Using DISE to Protect Return Addresses from Attack*

Marc L. Corliss E Christopher Lewis Amir Roth

Department of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104
{*mcorliss,lewis,amir*}*@cis.upenn.edu*

Abstract

Stack-smashing by buffer overflow is a common tactic used by viruses and worms to crash or hijack systems. Exploiting a bounds-unchecked copy into a stack buffer, an attacker can—by supplying a specially-crafted and unexpectedly long input—overwrite a stored return address and trigger the execution of code of her choosing. In this paper, we propose to protect code from this common form of attack using dynamic instruction stream editing (DISE), a previously proposed hardware mechanism that implements binary rewriting in a transparent, efficient, and convenient way by rewriting the dynamic instruction stream rather than the static executable. Simply, we define productions (rewriting rules) that instrument program calls and returns to maintain and verify a "shadow" stack of return addresses in a protected region of memory. When invalid return addresses are detected, the application is terminated.

The DISE implementation resembles previous software schemes like StackGuard and the Return Address Defender (RAD), but it can operate without source code and in dynamically-linked libraries and dynamically-generated code. It also has natural facilities for protecting the shadow stack, which provides little security if it itself is vulnerable. Finally, unlike software instrumentation, DISE checks—which are inserted by the processor at runtime—cannot be bypassed or subverted.

*The work was funded in part by NSF grant CCR-03-11199. Amir Roth is supported by NSF CAREER Award CCR-0238203, and E Lewis is supported by NSF CAREER Award CCF-0347290.

1 Introduction

Buffer-overflow security vulnerabilities represent the largest share of CERT advisories over the past six years [15]. The simplest and most common exploit overflows a stack-resident buffer for the purpose of overwriting the current function's return address and allowing the attacker to redirect execution to arbitrary code at the protection level of the compromised program. This code may be supplied by the attacker via the stack-resident buffer (*stack smashing* [1]) or already exist in a library (*return-to-libc* or more generally *arc injection* [15]).

Several techniques have been proposed to prevent these attacks by protecting return addresses [2, 4, 9, 11]. In this paper, we propose and evaluate a new hardware-assisted implementation that uses *dynamic instruction stream editing* (DISE) [6]. DISE is a one-to-many instruction macro-expander; it resembles the CISC-to-RISC decoder present in IA-32 processors in structure and operation [10, 12], but has programmable rewriting rules. DISE inspects every fetched instruction and potentially rewrites it, feeding the execution engine a modified instruction stream. Although the present work uses DISE to detect attacks, DISE is a general tool for customizing application execution and has been used for profiling [5], memory safety checking [6], code decompression [7], and debugging [8]. These customizations would otherwise be implemented statically via a compiler or binary rewriter or dynamically via *ad hoc* hardware. DISE is a hybrid that combines the flexibility of binary rewriting and the trans-

parency and performance of custom hardware.

We use DISE to protect return addresses by defining productions (rewrite rules) that instrument call and return instructions to maintain a "shadow" stack of return addresses in a protected region of memory. The return instrumentation also verifies that the intended return address is identical to the address stored in the shadow stack by the corresponding call. If it is not, the program is terminated. Although logically similar to other schemes, a DISE implementation has several advantages. (i) It is transparent; programs for which source code is unavailable can be protected, including shared/dynamically-linked libraries and dynamically-generated code. (ii) It is not subvertible; the hardware inserts the instrumentation and ensures that it is not bypassed. (iii) It is efficient; there is no static rewriting overhead or instruction memory footprint. (iv) It has a conceptually-simple, concise, declarative interface; DISE "patches" require only a few lines of code, may be easily extended to guard against new forms of attacks, and are less vulnerable to complexity-related bugs.

The remainder of this paper is organized as follows. The next section summarizes prior techniques for protecting return addresses from stack-smashing attacks. Sections 3 and 4 describe DISE and the implementation of return-address protection. Section 5 presents experimental results.

2 Related Work

A number of techniques for protecting return addresses from buffer-overflow attacks have been proposed and implemented. The most general modify the C compiler to generate code that maintains runtime location and size information for each allocated object and checks that object bounds are respected [14, 16]. Unfortunately, this approach introduces considerable overhead (up to an order of magnitude) and may result in false positives. Libsafe [2] is a new version of the C standard library that uses stack inspection to infer the maximum buffer sizes associated with pointers to stack buffers and ensures that this maximum is not exceeded. Libsafe thwarts buffer-overflow attacks only for stack buffers and only *by* standard library data-copy functions (*e.g.*, **strcpy()**). This far less general solution catches a common form of attack, does not require compiler support (although relinking may be necessary), and is quite efficient.

Alternatively, one may forgo general buffer-overflow detection and focus on detecting return-address corruption, a key component of most buffer-overflow attacks. StackGuard [9] modifies the C compiler to generate function prolog/epilog sequences that either (a) insert a "canary" word before the return address stored in each stack frame or (b) dynamically update virtual memory page permissions to prevent the stack-resident return address from being written. If a buffer overflows across a return address, it (likely) corrupts the canary in the former case or triggers a VM trap in the latter.

Libverify [2], Return Address Defender (RAD) [4], and StackGhost [11] record the return address on a parallel, protected shadow stack at function entry and compare it to the return address on the user stack at function exit, terminating the program if there is a discrepancy. Libverify transforms the program via binary rewriting, while RAD is integrated into the C compiler. StackGhost is a SPARC-specific implementation that hooks into the register-window overflow/underflow handlers to identify and check return addresses without transforming the program binary.

3 DISE

In this section, we give an overview of DISE, focusing on aspects relevant to return-address protection. Full descriptions are available elsewhere [6]. The DISE hardware has two components. The *engine* is similar to the CISC-to-RISC decoder in IA-32 processors [10, 12]. It inspects and potentially rewrites every fetched instruction, feeding the execution engine an instruction stream with enhanced functionality. The *controller* is a gateway through which the engine is dynamically programmed with customization-specific *productions*. Only the OS has access to the controller.

Basic functionality. DISE performs *instruction*

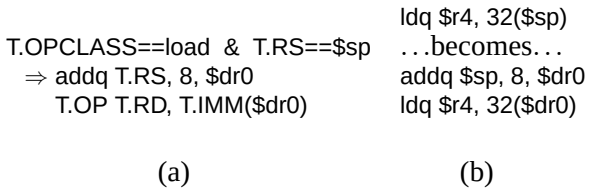

Figure 1: Production example (a) and its use (b).

pattern matching and *parameterized instruction sequence replacement*. A pattern may specify any aspect of a single instruction: opcode, register, *etc.* An instruction that matches a pattern (called a *trigger*) is replaced in the dynamic instruction stream by the corresponding replacement sequence. Replacement sequences are parameterized, so they can be thought of as templates in which some fields are literal and others are instantiated using fields from the trigger. Figure 1 shows a contrived DISE production that adds eight bytes to the address of every load that uses the stack pointer as its base address. Part (a) shows the production. In the replacement sequence, **T.OP**, **T.RS**, **T.RD**, and **T.IMM** are directives to fill the corresponding holes with the trigger's opcode, source register, destination register, and immediate, respectively. Part (b) shows the expansion of a particular load by this production.

Two useful features. DISE has two features that facilitate the orchestration of global behavior from "peephole," single-instruction expansions. A dedicated register set, accessible only to replacement instructions, provides inserted customization code with fast storage—both for intra-sequence temporaries and for passing values from one dynamic replacement sequence to a future one—without the need to save/restore or reserve application registers. The DISE registers have the same basic status as the MIPS exception co-processor registers; they are visible only from the right context, *i.e.*, from within DISE replacement sequences. In Figure 1 **$dr0** is a dedicated DISE register. In examples, we often use mnemonics for DISE registers (*e.g.*, **$dssp** for DISE shadow stack pointer), but they do not have predefined uses.

DISE replacement sequences may also contain control flow: conditionals, loops, and even function calls. However, all replacement sequence control flow must be internal. A dynamic replacement sequence must appear to *atomically* replace the corresponding original program trigger. There is no way to jump into the middle of a replacement sequence from another point in the program. This atomicity, which is enforced in hardware, is important for security-related customizations; it ensures that checks inserted by DISE cannot be subverted or bypassed.

DISE functions and function calls. DISE internal control transfer (*i.e.*, intra-replacement-sequence control flow) is possible but inefficient [6]. If a replacement sequence requires complex control logic, it is usually better to implement this logic in a function using conventional instructions and to call this function from within the replacement sequence. The DISE replacement ISA includes a conditional-call instruction (**ccall**) to efficiently support functions that are called only infrequently. DISE itself is disabled within the body of a function called from within a replacement sequence, preserving the invariant that DISE transformations are "flat," *i.e.*, that replacement sequences are not recursively (and potentially infinitely) expanded.

Functions called from within DISE—which we call *DISE functions*—use conventional instructions, but are different from conventional functions in several ways. First, DISE functions can use three instructions that are not accessible to conventional code: **dmfr** (DISE move from register), **dmtr** (DISE move to register), and **dret** (DISE return). These instructions move DISE register values to and from the standard set of registers (analogous to the **mfc0** and **mtc0** instructions in MIPS). They are only legal if the processor is currently in DISE mode, *i.e.*, within a replacement sequence or a function called from within a replacement sequence. Second, because DISE calls are not orchestrated by the compiler and do not use the standard calling conventions, DISE functions must be written as if all non-DISE registers are callee-saved.

DISE address space. DISE operates in the virtual address space of the application which it modifies. This arrangement reduces overhead and meshes conveniently with multiprogramming (DISE state is automatically saved on a context-switch), but does require some loading/linking initialization

steps. At load time, DISE functions are loaded into the application's address space; their addresses are hard coded into the productions that call them prior to the latter being loaded into the DISE engine. DISE is also allocated a small, fixed-size global memory area in which it stores initial state. If more memory is needed at runtime, DISE can simply call **malloc()** from within one of its own DISE functions.

4 DISE Return Address Protection

We describe a DISE implementation of a mechanism for protecting return addresses from stack-resident buffer-overflow attacks. The basic functional design is simple. We maintain a heap-based shadow stack that mirrors the return addresses stored in the call stack. At each function return, we check that the actual return address matches the address on top of the shadow stack and alert the OS on a mismatch. Although the approach is not new—it resembles several existing techniques [2, 4, 11]—the DISE implementation is conceptually simpler, more efficient, and more secure (*i.e.*, less vulnerable). It can also operate on legacy code, dynamically linked code, and even dynamically generated code. There is also a software-distribution advantage. A patch distributed in "DISE" form can be applied transparently to all applications. The equivalent software patches must be distributed and applied on an application or DLL basis.

Our implementation requires two (or three) productions for rewriting calls, returns, and (potentially) stores. We also load two DISE functions, **addrcheck()** and **expand()**. Finally, we use three dedicated DISE registers which we refer to mnemonically as **$dssb** (shadow stack base), **$dssp** (shadow stack pointer), and **$darp** which points to the top of the currently allocated shadow stack region.

Maintaining the shadow stack. Shadow-stack management is performed by productions for call (**jsr** and **bsr**) and return (**ret**) instructions. The replacement sequence in the call production (top of Figure 2) computes the return address using the trigger's own program counter, pushes it onto the shadow stack, checks for shadow-stack overflow calling **expand()** if necessary, and performs the original call (**T.INST**). If **expand()** is called, it will allocate a larger stack region, copy the old stack into the new buffer, and update the DISE registers to reflect the new location. The return replacement sequence (bottom) pops the shadow stack and performs the original return (again, **T.INST**).

```
T.OPCLASS==jsr|bsr                  # match call insns
  ⇒ add T.PC, 4, $dr0               # compute ret addr
    xor $dr0, $dxr, $dr0            # encode it (optional)
    add $dssp, 16, $dssp            # push it on...
    stq $dr0, -8($dssp)             # ... shadow stack
    stq $sp, -16($dssp)             # ... along w/ stack ptr
    cmpeq $dssp, $darp, $dr0        # stack full?
    ccall $dr0, expand              # yes, expand stack
    T.INST                          # perform the call

T.OPCLASS==ret                      # match return insns
  ⇒ ldq $dr0, -8($dssp)             # pop ret addr...
    add $dssp, -16, $dssp           # ... from shadow stack
    xor $dr0, $dxr, $dr0            # decode it (optional)
    cmpne T.RS, $dr0, $dr0          # comp. to actuar ret addr
    ccall $dr0, addrcheck           # diff? figure out why
    T.INST                          # perform the return
```

Figure 2: DISE productions for return-address verification. Instructions implementing shadow-stack maintenance and return-address verification are emboldened.

Verifying return addresses. In addition to popping the shadow stack, the return production verifies that the return address matches the address at the top of the shadow stack. The replacement sequence compares the popped address to the address specified by the return's source register (**T.RS**). On a match—this is the common case—the original return instruction is executed. On a mismatch, the **addrcheck()** function is called.

Normally, **addrcheck()** will terminate the program because address mismatch indicates tampering. However, there are circumstances in which return-address mismatches are legal. The use of non-local returns (*e.g.*, exceptions or **setjmp()**/**longjmp()**) will cause the system to falsely report a corrupted return address. Previous systems [4, 11] handled these situations by repeatedly popping the shadow stack until an address match is obtained, terminating the program only when the shadow stack underflows. This solution has two drawbacks. First, it allows the shadow and

```
T.OPCLASS==store              # match store instructions
  ⇒ lda $dr0, T.IMM(T.RS1)    # compute target addr
    srl $dr0, 26, $dr0        # get segment of addr
    cmpeq $dr0, $dsr, $dr0    # comp. to shad. stack seg.
    ctrap $dr0, error         # trap if equal
    T.INST                    # original store
```

Figure 3: DISE production to protect the shadow stack by monitoring all stores.

runtime stacks to get out of sync when multiple instances of the same call site are active. Second, it does not prevent an attacker from diverting control to arbitrary locations in the call chain; although it would be challenging to exploit this vulnerability, it is certainly possible. We solve this problem by pushing the current stack pointer (**$sp**) along with the return address onto the shadow stack. On a return-address mismatch, we repeatedly pop shadow stack entries until the return address and stack pointer *both* match (note that this is performed by code in **addrcheck()**). We depend on the fact that the stack pointer itself, which is incremented and decremented but not stored in memory, cannot be smashed and can be reliably used to identify the calling context of a function and thus distinguish benign non-local returns from malicious ones. If **addrcheck()** recognizes a non-local return and returns to the replacement sequence without terminating the program, the actual return instruction (**T.INST**) is executed and program execution continues.

Protecting the shadow stack itself. These DISE productions ensure that actual return addresses correspond to those in the shadow stack, so it is essential that the shadow stack be protected from attack. There are several approaches to achieving this. We may encode addresses in the shadow stack [11], so that even if an attacker manages to corrupt it, she will be unable to divert return destinations unless she is able to duplicate the encoding. We can do this (as in Figure 2) by XORing addresses with a secret, randomly-chosen (at application startup) value. Because this value is held in a DISE register (**$dxr**) it is invisible to the program and the attacker. Alternatively, we may sandwich the shadow stack between two unused, write-protected pages (*e.g.*, via **mprotect()**), thus preventing any buffer from overflowing into it [2, 4].

In DISE, there is a more direct way of protecting the shadow stack. We add a third production that expands all stores and checks that they do not write into DISE's data area. We could achieve this by checking the store address against **$dssb** and **$darp**, terminating the program if it is between them, but this is unnecessarily expensive. Alternatively, we could allocate the shadow stack so that all entries (and only stack entries) share the same set of high-order bits (*i.e.*, segment) and test that the target of all stores do not refer to this segment. The production in Figure 3 uses the latter approach. DISE register **$dsr** holds the segment identifier of the shadow stack. This is logically similar to software-based fault isolation [18] which itself has been implemented in DISE [6]. Note that the stores that update the shadow stack appear in replacement sequences which are not recursively expanded, so they are not checked via the above mechanism. This is the desired effect.

Discussion: Threat/security model. Our implementation targets a common form of attack: return-address smashing or hijacking. It detects this attack by maintaining a non-corruptible shadow stack of return addresses and enforcing correspondence between the shadow and actual return sequences. Our scheme works even if the code pointed to by a hijacked return address is not stack-resident (*e.g.*, a "return-to-libc" attack [15]), scenarios which non-executable stacks [17] fail to protect against. It is important to note that we are not offering protection from general stack or heap buffer overruns, data attacks due to such overruns (*e.g.*, "malloc" attacks), or even control-flow attacks on stored addresses other than return addresses (*e.g.*, "VPTR smashing" attacks [15]). We are currently investigating the use of DISE to protect against these and other attacks.

There are two other security issues that relate to DISE specifically. First, since DISE does not operate in functions called from within replacement sequences, how are functions that are called from within DISE functions—in the way that **malloc()** is called from within **expand()**—protected? The answer is that these functions are not directly protected, because their inputs and environments are

assumed to be safe because they are provided either directly by DISE code (which we assume to be safe) or are passed to DISE from the parts of the application that are themselves protected by DISE. The second issue involves an attacker getting control of DISE itself and programming it with malicious rules. Here the answer is that since only the OS has direct access to the DISE controller, malicious access to the controller implies that the OS has already been compromised.

Discussion: Disadvantages and limitations. We have already enumerated the advantages of dynamic instrumentation via DISE over static approaches, but there are potential disadvantages too. The most significant of these is the inability to optimize the instrumentation code in a larger scope. One particularly powerful optimization that isn't available to DISE is the elimination of redundancies across dynamic instrumentation snippets, which includes hoisting the invariant portion of a snippet out of an enclosing loop. The inability to apply this optimization may result in unnecessary dynamic instruction overhead.

While this shortcoming exists in general, it does not play a significant role for this particular use of DISE. The code snippets that maintain the shadow stack and verify return addresses—i.e., the call/return replacement sequences—do not have redundant portions. There is, however, potential redundancy across dynamic replacement sequences of stores. In general, this redundancy would also be difficult to eliminate statically since doing so requires static address disambiguation. However, there is at least one common scenario in which address disambiguation is easy: stores through the stack pointer. A static framework could avoid instrumenting most stack-pointer stores, or even all of them if it could prove that the stack pointer itself never overlaps with DISE's data area. Fortunately, an analog of this optimization is available to the DISE implementation as well. Specifically, we could implicitly "trust" the stack pointer and only instrument stores with non-stack-pointer base registers. Such trust is warranted because the stack pointer is updated using arithmetic operations and rarely saved to or restored from memory, and thus any smashing of the stack pointer effectively implies that the program has already been compromised.

5 Evaluation

We use cycle-level simulation to evaluate the DISE-based approach to buffer-overflow detection both in terms of effectiveness and performance overhead. First, we describe our simulator and benchmarks.

Apparatus. Our simulator is built using SimpleScalar's Alpha AXP ISA and system call definition modules [3]. We model a 4-way superscalar processor with a 12-stage pipeline and an execution core that resembles an unclustered Alpha 21264 with a 128-entry re-order buffer and 80 reservation stations. The on-chip memory system is composed of 32KB 2-way set-associative instruction and data caches, 64-entry 4-way set-associative instruction and data TLBs, and a 1MB, 4-way set associative L2. The "infinite" main memory has a 200 cycle latency and is access via a 32 byte bus that operates at one quarter processor frequency. Our benchmarks are selected from SPEC, MiBench [13], and CommBench [19]. Other than the SPEC benchmarks, we choose codes that would likely be sensitive to attacks (*e.g.*, those running with root permissions or in a trusted piece of hardware) and thus would benefit from return address protection.

Protection effectiveness. We have identified three vulnerable programs: *overflow1*, *gzip-1.2.4*, and *sendmail-8.7.5*. The first was presented in a hacker's tutorial on buffer-overflow attacks [1] and represents a prototypical vulnerability. The others are versions of well-known codes that are vulnerable to return address hijacking. In all three cases, our DISE implementation successfully detected an input attack and terminated the program. At the same time, non-malicious inputs did not spuriously signal an attack for these three programs or any of the benchmarks used to evaluate performance.

Performance overhead. The additional instructions inserted at function call and return naturally increase program execution time. Figure 4 plots this overhead for two implementations—DISE and software-only binary rewriting (*BR*)—of each of

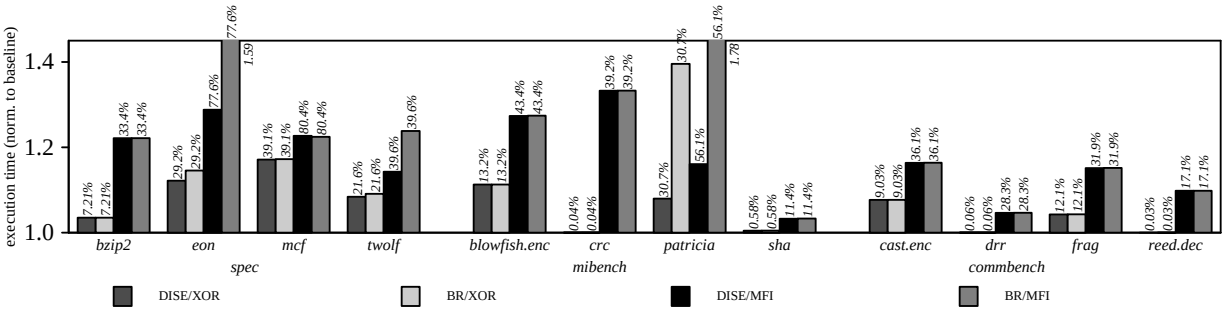

Figure 4: Return-address protection overhead.

two shadow stack protection schemes. The first (*XOR*) encodes the shadow stack (as in Figure 2), while the second (*MFI*) protects it using memory fault isolation (*i.e.*, checks before each store as in Figure 3). The *BR* implementations contain exactly the same instructions as their DISE counterparts (after dynamic instrumentation) because the former are not statically optimized. As discussed in Section 4, return-address protection offers little opportunity for static optimization. There are four total experiments. Each bar represents execution time normalized to a baseline with no return-address protection. Above each bar is the instrumentation overhead in terms of instruction count.

DISE/XOR overhead is generally low (*e.g.*, less than 10%). It is primarily a function of average dynamic function size (equivalently, call/return frequency). Programs with long-running functions and few calls (*e.g.*, *crc*, *drr*, and *reed.dec*) have less overhead than ones with many calls to shorter-running functions (*e.g.*, *mcf*, *twolf*, *blowfish.enc*). At first glance, it would seem that the relative overheads of *blowfish.enc* and *patricia* are anomalous: instruction overhead is higher for *patricia* but execution time overhead is higher for *blowfish.enc*. This is due to a secondary effect: baseline IPC. *Blowfish.enc* has a high baseline IPC that nearly saturates the machine, so additional instructions are expensive. *Patricia* has a low IPC and a lot of "free" execution bandwidth; here additional instructions are relatively cheaper. It is important to note that return-address protection instrumentation does not add to the execution critical path of the protected application—there are no dependence edges that flow from instrumentation code to application code—so that when pipeline utilization is low, they are effectively "soaked up" by the available resources.

DISE/MFI adds significantly more instructions—three additional instructions are needed for every store—and can result in higher overheads for store-heavy benchmarks (*e.g.*, 33% on *crc*). Overheads in general, however, are still reasonable at less than 20%.

The BR implementations generally perform worse than their DISE counterparts. This difference is due to reduced instruction cache capacity and a corresponding increase in miss rate. For benchmarks with large instruction footprints (*e.g.*, *patricia*, *eon*, *twolf*), DISE can dramatically outperform BR. This performance advantage is beyond the previously stated, less concrete advantages of flexibility, transparency, non-subvertibility, and ease-of-use.

6 Conclusion

Buffer-overflow attacks that hijack program control by overwriting a return address stored in a stack frame are common. Existing techniques for detecting these attacks are effective but often inconvenient, difficult to implement, and inefficient. We argue that the dynamic instruction stream editor (DISE) is a useful infrastructure for building return-address protection mechanisms. DISE is simple, non-subvertible, flexible, transparent, and inexpensive. The general utility of DISE in the security domain remains an open question. We are experimenting with DISE instrumentation for detecting and thwarting other forms of attack.

References

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

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

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin–Madison Computer Sciences Department, 1997.

[4] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of 21st Int. Conf. on Distributed Computing Systems*, Apr. 2001.

[5] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: Dynamic instruction stream editing. Technical Report MS-CIS-02-24, University of Pennsylvania, Jul. 2002.

[6] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *Proc. 30th Intl. Symp. on Computer Architecture*, Jun. 2003.

[7] M. L. Corliss, E. C. Lewis, and A. Roth. A DISE implementation of dynamic code decompression. In *Proc. of Conf. on Languages, Compilers, and Tools for Embedded Systems*, pages 232–243, Jun. 2003.

[8] M. L. Corliss, E. C. Lewis, and A. Roth. Low-overhead debugging via flexible dynamic instrumentation. Technical Report MS-CIS-04-06, University of Pennsylvania, Mar. 2004.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention buffer overflow attacks. In *Proc. of 7th USENIX Security Conference*, pages 63–78, Jan. 1998.

[10] K. Diefendorf. K7 challenges Intel. *Microprocessor Report*, 12(14), Nov. 1998.

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

[12] P. Glaskowsky. Pentium 4 (partially) previewed. *Microprocessor Report*, 14(8), Aug. 2000.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of 4th Annual IEEE Workshop on Workload Characterization*, Dec. 2001.

[14] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. of the Int. Workshop on Automatic Debugging*, pages 13–26, May 1997.

[15] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, Jul./Aug. 2004.

[16] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proc. of the 11th Network and Distributed Systems Security Symposium*, Feb. 2004.

[17] Solar Designer. Linux kernel patch from the openwall project. *http://www.openwall.com/linux/*, 2004.

[18] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of 14th ACM Symp. on Operating Systems Principles*, Dec. 1993.

[19] T. Wolf and M. Franklin. CommBench – a telecommunications benchmark for network processors. In *Proc. of IEEE Int. Symp. on Performance Analysis of Systems and Software*, Apr. 2000.