Protecting C Programs from Attacks via Invalid Pointer Dereferences

Suan Hsi Yong and Susan Horwitz
Dept. of Computer Sciences, University of Wisconsin-Madison
1210 W. Dayton St., Madison WI 53706 USA.
{suan,horwitz}@cs.wisc.edu

ABSTRACT

Writes via unchecked pointer dereferences rank high among vulnerabilities most often exploited by malicious code. The most common attacks use an unchecked string copy to cause a buffer overrun, thereby overwriting the return address in the function's activation record. Then, when the function "returns", control is actually transferred to the attacker's code. Other attacks may overwrite function pointers, setjmp buffers, system-call arguments, or simply corrupt data to cause a denial of service.

A number of techniques have been proposed to address such attacks. Some are limited to protecting the return address only; others are more general, but have undesirable properties such as having a high runtime overhead, requiring manual changes to the source code, or forcing programmers to give up control of data representations and memory management.

This paper describes the design and implementation of a security tool for C programs that addresses all these issues: it has a low runtime overhead, does not require source code modification by the programmer, does not report false positives, and provides protection against a wide range of attacks via bad pointer dereferences, including but not limited to buffer overruns and attempts to access previously freed memory. The tool uses static analysis to identify potentially dangerous pointer dereferences, and memory locations that are legitimate targets of these pointers. Dynamic checks are then inserted; if at runtime the target of an unsafe dereference is not in the legitimate set, a potential security violation is reported, and the program is halted.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.
Copyright 2003 ACM 1-58113-743-5/03/0009 ...$5.00.

General Terms

Security

Keywords

Security, Buffer overrun, Static analysis, Instrumentation.

1. INTRODUCTION

Writes via unchecked pointer dereferences rank high among vulnerabilities most often exploited by malicious code. The most common attacks use an unchecked string copy to cause a buffer overrun [33]. For example, consider a procedure that uses a pointer to copy an input string into a buffer stored on the stack, incrementing the pointer after copying each character without checking whether the pointer is past the end of the buffer. An attacker can provide a long string that causes the procedure to write past the end of the buffer, overwriting other locations on the stack, including the procedure's return address. In this way, the attacker can replace the procedure's return address with the address of code written by the attacker (stored, for example, in the string buffer that was overwritten, or in an environment variable), so that when the procedure returns, control is transferred to the attacker's code. Details about this kind of attack, commonly known as "stack smashing", have been documented by many authors (e.g., [29]).

A number of techniques have been proposed to address this vulnerability. StackGuard [10, 30] and StackShield [31] add code to a compiled program to detect attacks on the return address. StackShield also implements range checks on the addresses used by calls via function pointers to guard against attacks on those pointers. The advantages of these approaches are that they require no changes to the source code and introduce very little overhead; however, they do not address attacks on other vulnerable locations, such as system-call arguments or setjmp buffers. Because they do not provide general protection against bad pointer dereferences, sophisticated attackers will find ways to get around the protections that they provide.

A more general technique for protecting against attacks via bad pointer dereferences is the use of "fat pointers". At runtime, extra information is maintained for every pointer[1]: the address and size of the object to which it points. Every pointer dereference is instrumented to use that informa-

[1]In this paper, arrays are considered to be pointers, and an array-index expression such as `A[k]` is treated as the equivalent pointer dereference `*tmp`, where `tmp` has been assigned the value `A+k`.

tion to check whether the current value of the pointer is in bounds; if not, an error is reported.

The fat-pointers approach has several weaknesses: It requires a change of data representation, which may be unacceptable to some applications, and it incurs a relatively high runtime overhead. Two ongoing projects address these limitations:

1. Cyclone [17] is a programming language based on C that prevents bad pointer dereferences while still giving programmers the same control over data representations and memory management that C does. However, to gain the benefits of this approach, existing C programs must be converted to Cyclone programs, which is an important barrier to widespread use.

2. The CCured [23, 9] system uses fat pointers, but cuts down on overhead by using static analysis to remove some instrumentation. Experiments in [23] reported that instrumented program ran up to $2\frac{1}{2}$ times slower than the uninstrumented version. However, there are some problems with CCured: it relies on the use of garbage collection (which may be a problem for applications such as real-time applications that depend on programmer-controlled memory management), it terminates the execution of some correct programs, and although the goal is to apply it to existing C programs, some modifications to the source code are required in some cases.

This paper describes a security-enforcement tool for C programs that provides protection against a wide range of attacks through unchecked pointer dereferences, including buffer overruns and attempts to access or free unallocated memory. The runtime overhead is low, no false positives are reported, no programmer annotations or modifications to existing source code are required, and the programmer maintains control over data representations and memory management. The core idea involves using static analysis to identify *unsafe* pointers in the program, as well as the memory locations that can be the legitimate targets of these pointers. The program is then instrumented so that at runtime each byte of memory is tagged to indicate if it is an *appropriate* location that can be pointed to by an unsafe pointer. Instrumentation is also used to check each write via an unsafe pointer dereference and each call to `free`; if the location being written or freed is not tagged *appropriate*, then an error message is issued to indicate a potential security violation, and the program is halted.

The remainder of this paper is organized as follows: Section 2 describes our security-enforcement tool in more detail. Section 3 describes experiments to demonstrate the effectiveness of the tool in detecting attacks, while Section 4 presents experimental results to measure the performance of the tool. Section 5 discusses several techniques that can be used to improve the coverage and performance of the tool. Section 6 discusses related work.

2. THE SECURITY-ENFORCEMENT TOOL

The approach of our tool stems from the same underlying insight used in the fat-pointers approach: At a given moment during program execution, each pointer has an "appropriate" set of referents (locations that are part of the object whose address was most recently written into the pointer), and a *bad pointer dereference* is one where the dereferenced pointer points outside its appropriate set. However, instead of keeping track of the appropriate set for each pointer individually, as is done by fat pointers, the idea is to keep track of a single set of appropriate pointer targets: those locations that may be pointed to by an unsafe pointer. This is implemented at runtime by associating extra information with the pointed-to locations rather than with the pointers. Specifically, we maintain a "mirror" of the memory used by the program, with one bit in the mirror for every byte in memory. That bit specifies whether or not the location is an appropriate target of some unsafe pointer. At runtime, when an unsafe pointer is dereferenced for writing, the mirror of the pointed-to location is checked; if the bits indicate that the location is not an appropriate target, a bad pointer dereference is reported (indicating a possible security violation) and the program is halted.

For example, consider the code shown in Figure 1. The function `f` copies a user-supplied string (pointed to by `src`) into a local array (`buf`) with no bounds-checking. An attacker could cause a buffer overrun by supplying a `src` string longer than 512 bytes in size, to overwrite the function pointer `p`, or the return address field in the function's activation record, with the address of the attacker's code. Both attacks would be detected by our security-enforcement tool: At runtime, the locations that correspond to the `buf` array would be tagged *appropriate* (because the array is the appropriate target of the pointer `dst`), while all other locations would be tagged *inappropriate* (as shown in the figure). When the loop executes, if the `dst` pointer goes out of bounds, the first attempt to write into an *inappropriate* location would be detected, and the program would be halted.

Our tool first performs static analysis to identify the *unsafe* pointers and the locations they might legitimately point to; these locations are called *tracked* locations. It then instruments the program to perform the following actions at runtime:

- Initially, all locations are tagged *inappropriate* in the mirror.
- Tracked globals are marked *appropriate* prior to the top-level call to `main`.
- Every allocation of a tracked location (on the stack or in the heap) is instrumented to change its tag from *inappropriate* to *appropriate*; a tracked static variable is marked *appropriate* the first time its declaration is encountered at runtime.
- Every deallocation of a tracked location is instrumented to set its tag to *inappropriate*.
- Every write via an unsafe pointer dereference and every call to `free` is instrumented to check the tag of the pointed-to location. If the tag is *inappropriate* an error is reported and the program is halted.

2.1 Static Analysis

2.1.1 Motivation

The static analysis that identifies unsafe pointers and tracked locations serves two purposes: minimizing overhead, and maximizing the likelihood of detecting an attack via a bad pointer dereference. If no static analysis were done, Vulnerable Code

```
void f(char *src) {
   FN_PTR p = ...
   char buf[512];
   char *dst = buf;
   do
      *dst++ = *src;
   while (*src++ != '\0')
   *p();
}
```

At Runtime

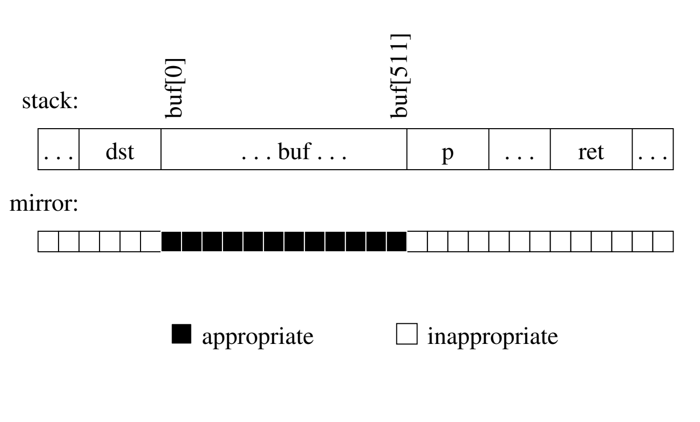


Figure 1: Security violation example.

every memory allocation and free, and every write via a pointer dereference, would have to be instrumented. As this would cause unacceptable overhead, we avoid instrumenting pointer dereferences that are guaranteed never to refer to invalid memory. A pointer is called *unsafe* if it may refer to invalid memory, and is dereferenced for writing or freeing. We use a static analysis to identify the set of unsafe pointers in a program, and only writes via unsafe pointers are instrumented to check for appropriateness.

Furthermore, if all active local variables and heap-allocated memory were tagged *appropriate*, an attack via a bad pointer dereference would only be detected if it involved an attempt to write to non-allocated storage, or to an inappropriate location on the stack (e.g., the return address). While this would still detect stack-smashing attacks, it could miss attacks on other memory locations, (e.g., on a function pointer, a system-call argument, or a `setjmp` buffer). Therefore, we also use static analysis to determine the set of tracked locations: those that may be appropriate targets of an unsafe pointer. Since only writes via unsafe pointer dereferences are instrumented, only the tracked locations need to be tagged *appropriate* at runtime. Every other location is always tagged *inappropriate*; thus, a write via an unsafe pointer that points to any untracked location will cause an error. This allows our tool to detect an attempt to overwrite vulnerable locations such as function pointers, system-call arguments, and `setjmp` buffers, as long as they are not appropriate targets of some unsafe pointer.

Example: In the example code of Figure 1, the function pointer `p` is not itself the target of any pointer; thus, it is not a tracked location, it is tagged *inappropriate* at runtime, and an attempt to write into `p` via a dereference of `dst` is detected by our security tool as an error.

If `p` were the target of a *safe* pointer (e.g., a pointer that only points to `p`), it would still be an untracked location tagged *inappropriate*, and an attempt to write into `p` via `dst` would still be detected as an error.

Only if `p` were the legitimate target of an *unsafe* pointer would it be tagged *appropriate*, causing our tool to miss an overwrite via a different unsafe pointer such as `dst`. □

2.1.2 The Analysis

The static analysis consists of three steps:

Step 1: Do extended points-to analysis.

Step 2: Identify the unsafe pointers.

Step 3: Identify the tracked locations.

Step 1: Extended Points-to analysis. The first step is to perform a slightly non-standard, flow-insensitive points-to analysis. The results of the points-to analysis are used in Step 2 to identify unsafe pointers, and in Step 3 to identify the tracked locations.

The goal of (standard) flow-insensitive points-to analysis is to determine, for each variable `v`, the set of locations that `v` may point to at some time during program execution. (Note that since C allows casting, values can be copied from a pointer to a non-pointer and vice versa. Therefore, points-to sets must be computed for all variables, not just pointers. For example, consider the following code fragment:

```
int x, y;
int *p, *q = &x;
y = (int)q;
p = (int *)y;
```

If no points-to set is computed for `y`, then `x` might be erroneously omitted from `p`'s points-to set.)

The non-standard aspect of our points-to analysis is that a special "bottom" location ($\perp$) is in v's points-to set if v may contain a numeric value other than zero[2] or a computed value (i.e., a value that is the result of applying an arithmetic or bitwise operator to one or more operands). For example, $\perp$ is added to `v`'s points-to set as a result of analyzing any of the following: `v = 3`, `v++`, `v = a+b`, `v = a|b`.

[2]Note that in the context of security, our only concern is to prevent overwriting inappropriate locations. Location zero can never be written into; therefore, a dereference of a pointer that might be NULL (i.e., might have the value zero) does not need to be instrumented, and so zero is considered a valid value for a pointer.

Note that the points-to analysis algorithm will take care of propagating the $\perp$ location from one points-to set to another as a result of assignments. For example, assume that $\perp$ is in `p`'s points-to set, and that `p` is in `q`'s points-to set. Given the assignments `a=p` and `b=*q`, points-to analysis will determine that $\perp$ is also in the points-to sets of both `a` and `b`.

Our implementation uses the near-linear-time points-to analysis defined in [11], extended to handle the special $\perp$ value. Other points-to analyses could be used as well; in general, a more precise points-to analysis requires more time, but would permit smaller sets of unsafe pointers and tracked locations to be identified, thus leading to lower overhead and better coverage.

Example: For the code fragment in Figure 1, points-to analysis would determine that the points-to sets of both `src` and `dst` include $\perp$, because of the expressions `dst++` and `src++`. It would also determine that `buf` is in the points-to set of `dst`. □

Step 2: Identify unsafe pointers. The second step of the static analysis uses the results of points-to analysis to identify unsafe pointers. A pointer variable p is unsafe if:

1. p may refer to invalid memory at runtime, and
2. (a) p is dereferenced for writing, or
   (b) `free`(p) is called.

Criterion 1 is satisfied by any variable whose points-to set includes $\perp$, or includes either a stack variable or a heap location that is `free`d (i.e., p may be a dangling pointer). Criterion 2(a) involves finding assignments in the program where the top-level operator for the left-hand side is a dereference, and 2(b) involves finding calls to `free` in the program. In both cases, the results of points-to analysis are used as necessary to handle multiple levels of dereference. For example, assume that `q` points to `r`. Given the assignments `*p = x` and `**q = *s`, this step would determine that both `p` and `r` (but neither `q` nor `s`) are dereferenced for writing.

Note that since an array-index expression `A[k]` is treated as the equivalent pointer dereference `*tmp`, where `tmp` has been assigned the value `A+k`, the `+` operator makes `tmp` an unsafe pointer; thus, all writes via array-index expressions are considered to be unsafe dereferences. The number of such unsafe dereferences could be reduced by using a more sophisticated analysis, such as array-bounds analysis (see Section 5).

Example: In the running example, `dst` is identified as the only unsafe pointer. (The pointer `src` may also contain an invalid pointer value, but is only dereferenced for reading.) □

Step 3: Identify tracked variables. This step identifies the set of *tracked* variables: variables that may be, at some point during execution, an appropriate target of some unsafe pointer. This is simply the union of the points-to sets of the unsafe pointers.

Example: In the running example, `buf` would be identified as a tracked variable because it is in the points-to set of the unsafe pointer `dst`. □

2.2 Runtime Checks

Once static analysis has identified the set of unsafe pointer dereferences and tracked locations, the tool instruments the program with calls to routines that update and check the tags in the mirror. These routines perform the following tasks:

- At each tracked location's allocation site (local variable declaration or `malloc` callsite), the mirror for each byte in that location is marked *appropriate*. For tracked globals, the mirror is marked prior to the top-level call to `main`.
- At each deallocation site of a tracked location (an exit point of a variable's scope, or a call to `free`), the mirror for each byte of that location is marked *inappropriate*.
- Prior to each write via an unsafe pointer dereference and each call to `free`, the mirror of the pointed-to location is checked; if it is tagged *inappropriate*, then an error message is issued and the program is halted.

The instrumentation engine is implemented as a source-to-source transformer using Ckit [8], a C front end written in ML. Instrumenting at the source level makes our tool portable, as an instrumented source file can be compiled on any platform that supports C.

The runtime routines to set and check tags are implemented as C macros and functions. In our prototype implementation, the mirror consists of 128 KB pages (each page mirroring 1 MB of user memory) which are allocated as the amount of memory in use by the program increases. Pointers to these pages are stored in a table indexed by the most significant 12 bits of the user-space address, so accessing a location's tag is fast. An unallocated mirror page means that the locations mapped to it are all *inappropriate*, so initially (at the start of program execution), with no mirror pages allocated, all memory is *inappropriate* by default.

Example: In the running example, with unsafe pointer `dst` and tracked location `buf`, instrumentation would be added to perform the following tasks:

- When function `f` starts executing, the tags of the elements in the `buf` array would be set to *appropriate*.
- Just before the statement `*dst++ = *src` executes, the tag of the location pointed to by `dst` would be checked; if the tag is *inappropriate* an error message would be issued and the program halted.
- When function `f` returns, the tags of the elements in the `buf` array would be set to *inappropriate*.

□

2.3 Handling Library Functions

Most programming environments include library functions, which may be written in a different language, or for which source code may be unavailable. Our tool supports the arbitrary linking of instrumented modules with uninstrumented ones, though in such cases (i) security violations occurring within the uninstrumented module will not be detected, and (ii) the tool may possibly report false positives (and thus terminate prematurely), if an unsafe pointer dereference in an instrumented module accesses a location allocated in an uninstrumented module. To overcome these problems, a wrapper function and a static model can be supplied for an uninstrumented function to simulate the memory allocation and pointer-access behavior of the function. A wrapper function performs runtime operations to update or check the mirror for appropriateness, while a static model is needed for the static analysis to determine which pointers are unsafe and which locations should be tracked. In general, memory management functions (`malloc`-like functions) and functions returning a pointer to a static buffer must be simulated and modeled to avoid false positives (problem (ii)). Functions that write to a buffer, like `memcpy`, should also be simulated and modeled to ensure that buffer overruns can be detected (problem (i)).

We have written wrappers and models for the standard C library functions that perform memory management (`malloc`, `calloc`, `free`, etc.), write into buffers (`memcpy`, `strcpy`, etc.), or return pointers to static buffers (e.g., `ctime`, `gethostbyname`).

For example, consider the function `memcpy(p1,p2,n)`, which copies `n` bytes of memory from the location pointed to by `p2` to the location pointed to by `p1`. Calls to `memcpy` are replaced by a call to the `memcpy` wrapper function, which checks at runtime the mirror for the first `n` bytes of the location pointed to by `p1`. If any of those `n` bytes is tagged *inappropriate*, an error message is issued and the program is halted; otherwise, `memcpy` itself is called.

When performing static analysis for `memcpy`, the analysis needs to identify all of the locations in parameter `p1`'s points-to set as tracked (because those are the locations whose tags will be checked at runtime by the wrapper function). The model for `memcpy` consists of the single assignment `*p1++ = *p2++`: The increment ensures that $\perp$ is in `p1`'s points-to set, and the use of `*p1` on the left-hand-side of the assignment ensures that it is identified as being dereferenced for writing; thus, `p1` is correctly identified as an unsafe pointer, and all locations in its points-to set will be (correctly) identified as tracked.

By including wrappers and static models for C library functions, our tool can be applied directly and automatically in normal programming environments that only use the standard C libraries. In environments that use custom library modules and components, the user can extend the coverage of the tool by writing wrapper functions and static models for uninstrumented modules.

3. EXPERIMENTS: DETECTING ATTACKS

To demonstrate the efficacy of our tool, we instrumented two Linux (RedHat 6.2) programs with known vulnerabilities and exploits. The first program, **cfingerd**, has a typical buffer-overrun bug vulnerable to stack-smashing; the second program, **traceroute**, has a more subtle bug involving freeing unallocated memory, with an exploit that overwrites an entry in the Global Offset Table (GOT) to gain control of the program. In both cases, our instrumented program detected the erroneous behavior when running the exploit, and halted execution before the exploit was able to gain control of the program. Our instrumented program also ran reliably (reporting no false errors) during normal (non-erroneous) executions.

A description of the two vulnerable programs and exploits follows. Exploits were obtained from the Packet Storm website [25].

3.1 Buffer Overrun in cfingerd 1.4.2

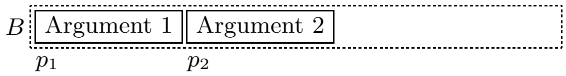
cfingerd [7] is a configurable *finger* daemon that allows each user to turn on or off the ability for others to look up information about them. A user can supply a generic message in a file `.nofinger` in their home directory that will be displayed by **cfingerd**.

Versions 1.4.2 and earlier of **cfingerd** have a buffer overrun bug in the function that processes the data in the `.nofinger` file: the data is read into an 80-byte buffer with no bounds-checking. An attacker can thus use a string longer than 80 characters in their `.nofinger` file to overwrite the function's return address; when the function returns, control is transferred to the attacker's code, which can also be part of the string in the `.nofinger` file. Since by default the **cfingerd** daemon is executed as root, this vulnerability allows an attacker to gain root privileges on the system.

When running a copy of **cfingerd** instrumented with our tool, the attempted attack failed with our tool detecting and reporting an out-of-bounds array write, and halting before the return address could be overwritten.

3.2 Erroneous free in traceroute 1.4a5

traceroute is a utility for collecting information about the path a packet takes when traveling through the internet to its destination. Certain command-line arguments are saved using a function that `calloc`s one large buffer B and returns sub-pieces of memory from B. For example, if given two arguments the function returns p_1 and p_2 pointing to sub-pieces of B containing the two arguments:



After each argument is processed, the program erroneously tries to free the corresponding individual sub-piece: first, it calls `free`(p_1) which unintentionally frees all of B, then it calls `free`(p_2), which usually causes a runtime error. However, this flaw can be exploited by adjusting the values in the memory locations immediately preceding p_2 (the so-called "malloc chunk") to fool `free`(p_2) into thinking p_2 points to allocated memory. In this case, instead of causing a runtime error, `free`(p_2) can be used to copy an arbitrary value into an arbitrary location in memory. This is a very subtle attack that relies on the way the GNU C library version of `free` implements coalescing of free blocks.

The specific exploit we tested uses this mechanism to copy the address of malicious shell-code (supplied as part of the command-line arguments) into the Global Offset Table (GOT), which is a table used to dynamically resolve library function addresses. In particular, it writes the shell-code address into the GOT entry for `free`, so that a subsequent call to `free` (which occurs on the next line in the program, immediately after the exploited `free`(p_2) call) will transfer control to the shell-code. Since **traceroute** is run with *setuid* as root, the attacker gains root access to the system.

This vulnerability is detected by our tool because the pointers p_1 and p_2 are *unsafe* (as determined by our static analysis), so each write via p_1 or p_2 is instrumented to check that it refers to *appropriate* memory. Since B is *tracked*, the mirror of B is marked *appropriate* when it is allocated, and *inappropriate* when it is freed (via the call free(p_1)); after B is freed, the program attempts to write via p_2 (while processing Argument 2), which is detected as an error by our tool.

	lines of	exec time (sec.)		slow-	% locs	% unsafe derefs		compile time (sec.)			executable size (KB)		
	code	orig.	inst.	down	tracked	static	dynamic	orig	inst	slowd	orig	inst	bloat
Program	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)
Cyclone													
aes	1822	2.48	3.86	1.53	3.51	97.1	99.6	5.8	18.5	3.20	26.2	91.1	3.48
cacm	340	3.19	4.02	1.18	3.51	92.3	100.0	1.0	6.2	6.51	16.7	55.1	3.30
cfrac	4218	5.26	11.04	2.08	5.28	98.2	99.7	11.7	79.9	6.84	53.8	151.1	2.81
grobner	4737	2.06	5.88	2.83	9.84	68.1	98.1	13.6	69.6	5.12	51.0	152.7	3.00
matxmult	1377	2.08	3.94	1.84	0.95	100.0	100.0	0.6	3.2	5.26	14.5	59.6	4.12
ppm	1421	1.57	2.49	1.58	4.71	100.0	100.0	4.8	11.6	2.42	31.3	77.3	2.47
tile	4880	1.10	4.87	4.22	1.52	100.0	100.0	5.1	22.2	4.34	41.0	111.6	2.73
Olden													
bh	3200	8.97	20.58	2.25	3.07	74.2	99.9	3.1	17.0	5.56	21.4	80.1	3.74
bisort	690	3.74	6.29	1.68	1.86	76.5	94.9	2.1	10.6	5.08	18.2	70.2	3.85
em3d	538	6.54	9.11	1.39	1.93	100.0	100.0	1.9	12.1	6.38	17.3	65.7	3.79
health	706	4.95	8.17	1.63	0.33	74.1	97.9	2.4	15.3	6.41	17.7	65.1	3.68
mst	610	5.45	6.05	1.11	0.61	88.2	100.0	2.0	12.1	6.20	16.9	71.3	4.22
perimeter	472	0.97	0.98	1.00	0.00	0.0	0.0	2.0	8.6	4.35	17.5	55.1	3.14
power	867	7.52	8.80	1.17	1.25	47.5	100.0	2.8	15.1	5.36	20.2	68.5	3.40
treeadd	375	1.55	1.61	1.03	0.00	0.0	0.0	1.1	7.5	6.80	15.2	52.7	3.47
tsp	684	5.22	5.73	1.10	0.32	79.6	60.0	2.4	12.6	5.23	19.2	67.9	3.54
Spec 95													
compress	3900	21.94	31.55	1.42	3.91	100.0	100.0	2.4	13.7	5.72	86.0	136.6	1.59
gcc	205106	4.69	11.13	2.18	22.24	84.2	96.3	419.0	2151.9	5.14	1347.2	2328.1	1.73
go	29629	9.34	21.60	2.29	6.92	96.9	98.2	72.2	276.0	3.82	318.2	641.2	2.02
ijpeg	31215	2.20	3.32	1.39	5.05	92.7	100.0	66.8	493.2	7.38	177.2	549.9	3.10
li	7630	2.77	4.90	1.76	17.51	98.7	100.0	25.4	181.7	7.16	86.0	248.0	2.88
m88ksim	19227	1.31	2.05	1.47	17.77	90.8	98.2	60.1	559.8	9.32	178.2	384.9	2.16
perl	26872	9.51	76.74	8.02	36.84	97.6	98.7	100.7	297.6	2.95	288.0	571.3	1.98
vortex	67219	8.53	29.37	3.13	19.11	85.7	83.0	171.6	1093.8	6.38	671.3	1471.0	2.19
Spec2000													
ammp	13483	46.94	112.45	2.38	2.93	77.8	99.6	46.5	216.5	4.66	132.0	326.8	2.48
art	1270	62.30	65.15	1.04	2.86	81.1	100.0	5.1	11.9	2.34	30.1	96.2	3.20
bzip2	4650	2.98	8.35	2.77	3.91	92.9	99.2	12.8	32.9	2.56	62.7	143.4	2.29
crafty	20545	2.28	3.63	1.54	7.14	92.5	98.3	77.6	308.3	3.97	277.9	519.3	1.87
equake	1513	5.94	9.75	1.63	8.26	92.1	95.2	5.4	20.3	3.73	32.8	111.3	3.39
gzip	8605	4.16	5.73	1.36	5.85	94.7	100.0	15.1	64.0	4.22	67.5	158.9	2.35
mcf	2412	1.07	1.70	1.57	1.72	78.6	99.9	8.0	51.7	6.47	25.9	103.5	4.00
parser	11391	7.46	14.00	1.90	29.80	98.4	100.0	46.0	161.8	3.52	155.3	337.3	2.17
twolf	20461	1.10	2.41	2.02	3.25	98.8	100.0	92.4	489.3	5.29	216.3	521.6	2.41
vpr	17730	2.99	4.44	1.48	6.66	95.8	99.4	46.9	173.7	3.70	162.8	311.1	1.91

Table 1: Experimental Results

4. PERFORMANCE OF THE TOOL

To evaluate the performance of our tool, we instrumented programs in the SPEC 95, SPEC 2000, and Olden benchmark suites, as well as some of the programs used to evaluate Cyclone [17]. The programs were compiled with `gcc`'s `-O3` optimization, and executed on a 333MHz Pentium II with 128MB RAM, running Linux. The running times of the instrumented programs were compared with those of the original uninstrumented program, and data about the amount of instrumentation (both static and dynamic), compilation time, and code bloat were gathered. Results are presented in Table 1.

Column (a) gives the sizes of the benchmarks in lines of code. Note that we are able to handle large programs; in particular, `gcc` (the largest program we tried) has over two hundred thousand lines of code. Columns (b) and (c) give the running times (elapsed time, in seconds) of the original program and the instrumented program, each averaged over five runs. Column (d) gives the slowdown factor of the instrumented program compared to the original program. These slowdown factors are graphed in Figure 2, with the benchmarks sorted in order of increasing size (lines of code). On average, the instrumented programs ran about 1.97 times slower than the original program. Note that the slowdown factor is independent of program size and execution time.

Figure 3 gives a comparison of the execution time slowdown of our tool compared to Cyclone (version 0.5) and CCured (version 1.1.2), for the Cyclone benchmarks. All were compiled with `-O3` optimization, and executed on the same machine on the same inputs. For Cyclone, we used the version of the source code that had been translated (manually) to Cyclone, so the better performance (average 1.4× slowdown) may be due in part to the fact that a human was involved in identifying pointers that should be made

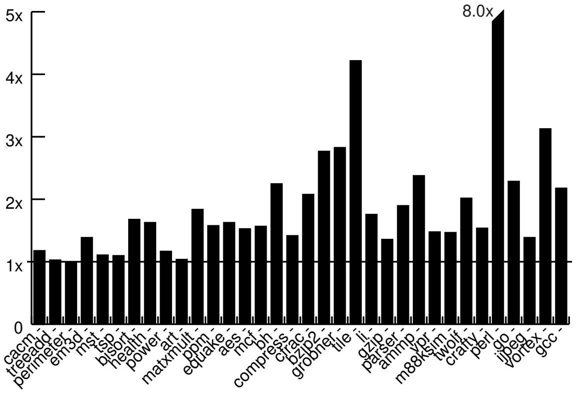


Figure 2: Runtime Overhead

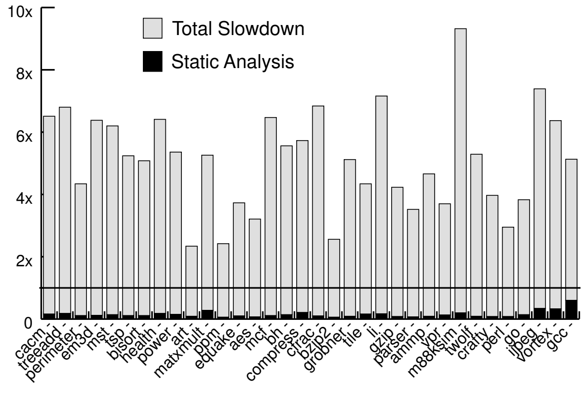


Figure 4: Compilation-Time Slowdown

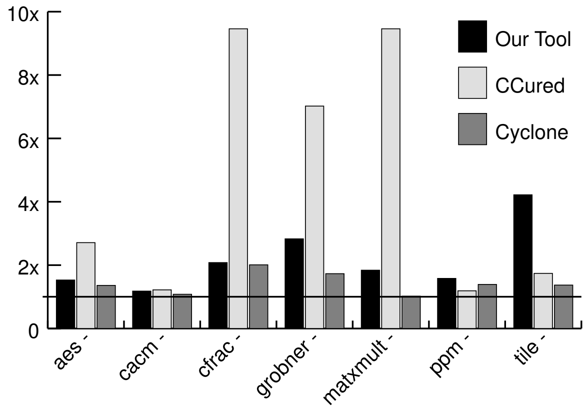


Figure 3: Runtime Overhead Comparison

fat pointers. For CCured, we changed the source code only minimally to get the benchmarks to compile and run without reporting errors, i.e., while a few changes were needed to eliminate compile-time errors and false positives at runtime, we ignored compile-time warnings, effectively relying on CCured's type-inference scheme to assign (possibly conservative) pointer annotations. The average slowdown is 4.7×, which incidentally is much higher than the slowdowns reported in their papers [23, 9].

Column (e) gives the percentage of tracked locations in the program. Each statically-declared object and dynamic-allocation callsite (i.e., each call to `malloc` or one of its relatives) is counted as one location; on average, about 7% of all locations are tracked, which means the remaining 93% of the locations are always marked *inappropriate*, so that malicious accesses that write to those locations will be detected by programs instrumented with our tool.

Columns (f) and (g) give the percentage of writes via dereferences that were instrumented because the dereferenced pointer was identified as *unsafe*: column (f) reports static instances, while column (g) reports dynamic occurrences. Except for two small programs (`perimeter` and `treeadd`, in which our static analysis was able to determine that all pointers were safe, so that no instrumentation was added), a high percentage of writes via dereferences were instrumented: on average, about 84% of the static writes and about 91% of the dynamic writes were instrumented.

Columns (h) to (j) give an indication of the compile-time overhead incurred by using our tool: on average, it took five times as long to analyze, instrument, and compile a program with our tool as it does to compile the program uninstrumented. These slowdown factors are shown in Figure 4, again sorted by lines of code, with the black areas indicating the proportion of the compilation time spent in performing static analysis (including points-to analysis). Note that both the static analysis and the overall compilation time scale well to large programs.

Finally, columns (k) to (m) provide information about code bloat: On average, our instrumented executable is 2.9 times[3] the size of the uninstrumented executable (compared to about 8× for both Cyclone and CCured).

5. IMPROVEMENTS AND FUTURE WORK

It is not clear whether the slowdown introduced by our tool is acceptable for deployed code. Therefore, one important direction of future work is to investigate ways to improve performance. The main direction of possible improvement is in the static analysis that identifies unsafe and tracked locations; improvements in this regard would be reflected both in reduced overhead and increased security.

One possibility is to use a more precise points-to analysis, including flow-sensitive and context-sensitive analyses (e.g., [1, 19, 34, 35]). Greater precision will result in fewer unsafe pointers and tracked locations, but scalability issues might then prevent the tool from being applicable to large programs.

Another possibility is to use flow-sensitive analysis to identify *redundant* checks that can be safely eliminated: if a pointer is dereferenced for writing several times with no intervening change to its value, instrumentation is needed only for the first dereference. We have implemented an intra-procedural dataflow analysis to detect such redundant checks, where a check of the dereference expression $*e$ is redundant at control-flow graph node n if every path from the

[3]Note that expressing the code bloat in this way is somewhat misleading, as the instrumented program size includes a constant component of about 54KB for our tool's runtime library, including wrappers for library functions. The bloat factor is thus smaller for larger programs.

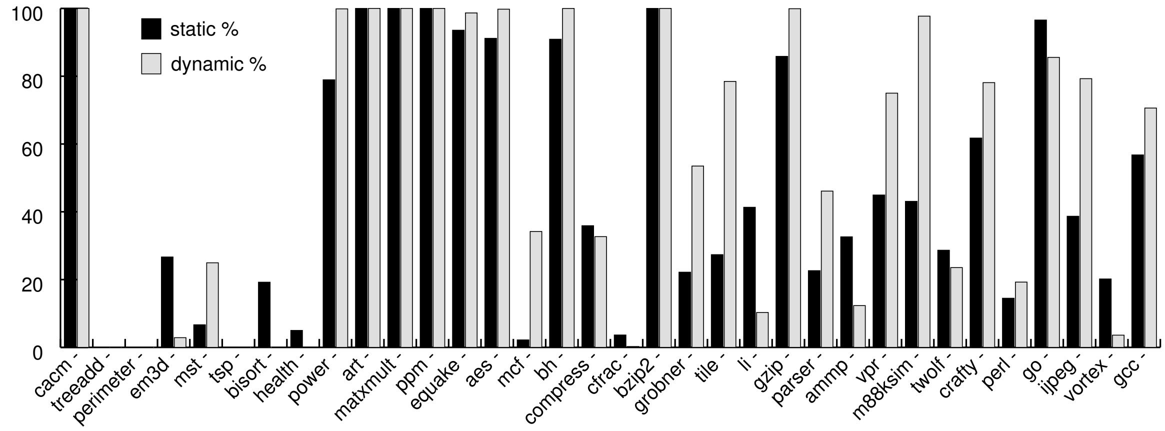


Figure 5: Percentage Array Dereferences

control-flow graph's *enter* node to n includes a node m such that:

- $*e$ is checked at m, and
- no path from m to n changes the l-value of $*e$ or deallocates the location pointed to by e.

However, the results of this optimization were disappointing: marginal performance improvements (less than 4%) were achieved in only a few of the benchmarks.

An analysis with more potential for improvements is array-bounds checking elimination. Recall that an array indexing expression is treated as an unsafe pointer dereference by our tool, so all writes via array index expressions are instrumented, and all arrays that are written to are tracked. If we can statically determine that an array access is guaranteed never to be out of bounds, then the check associated with that access can be eliminated, and if all accesses to a given array are safe, then the array need not be tracked.

The detection and elimination of unnecessary array-bounds checks in Java and other safe languages has been studied extensively (e.g., [32, 22, 15, 18, 5, 21]), yielding good results: for example, Gupta's dataflow analysis [15] eliminated 40%-100% of array-bounds checks on some small benchmarks; Kolte and Wolfe [18] eliminated 60%-90% of dynamic checks in Fortran benchmarks; and the algorithm of Bodik *et al.* [5] eliminated about 31% of static upper-bound checks in Java benchmarks. Figure 5 gives the percentage of unsafe pointer dereferences (both static and dynamic instances) in our instrumented programs that were array-indexing expressions. As both percentages are quite high (over 90% for many of the benchmarks), it suggests that array-bounds analysis can greatly reduce the number of unsafe pointer dereferences that are instrumented.

5.1 Instrumenting Reads

Another direction of improvement is to increase the scope of detected errors. Note that our tool only checks *writes* via dereferences, and not *reads*. We chose to do this partly for performance: as programs typically perform more reads than writes, the runtime overhead of instrumenting reads is significantly higher. Further, most important classes of vulnerabilities can only be exploited via malicious writes, so a security tool that monitors only writes is sufficient to detect most attacks.

However, checking *reads* through unsafe pointer dereferences may be desirable if the user is concerned about an attacker gaining read access to confidential data. Such a tool may also be useful for debugging purposes, as an invalid read indicates a programming error that should be corrected. The modification to our tool needed to do this is minimal. While the runtime overhead will increase (since many more pointer dereferences will be instrumented) the redundant-check analysis that we have implemented might be more effective in this context (since multiple reads from the same memory location are more common than multiple writes).

5.2 Function Pointers

One class of likely targets of attacks are function pointers. In the benchmarks we tested, there were 101 function pointers (mostly in the SPEC95 benchmarks `gcc`, `ijpeg`, and `perl`). Our current implementation identified only 2 of these function pointers as *tracked* (both in `vortex`). This means that an attack that tries to overwrite the value of any of the 99 untracked function pointers would be detected by our tool. This indicates that our tool is effective at protecting against attacks via function pointers.

Nonetheless, the two tracked function pointers remain vulnerable, and there may be other programs for which our static analysis will not identify as many untracked function pointers. Since function pointers are considered dangerous targets of potential attacks, it may be worthwhile to handle them specially: the set of appropriate targets of function pointers could be tracked separately from the appropriate targets of unsafe pointer dereferences. This tracked set could be identified as the set of all defined functions, or more precisely, the set of functions whose addresses are taken. At runtime, each function call via a function pointer would be checked against this set of appropriate functions. A call to a function not in the appropriate function set would indicate a security violation, and the program would be halted. For space efficiency, because the set of appropriate functions is relatively small and sparsely distributed, this set could be stored in a hash table rather than a mirror of memory.

Note that, like the current approach, this approach can handle arbitrary manipulation and passing of function pointer values, including via callbacks and indirection tables; runtime overhead would be incurred only when a function is called via a function pointer.

6. RELATED WORK

Dynamic techniques can be used in deployed software to prevent security attacks, and can also be used during software development to find bad pointer dereferences. In both cases, the program is instrumented to permit runtime monitoring; however, the difference in *when* the analyses are used leads to importance differences in how they handle errors, and also in their design goals:

- In the context of debugging, erroneous behaviors are reported, but execution continues (so that as many errors as possible can be found during one run of the program). In the context of security checking, erroneous behavior that indicates a potential security violation causes program execution to be terminated.

- For the purposes of debugging, a fair amount of overhead can be tolerated. When an analysis is intended for use in deployed software, the more execution is slowed down, the less likely it is that the analysis will actually be used. Therefore, efficiency is one of the most important goals.

Examples of dynamic tools intended for deployed code are StackGuard [10, 30] and StackShield [31], described in the Introduction. They achieve low overhead at the expense of generality (StackGuard only detects attacks on the return address, while StackShield only detects attacks on the return address and function pointers). In contrast, the security tool discussed in this paper is designed to detect a wide range of security attacks, including simple denial-of-service attacks that only corrupt data, without wresting control of the program.

Another facility for preventing attacks in deployed code is to make the stack space non-executable [24]. This eliminates the ability for attackers to introduce their own code for execution, but also limits the flexibility of certain programming techniques (e.g., "trampolines"), and does not prevent attacks via system-call arguments or other data-corruption attacks.

Cyclone [17] and CCured [23, 9] are the two dynamic tools that are most closely related to our tool. They both use static analysis to identify unsafe pointers, and use fat pointers to detect out-of-bounds pointer accesses. As they are both new languages, they suffer from porting issues (while they include facilities for importing existing C programs, these facilities disallow certain programs), while our tool works for all ANSI C programs, and can be applied directly to legacy code.

Purify [16] is a dynamic tool that is effective at detecting buffer overruns, memory leaks, and other errors. It instruments object code, thus it does not require source code, but is platform-specific and not portable. Since it is designed as a debugging tool, it suffers from a high overhead (about 15× slowdown) which is impractical for use in deployed code. Valgrind [28] checks for similar violations by interpreting the executable binary on a "synthetic CPU"; thus, it has an even higher runtime overhead (about 40× slowdown), but does not require any compile-time actions. The `memcheck` component of Valgrind associates each byte of memory with a *valid-address* bit, similar in meaning to our *appropriate* tag (additionally, each byte is associated with eight *valid-value* bits to detect uses of uninitialized memory). Insure++ [26] likewise is designed as a relatively heavyweight debugging tool rather than for use in deployed code; it detects a number of common sources of program errors, including out-of-bounds array accesses and null-pointer dereferences.

Safe-C [3, 27] uses fat pointers, and includes a multiprocessor optimization, in which instrumentation operations are performed in a "shadow" process on a separate processor. It does not distinguish between "safe" and "unsafe" pointer dereferences, but does use dataflow analysis to eliminate redundant checks. The Runtime Type-checker [20] is a debugging tool that detects a broader class of errors, by tracking the runtime types of values in the program, and reporting an error when a value of one type is used in the context of an incompatible type. The approach described in this paper is derived from the Runtime Type-checker, and has a much lower runtime overhead, while detecting a smaller class of errors.

Static error-detection techniques [2, 4, 6, 12, 13, 14, 33] analyze a program without executing it to find errors or security holes. The main benefit of static techniques is that they can detect errors in portions of the program that are infrequently executed. Unfortunately, precise static techniques are expensive, and thus do not scale to large programs. To improve performance, either the user must supply annotations (as in LCLint [14] and ESC [12]), or a less precise analysis must be performed (which may lead to missing some potential problems, or to reporting false positives), or the scope of the analysis must be limited, either by checking only certain paths (like PREfix [6]), or to work intraprocedurally, leaving more "global" problems undetected.

7. CONCLUSION

We have presented a security tool for C that instruments programs to check at runtime for invalid pointer dereferences that may be vulnerable to malicious attack. It uses static analysis to identify unsafe pointers and their legitimate targets, and at runtime maintains a tag for each memory location to indicate whether it is an appropriate target of an unsafe pointer. The tool is fully automatic (it does not require programmer annotations), and is compatible with ANSI C (it is portable, and does not limit the flexibility provided by the C language). While the instrumented program is not guaranteed to prevent all attacks, it will detect attempts to exploit a large class of vulnerabilities, from the popular stack smashing to more subtle attacks, and it does not report false positives. The execution-time overhead is relatively low, and we believe it can be improved with more aggressive static analysis.

Acknowledgements

This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435. We thank George Necula for help with CCured, and the anonymous reviewers for their useful suggestions.

8. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.D. thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, May 2002.

[3] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 290–201, Orlando, FL, June 1994.

[4] T. Ball and S. Rajamani. The SLAM toolkit. In *13th Conf. on Computer Aided Verification*, pp. 260–264, July 2001.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 321–333, Vancouver, BC, June 2000.

[6] W. Bush, J. Pincus, and D. Sielaff. A static analyazer for finding dynamic programming errors. *Software–Practice and Experience*, 30(7):775–802, June 2000.

[7] cfingerd: Configurable finger daemon. http://www.infodrom.org/projects/cfingerd/

[8] Ckit. http://www.smlnj.org/doc/ckit/

[9] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 232–244, San Diego, CA, June 2003.

[10] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, San Antonio, TX, Jan. 1998.

[11] M. Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 35–46, Vancouver, BC, June 2000.

[12] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Tech. Report SRC-159, Compaq SRC, 1998.

[13] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *The 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, page 194. Springer, July 2001.

[14] D. Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 44–53, Philadelphia, PA, May 1996.

[15] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, Mar.–Dec. 1993.

[16] R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.

[17] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[18] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 270–278, La Jolla, CA, June 1995.

[19] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 235–248, San Francisco, CA, June 1992.

[20] A. Loginov, S. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pp. 217–232. Springer, Apr. 2001.

[21] M. Lujan, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of java array bounds checks in the presence of indirection. Tech. Report CSPP-13, Department of Computer Science, University of Manchester, Feb. 2002.

[22] V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 17(6)*, pp. 114–119, Boston, MA, June 1982.

[23] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symp. on Principles of Programming Languages*, Portland, OR, Jan. 2002.

[24] Openwall project linux kernel patches. http://www.openwall.com/

[25] Packet storm. http://packetstormsecurity.org/

[26] Parasoft. Insure++: An automatic runtime error detection tool. http://www.parasoft.com/insure/

[27] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software–Practice and Experience*, 27(1):87–110, Jan. 1997.

[28] J. Seward. The design and implementation of Valgrind. http://developer.kde.org/~sewardj/

[29] N. P. Smith. Stack smashing vulnerabilities in the UNIX operating system. Technical report, Computer Science Department, Southern Connecticut State University, 1997.

[30] Immunix Stack Guard. http://immunix.org/stackguard.html

[31] Stack Shield. http://www.angelfire.com/sk/stackshield/

[32] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *ACM Symp. on Principles of Programming Languages*, pp. 132–143, Los Angeles, CA, Jan. 1977.

[33] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symp. on Network and Distributed Systems Security*, pp. 3–17, San Diego, CA, Feb. 2000.

[34] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for c programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 1–12, La Jolla, CA, June 1995.

[35] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 91–103, Atlanta, GA, May 1999.