reversible transformation between the processor and main memory. Effectively, we create new instruction sets for each process executing within the same system. Code-injection attacks against this system are unlikely to succeed as the attacker cannot guess the transformation that has been applied to the currently executing process. Of course, if the attackers had access to the machine and the randomized binaries through other means, they could easily mount a dictionary or known-plaintext attack against the transformation and thus "learn the language". However, we are primarily concerned with attacks against *remote services* (*e.g.*, http, dhcp, DNS, and so on). Vulnerabilities in this type of server allow external attacks (*i.e.*, attacks that do not require a local account on the target system), and thus enable large-scale (automated) exploitation. Protecting against internal users is a much more difficult problem, which we do not address in this work.

2.1 Randomization Process

The machine instructions for practically all common CPUs consist of *opcodes* that may be followed by one or more arguments. For example, in the Intel *x86* architecture, the code for the software interrupt instruction is `0xCD`. This is followed by a single one-byte argument which specifies the type of interrupt. By changing the relationship between the op code (`0xCD`) and the instruction (`INT`), we can effectively create a new instruction without affecting the processor architecture.

For this technique to be effective, the number of possible instruction sets must be relatively large. If the randomization process is driven by a key[1], we would like this key to be as large as possible. If we consider a generic CPU with fixed 32-bit instructions (like most popular RISC processors), hardware-efficient randomization techniques would consist of XOR'ing each instruction with the key or randomly (based on the key) transposing all the bits within the instruction, respectively (see Figure 1). An attacker trying to guess the correct key would have a worst-case work factor of 2^{32} and $32!$ for XOR and transposition respectively (notice that $32! >> 2^{32}$).

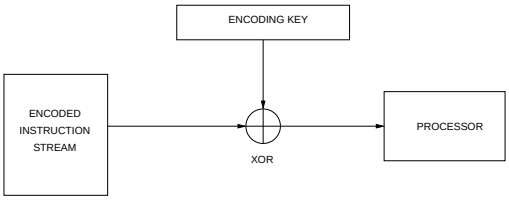

Figure 1: Previously encoded instructions are decoded before being processed by the CPU.

Notice that, in the case of XOR, using a larger block size does not necessarily improve security, since the attacker may be able to attack the key in a piece-meal fashion (*i.e.*, guess the first 32 bits by trying to guess only one instruction, then proceed with guessing the second instruction in a sequence, *etc.*). In any case, we believe that a 32-bit key is sufficient for protecting against code-injection attacks, since the rate at which an attacker can launch these brute-force probing attacks is much smaller than in the case of modern cryptanalysis. Processors with 64-bit instructions (and thus 64-bit keys, when using XOR for the randomization) are even more resistant to brute-force attacks. When using bit-transposition within a 32-bit instruction, we need 160 bits to represent the key, although not all possible permutations are valid (the effective key size is $log_2(32!)$). Increasing the block size (*i.e.*, transposing bits between adjacent instructions) can further increase the work factor for an attacker. The drawback of using larger blocks is that the processor must have simultaneous access to the whole block (*i.e.*, multiple instructions) at a time, before it can decode any one of them. Because we believe this greatly increases complexity, we would avoid this scheme on a RISC processor. Unfortunately, the situation is more complicated on the *x86* architecture, which uses variable-size instructions, as we discuss in Section 3.

Finally, note that the security of the scheme depends on the fact that injected code, after it has been transformed by the processor as part of the de-randomizing sequence, will raise an exception (*e.g.*, by accessing an illegal address or using an invalid op code). While this will generally be true, there are a few permutations of injected code that will result in working code that performs the attacker's task. We believe that this number will be statistically insignificant — the same probability as creating a valid buffer-overflow exploit for a known vulnerability by using the output of a random number generator as the injected code.

2.2 System Operation

Let us consider a typical system with an operating system kernel and a number of processes. Since code-injection attacks usually target applications, rather than the kernel, we consider using this mechanism only when the processor runs in "user" mode. Therefore, the kernel always runs the native instruction set of the proces-

[1]The meaning of the term "key" here is similar to its use in modern cryptography, *i.e.*, the security of the randomization process depends on the entropy and secrecy of a random bit-string.

sor, although it is possible to run the kernel itself under our scheme as well. The only complications relate to interrupt handling, processor initialization, and calling system ROM code. In all of these cases, the operating system is using the native instruction set and can thus handle external code, *e.g.,* ROM. Randomization is in effect only while a process is executing in user-level mode.

The code section of each process is loaded from an executable file stored on some mass-storage device. The executable file contains the appropriate decoding key in a header, embedded there by the randomizing component of our architecture, described in section 2.3. For the time being, we will assume that the executables are statically linked (*i.e.,* there is no code loaded from dynamically linked libraries). We expect only a small number of programs to require static linking, *i.e.,* network services. The decoding key in the program header is associated with the encoding key used for the encoding of the text segment. Specifically, it would be the same when using XOR as the randomization function, and a key specifying the inverse transposition in the second scheme we discussed earlier. When a new process is loaded from disk, the operating system extracts the key from the header, and stores it in the process control block (PCB) structure.

Our approach provides for a special processor register where the decoding key will be stored, and a special privileged instruction (called *GAVL*) that allows write-only access to this register. Thus, when the operating system is ready to schedule the execution of a process, the *GAVL* instruction is used to copy the de-randomization key from the PCB to the decoding register. To accommodate programs that have not been randomized, we provide a special key that, when loaded via the *GAVL* instruction, disables the decoding process. For programs that have not been randomized, the operating system will load the null decoding key in the PCB. Since the key is always brought in from the PCB, there is no need to save its value during context switches. There is thus no instruction to read the value of the decoding register.

As we mentioned earlier, the decoding key is associated with an entire process. It is thus difficult to accommodate dynamically linked libraries, as these would either have to be encoded as they are loaded from disk, or be encoded and copied into a completely disjoint set of memory pages in the case of already memory-resident libraries. In both cases, the memory occupied by the encrypted code for the libraries will not be shareable with other processes, or all the processes would have to share the same key (the one used by the libraries). Since neither approach is appealing, we decided to require statically-linked executables. In practice, we would seek to randomize (and thus statically-link) only those programs that are exposed to remote exploits, *i.e.,* network daemons, thus minimizing the overall impact of static linking to the system. Another way of addressing this problem is to associate keys with segments rather than processes. The processor would then decode instructions based on which segment they were located. We hope to be able to evaluate this approach in the future.

2.3 Randomized ELF Executables

The Executable and Linking Format (ELF) [57] was developed by UNIX System Laboratories and is the standard file format used with the *gcc* compiler, and associated utilities like the assembler and linker in many architectures for encoding executable and library files. The most outstanding innovation of ELF over earlier file formats like *a.out* and *coff* is the complete separation of code and data sections. This was very useful for us to be able to single out the executable sections in an ELF executable so that we could then carry out their block-randomization. At the time this paper was being written, OpenBSD implemented extensions to the *a.out* format that also allow for separation of program text from read-only data (which were lumped together in the `.text` segment). The latest versions of OpenBSD use ELF.

Rather than build a complete ELF parsing and transmutation system from scratch, we found it easier to modify an existing utility that already had the means of performing general-purpose transformations on ELF files. The *objcopy* program that is incorporated into the GNU *binutils* package was chosen for this task. Not only did *objcopy* handle processing the ELF headers, but it also conveniently provided a reference to a byte-array (representing the machine instruction block) for each given code section in the file. We were then able to take advantage of this fact by randomizing each 16-bit block in this array before letting the rest of the original program continue producing the target file. Our modified *objcopy* can process all text sections in an executable in a single pass.

3. IMPLEMENTATION

To determine the feasibility of our approach, we built a prototype of the proposed architecture using the *bochs* emulator [1] for the *x86* processor family. As we discussed in Section 2.1, randomization on the *x86* is more complicated than with other RISC-type processors because of its use of variable-size instructions. However, we decided to implement the randomization for the *x86* both to test its feasibility in a worst-case scenario and because of the processor's wide use.

Recalling our discussion in Section 2.1, we should use the largest possible block size (and corresponding key). However, the *x86* has several 1-byte instructions. A block size of 8 bits is insufficient, as the attacker only needs to try at most 2^{16} different versions of the exploit, when using bit-permutation as the randomizing principle ($8! \approx 2^{16}$) . A 16-bit block size yields an effective key size of 41 bits ($16! \approx 2^{41}$). Thus, we have to ensure that the processor always has access to 16 bits at a time. If we try to pad all instructions with an odd number of bytes with 1-byte NOP (No OPeration) instructions, the code size will be increased and execution time will suffer, as the processor will have to process the additional NOPs.

Our workaround is to ensure that the program always branches to even addresses. The GNU C compiler conveniently allows this to be specified at compile time (using the "-falign-labels=2" command line option, which ensures that all branch-instruction targets are aligned on a 2-byte boundary), but we have also modified the GNU Assembler to force all labels to be aligned to 16-bit boundaries. It does so by appending a NOP instruction, as necessary. Surprisingly, in the few experiments we performed (statically linking the *ctags* utility, with and without the "-falign-labels=2" compilation flag) we did not notice a large increase in size. In particular, there was only a 0.12% increase in the size of *glibc* when compiled with the alignment flag. Examining the produced code revealed that, although new NOPs were added to ensure proper alignment, some NOPs that were present when compiling without the flag were not produced in the second version. Consequently, the performance impact of additional NOPs is minimal.

The decoding is performed when instructions and associated arguments are fetched from main memory. Since the randomization block size is 16 bits, we have to be careful to always start decoding from even addresses. Assuming we start executing code from an even address (which is the default for *gcc*-produced code), the decoding process can proceed without complications as we continue reading pairs of bytes.

This is the technique we used in the original prototype. It has a number of limitations, which include the need to have access to the program source code (since the program has to be compiled with the special alignment flag, and then processed by our modified

assembler) and increased overhead when moving to larger block sizes. Instead, we can use XOR of each 32-bit word with a 32-bit key. The use of this scheme has a major advantage: there is no longer any need for alignment, since the address of the memory access allows us to select which part of the 32-bit key to use for the decoding. In any case, a real processor with a 32-bit or 64-bit data bus will always be performing memory accesses aligned to 32 or 64 bit boundaries (in fact, the Pentium uses a 16-byte "streaming" instruction cache inside the processor), so the decoding process can be easily integrated into this part of the processor. The danger is that the effective key size is not really 32 or 64 bits: many of the "interesting" instructions in the *x86* are 2 bytes long. Thus, an attacker will have to guess two (or four) independent sub-keys of 16 bits each. At first glance, it appears that the work factor remains the same ($2^{32} = 2^{16\times 2}$), it may possible for an attacker to independently attack each of the sub-keys. We intend to investigate the feasibility of such an attack in the future.

3.1 Runtime Environment

Even without a specially-modified CPU, the benefits of randomized executables can be reaped by combining a sandboxed environment that emulates a conventional CPU with the instruction randomization primitives discussed earlier. Such a sandboxing environment would need to include a CPU emulator like *bochs* [1], its own operating system, and the process(es) we wish to protect.

3.1.1 Bochs Modifications

Bochs is an open-source emulator of the *x86* architecture. Since it interprets each machine instruction in software, *bochs* allows us to perform any restoration operations on the instruction bytes as they are fetched from instruction memory. The core of *bochs* is implemented in the function *cpu_loop()* that uses another function, *fetchDecode()*, passing a reference into an array representing a block of instruction code. The *fetchDecode()* function incrementally extracts a byte from that array until it can complete decoding of the current instruction. This behavior closely simulates the *i486* and *Pentium* processors, with their instruction "prefetch streaming buffers". On the *i486*, this buffer held the next 16-bytes worth of instructions; on later processors, this has typically been 32 bytes.

We carry out our de-randomizing of this instruction at the beginning of *fetchDecode()* by restoring the next 16 bytes in the prefetch block so that further processing can proceed normally in actually decoding the correct instruction. To ensure that future calls to *fetchDecode()* execute in the same manner, we reverse the de-randomization at the end of each invocation of *fetchDecode()*. Both the de-randomization and re-randomization are conditionally carried out based on the decoding key value that is currently in the special processor register as discussed in section 2.

3.1.2 Single-System Image Prototype

To minimize configuration and administration overheads, the operating system running within the sandbox should offer a minimal runtime environment and should include a fully automated installation. For our prototype, we adopted the techniques we used to construct embedded systems for VPN gateways [51]. We use automated scripts to produce compact (2-4MB) bootable single-system images that contain a system kernel and applications. We achieve this by linking the code of all the executables that we wish to be available at runtime in a single executable using the `crunchgen` utility. The single executable alters its behavior depending on the name under which it is run (`argv[0]`). By associating this executable with the names of the individual utilities (via file system hard-links), we can create a fully functional `/bin` directory where all the system commands are accessible as apparently distinct files. This aggregation of the system executables in a single image greatly simplifies the randomization process, as we do not need to support multiple executables or dynamic libraries. The root of the run-time file system, together with the executable and associated links, are placed in a RAM-disk that is stored within the kernel binary. The kernel is then compressed (using `gzip`) and placed on a bootable medium (in our case a file that *bochs* considers to be its boot device). This file system image also contains the `/etc` directory of the running system in uncompressed form, to allow easy configuration of the runtime parameters.

At boot time, the kernel is copied from the boot image to *bochs'* main memory, and is uncompressed and executed. The file system root is then located in the RAM-disk. The `/etc` directory copied to the RAM-disk from the temporarily mounted boot partition. The system is running entirely off the RAM-disk and proceeds with the regular initialization process. This organization allows multiple applications to be combined within a single kernel where they are compressed, while leaving the configuration files in the `/etc` directory on the boot partition. Thus, these files can be easily accessed and modified. This allows a single image to be produced and the configuration of each sandbox to be applied to it just before it is copied to this separate boot partition.

3.1.3 Performance

Since our goal was simply to demonstrate the feasibility of our approach, we chose a few, very simple benchmarks. Generally, interpreting emulators (as opposed to virtual machine emulators, such as VMWare) impose a considerable performance penalty; depending on the application, the slow-down can range from one to several orders of magnitude. This makes such an emulator generally inappropriate for high-performance applications, although it may be suitable for certain high-availability environments.

The first two columns in table 1 compare the time taken by the respective server applications to handle some fairly involved client activity. The times recorded for the *ftp* server was for a client carrying out a sequence of common file and directory operations, *viz.*, repeated upload and download of a $\approx 200KB$ file, and creation, deletion and renaming of directories, and generating directory listings by means of an automated script. This script was executed 10 times to produce the times listed. This *ftp* result illustrates how a network I/O-intensive process does not suffer execution time slowdown proportional to the reduction in processor speed. The *sendmail* numbers, taken from the mail server's logging file, represent the overall time taken to receive 100 short e-mails ($\approx 1KB$) from a remote host.

Table 1: Experimental results: execution times (in seconds) for identical binaries on Bochs and a regular Linux machine (the one that hosted the bochs emulator).

	ftp	**sendmail**	**fibonacci**
bochs	39.0s	$\approx 28s$	5.73s (*93s*)
linux	29.2s	$\approx 1.35s$	0.322s

The final column demonstrates the tremendous slowdown incurred in the emulator when running a CPU-intensive application (as opposed to the I/O-bound jobs represented in the first two examples), such as computation of the *fibonacci* numbers. However, this only helps confirm the existence of real-world applications for emulators. Note though that the execution time reported by the emulator itself (5.73 seconds) is actually less than that reported for

executing the *fibonacci* experiment on a real linux machine. The actual wall time for running *fibonacci* in the emulator is 93 seconds. All the applications were compiled with the `-static -falign-labels` option for *gcc*, with zero optimization.

Emulator-based approaches have also been proposed in the context of intrusion and anomaly detection [25, 32], as well as one way to retain backward compatibility with older processors[2] — often exhibiting better performance. However, to make our proposal fully practical, we will need to modify an actual CPU.

3.2 Randomized Perl

In the Perl prototype, we randomized all the keywords, operators, and function calls in a script. We did so by appending a random 9-digit number ("tag") to each of these. For example, the code snippet

```
foreach $k (sort keys %$tre) {
  $v = $tre->{$k};
  die ''duplicate key $k\n''
      if defined $list{$k};
  push @list, @{ $list{$k} };
}
```

by using "123456789" as the tag, becomes

```
foreach123456789 $k (sort123456789 keys %$tre)
{
  $v =1234567889 $tre->{$k};
  die123456789 ''duplicate key $k\n''
    if123456789 defined123456789 $list{$k};
  push123456789 @list, @{ $list{$k} };
}
```

Perl code injected by an attacker will fail to execute, since the parser will fail to recognize the (missing or wrong) tag.

We implemented the randomization by modifying the Perl interpreter's lexical analyzer to recognize keywords followed by the correct tag. The key is provided to the Perl interpreter via a command-line argument, thus allowing us to embed it inside the randomized script itself, *e.g.*, by using "#!/usr/bin/perl -r123456789" as the first line of the script. Upon reading the tag, the interpreter zeroes it out so that it is not available to the script itself via the ARGV array. These modifications were fairly straightforward, and took less than a day to implement. To generate the randomized code, we used the Perltidy [2] script, which was originally used to indent and reformat Perl scripts to make them easier to read. This allowed us to easily parse valid Perl scripts and emit the randomized tags as needed.

One problem we encountered was the use of external modules. These play the role of code libraries, and are usually shared by many different scripts and users. To allow their sharing in randomized scripts, we use two tags: the first is supplied by the user via the command line, as discussed above, while the second is a system-wide key known to the Perl interpreter. We extended the lexical analyzer to accept either of these tags. Using this scheme, the administrator can periodically randomize the system modules, without requiring any action from the users. Also, note that we do not randomize the function definitions themselves. This allows scripts that are not run in randomized mode to use the same modules. Although the size of the scripts increases considerably due to the randomization process, some preliminary measurements indicate that performance is unaffected.

A similar approach can counter attacks against web CGI scripts that dynamically generate SQL queries to a back-end database. Such attacks can have serious security and privacy impact [7]. In such a scenario, we would modify the SQL interpreter along the same lines as we described for Perl, and generate randomized SQL queries in the CGI script. To avoid modifying the database front-end, we can use a validating proxy that intercepts randomized SQL queries. If the queries are syntactically correct (*i.e.*, appropriately randomized), they are de-randomized and passed on to the database.

We can also use randomization with CGI scripts that issue unix shell commands — randomizing the shell interpreter, we can avoid attacks such as [8]. In this scenario, we would also randomize the program names, as shown in the following fictional script:

```
#!/bin/sh
if987654 [ x$1 ==987654 x""; then987654
  echo987654 "Must provide directory name."
  exit987654 1
fi987654
/bin/ls987654 -l $1
exit987654 0
```

In all cases, we must hide low-level (*e.g.*, parsing) errors from the remote user, as these could reveal the tag and thus compromise the security of the scheme. Other applications of our scheme include VBS and other email- or web-accessible scripting languages.

4. FURTHER DISCUSSION

4.1 Advantages

Randomizing network services and scripts not only hardens an individual system against code-injection attacks, but also minimizes the possibility of network worms spreading by exploiting the same vulnerability against a popular software package: such malicious code will have to "guess" the correct key. As we saw in Section 2, the length of the key depends on certain architectural characteristics of the underlying processor, and is typically much shorter than cryptographic keys. Nonetheless, the workload for a worm can increase by 2^{16} to 2^{32}, or even more. Periodically re-randomizing programs (*e.g.*, when the system is re-compiled for open-source operating systems, or at installation time and then periodically by an automated script for binary-only distributions) will further minimize the risk of persistent guessing attacks.

Compared to other protection techniques, our approach offers greater transparency to applications, languages and compilers, none of which need to be modified, and better performance at a fairly low complexity. Based on the ease with which we implemented the necessary extensions on the *bochs* emulator, we speculate that designers of new processors could easily include the appropriate circuitry. Since security is becoming increasingly important, adding security features in processors is seen as a way to increase market penetration. The Trusted Computing Platform Alliance (TCPA) architecture [3] already has some provisions for cryptographic functionality embedded inside the processor, and the latest Crusoe processor includes a DES encryption engine. Although these seem to have been designed with digital-rights management (DRM) applications in mind, it may be possible to use the same mechanisms to enhance security in a different application domain. However, using a full-featured block cipher such as DES or AES in our system is likely to prove too expensive, even if it is implemented inside the processor [37]. In the future, we intend to experiment with the Crusoe TM5800 processor, both to evaluate the feasibility of using the built-in DES engine and to further investigate any issues our approach may uncover, when embedded in the processor.

4.2 Disadvantages

Perhaps the main drawback of our approach as applied to code that is meant to execute on a processor is the need for special support by the processor. In some programmable processors, it is possible to introduce such functionality in already-deployed systems. However, the vast majority of current processors do not allow for such flexibility. Thus, we are considering a more general approach of randomizing software as a way of introducing enough diversity

[2]Recent versions of a popular PDA use a StrongARM processor, while older versions use a Motorola 68K variant.

among different instances of the same version of a piece of popular software that large-scale exploitation of vulnerabilities becomes infeasible. We view that work, which is still in progress, as complementary to the work we presented in this paper.

A second drawback of our approach is that applications have to be statically linked, thus increasing their size. In our prototype of Section 3, we worked around this issue by using a single-image version of OpenBSD. In practice, we would seek to randomize (and thus statically-link) only those programs that are exposed to remote exploits, *i.e.*, network daemons, thus minimizing the overall impact of static linking to the system. Furthermore, it must be noted that avoiding static linking is going to help reduce only the disk usage, not the runtime memory requirements. Each randomized process will need to acquire (as part of process loading) and maintain its own copy of the encrypted libraries using either of the following two mechanisms, neither of which is desirable. Firstly, the process loader can load just the main program image initially, and dynamically copy/load and encrypt libraries *on-demand*. This will incur considerable runtime overhead and also require complex process management logic, in the absence of which it will degrade to the other mechanism, described next. The second approach is to load and encrypt the program code, and all libraries that are referenced, right at the beginning. It is obvious that this will result in large amounts of physical RAM being used to store multiple copies of the same code, some of which may never be executed during the entire life of the process.

Also note that our form of code randomization effectively precludes polymorphic and self-modifying code, as the randomization key and relevant code would then have to be encoded inside the program itself, potentially allowing an attacker to use them.

Instruction randomization should be viewed as a self-destruct mechanism: the program under attack is likely to go out of control and be terminated by the runtime environment. Thus, this technique cannot protect against denial of service attacks and should be considered as a safeguard of last resort, *i.e.*, it should be used in conjunction with other techniques that prevent vulnerabilities leading to code-injection attacks from occurring in the first place.

Debugging is made more difficult by the randomization process, since the debugger must be aware of it. The most straightforward solution is to de-randomize the executable prior to debugging; the debugger can do this in a manner transparent to the use, since the secret key is embedded in the ELF executable. Similarly, the debugger can use the key to de-randomize core dumps.

Finally, our approach does not protect against all types of buffer overflow attacks. In particular, overflows that only modify the contents of variables in the stack or the heap and cause changes in the control flow or logical operation of the program cannot be defended against using randomization. Similarly, our scheme does not protect against attacks that cause bad data to propagate in the system, *e.g.*, not checking for certain Unix shell characters on input that is passed to the *system()* call. None of the systems we examined in Section 5 protect against the latter, and very few can deter the former type of attack. A more insidious overflow attack would transfer the control flow to some library function (*e.g., system()*). To defend against this, we propose to combine our scheme with randomizing the layout of code in memory at the granularity of individual functions, thus denying to an attacker the ability to jump to an already-known location in existing code.

5. RELATED WORK

In the past, encrypted software has been proposed in the context of software piracy, treating code as content (see digital rights management). Users were considered part of the threat model, and those approaches focused on the complex task of key management in such an environment (typically requiring custom-made processors with a built-in decryption engine such as DES [47]). Our requirements for the randomization process are much more modest, allowing us to implement it on certain modern processors, emulators, and interpreters. However, we can easily take advantage of any built-in functionality in the processor that allows for encrypted software (*e.g.*, the Transmeta Crusoe TM5800 processor).

PointGuard [22] 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 [18], 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. This approach can be effectively combined with instruction randomization to offer comprehensive protection against all memory-corrupting attacks.

Forrest et al. [27] 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.

There has been considerable research in preventing buffer overflow attacks. Broadly, it can be classified into four categories: safe languages and libraries, source code analysis, process sandboxing, and, for lack of a better term, compiler tricks.

Safe Languages and Compilers Safe languages, (*e.g.*, Java) 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. In particular, most widely-used operating systems (Windows, Linux, *BSD, *etc.*) are written in C and are unlikely to be ported to a safe language anytime soon. Furthermore, learning a new language is often considered a considerable barrier to its use. Java has arguably overcome this barrier, and other safe languages that are more C-like (*e.g.*, Cyclone [35]) 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 programmable embedded systems.

One step toward the use of safe constructs in unsafe languages is the use of "safe" APIs (*e.g.*, the strl*() API [44]) and libraries (*e.g.*, *libsafe* [16]). 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.

[14] describes some design principles for safe interpreters, with a focus on JavaScript. The Perl interpreter can be run in a mode that implements some of these principles (access to external interfaces, namespace management, *etc.*). While this approach can somewhat mitigate the effects of an attack, it cannot altogether prevent, or even contain it in certain cases (*e.g.*, in the case of a Perl CGI script that generates an SQL query to the backend database).

Source Code Analysis Increasingly, source code analysis techniques are brought to bear on the problem of detecting potential code vulnerabilities. The most simple approach has been that of the compiler warning on the use of certain unsafe functions, *e.g.*, *gets()*. More recent approaches [28, 59, 39, 54] 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 [41, 40]) offer incomplete protection, as they can only protect against and detect *known* classes of attacks and vulnerabilities. MOPS [20] 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. CCured [48] 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%.

Process Sandboxing Process sandboxing [49] is perhaps the best understood and widely researched area of containing bad code, as evidenced by the plethora of available systems like Janus [34], Consh [13], Mapbox [11], OpenBSD's *systrace* [53], and the Mediating Connectors [15]. These operate at user level and confine applications by filtering access to system calls. To accomplish this, they rely on *ptrace(2)*, the */proc* file system, and/or special shared libraries. Another category of systems, such as Tron [17], SubDomain [23] and others [30, 33, 60, 45, 61, 42, 52], go a step further. They intercept system calls inside the kernel, and use policy engines to decide whether to permit the call or not. The main problem with all these is that the attack is not prevented: rather, the system tries to limit the damage such code can do, such as obtain superuser privileges. Thus, the system does not protect against attacks that use the subverted process' privileges to bypass application-specific access control checks (*e.g.*, read all the pages a web server has access to), nor does it protect against attacks that simply use the subverted program as a stepping stone, as is the case with network worms. [31] identifies several common security-related problems with such systems, such as their susceptibility to various types of race conditions.

Another approach is that of program shepherding [38], 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* [16], which dynamically rewrites executed binaries to add run-time return-address checks, thus imposing a significant overhead.

Compiler Tricks Perhaps the best-known approach to countering buffer overflows is Stack Guard [24]. 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 [19], which has since been amended to address the problem.

A similar approach [36], 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. Stack Shield [58] 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 [50] is very similar in that it 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 solely on the return address. ProPolice [26], 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 [24] 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 [29] 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. 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. FormatGuard [21] 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.

Generally, these approaches have three limitations. First, the performance implications (at least for some of them) are non-trivial. Second, they do not seem to offer sufficient protection against stack-smashing attacks on their own, as shown in [19, 63] (although

work-arounds exist against some of the attacks). Finally, they do not protect against other types of code-injection attacks, such as heap overflows [43]. Our goal is to develop a system that can protect against any type of code-injection attack, regardless of the entry point of the malicious code, with minimal performance impact.

6. CONCLUSIONS

We described our *instruction-set randomization* scheme for countering code-injection attacks. We protect against **any** type of code-injection attacks by creating an execution environment that is unique to the running process. Injected code will be invalid for that execution environment, and thus cause an exception. This approach is equally applicable to machine-code executables and interpreted code. To evaluate the feasibility of our scheme, we built two prototypes, using the *bochs* emulator for the *x86* processor family, and the Perl interpreter respectively. The ease of implementation in both cases leads us to believe that our approach is feasible in hardware, and offers significant benefits in terms of transparency and performance, compared to previously proposed techniques. Furthermore, the operating system modifications were minimal, making this an easy feature to support. In the future, we plan to evaluate our system by modifying a fully-programmable processor.

Admittedly, our solution does not address the core issue of software vulnerabilities, which is the bad quality of code. Given the apparent resistance to the wide adoption of safe languages, and not foreseeing any improvement in programming practices in the near future, we believe our approach can play a significant role in hardening systems and invalidating the "write-once exploit-everywhere" principle of software exploits.

7. ACKNOWLEDGEMENTS

We thank Alfred Aho for his insightful comments on our approach and Spiros Mancoridis for useful discussions on vulnerabilities and other security issues related to open source projects.

8. REFERENCES

[1] Bochs Emulator Web Page. `http://bochs.sourceforge.net/`.

[2] The Perltidy Home Page. `http://perltidy.sourceforge.net/`.

[3] Trusted Computing Platform Alliance. `http://www.trustedcomputing.org/`.

[4] 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.

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

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

[7] CERT Vulnerability Note VU#282403. `http://www.kb.cert.org/vuls/id/282403`, September 2002.

[8] CERT Vulnerability Note VU#496064. `http://www.kb.cert.org/vuls/id/496064`, April 2002.

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

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

[11] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 9th USENIX Security Symposium*, pages 1–17, August 2000.

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

[13] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, December 1998.

[14] V. Anupam and A. Mayer. Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies. In *Proceedings of the 7th USENIX Security Symposium*, pages 187–200, January 1998.

[15] R. Balzer and N. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, June 1999.

[16] 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.

[17] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX Technical Conference*, January 1995.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Security for Server Appliances. In *Proceedings of the 14th USENIX System Administration Conference (LISA 2000)*, March 2000.

[24] 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.

[25] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

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

[27] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *HotOS-VI*, 1997.

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

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

[30] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[31] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 163–176, February 2003.

[32] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 191–206, February 2003.

[33] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 39–52, June 1998.

[34] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Procedings of the 1996 USENIX Annual Technical Conference*, 1996.

[35] 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, Monterey, California, June 2002.

[36] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *3rd International Workshop on Automated Debugging*, 1997.

[37] A. D. Keromytis, J. L. Wright, and T. de Raadt. The Design of the OpenBSD Cryptographic Framework. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[38] 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.

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

[40] 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.

[41] 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.

[42] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 29–40, June 2001.

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

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

[45] T. Mitchem, R. Lu, and R. O'Brien. Using Kernel Hypervisors to Secure Applications. In *Proceedings of the Annual Computer Security Applications Conference*, December 1997.

[46] 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.

[47] National Bureau of Standards. Data Encryption Standard, January 1977. FIPS-46.

[48] 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.

[49] D. S. Peterson, M. Bishop, and R. Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002.

[50] 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.

[51] V. Prevelakis and A. D. Keromytis. Drop-in Security for Distributed and Portable Computing Elements. *Internet Research: Electronic Networking, Applications and Policy*, 13(2), 2003.

[52] V. Prevelakis and D. Spinellis. Sandboxing Applications. In *Proceedings of the USENIX Technical Annual Conference, Freenix Track*, pages 119–126, June 2001.

[53] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, August 2003.

[54] 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.

[55] E. H. Spafford. The Internet Worm Program: An Analysis. Technical Report Technical Report CSD-TR-823, Purdue University, West Lafayette, IN 47907-2004, 1988.

[56] Technology Quarterly. Bespoke chips for the common man. *The Economist*, pages 29–30, 14-20 December 2002.

[57] Tool Interface Standards Committee. Executable and Linking Format (ELF) specification, May 1995.

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

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

[60] K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman. Confining root programs with domain and type enforcement. In *Proceedings of the USENIX Security Symposium*, pages 21–36, July 1996.

[61] R. N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 15–28, June 2001.

[62] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and*

Implementation (OSDI), December 2002.

[63] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Intrusion Prevention. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 123–130, February 2003.

[64] 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.