PointGuard™: Protecting Pointers From Buffer Overflow Vulnerabilities

Crispin Cowan, Steve Beattie, John Johansen and Perry Wagle

Immunix, Inc. http://wirex.com/
crispin@immunix.com

Abstract

Despite numerous security technologies crafted to resist buffer overflow vulnerabilities, buffer overflows continue to be the dominant form of software security vulnerability. This is because most buffer overflow defenses provide only *partial* coverage, and the attacks have adapted to exploit problems that are not well-defended, such as heap overflows. This paper presents PointGuard, a compiler technique to defend against most kinds of buffer overflows by encrypting pointers when stored in memory, and decrypting them only when loaded into CPU registers. We describe the PointGuard implementation, show that PointGuard's overhead is low when protecting real security-sensitive applications such as OpenSSL, and show that PointGuard is effective in defending against buffer overflow vulnerabilities that are not blocked by previous defenses.

1 Introduction

Despite numerous technologies designed to prevent buffer overflow vulnerabilities, the problem persists, and buffer overflows remain the dominant form of software security vulnerability. Attackers have moved from stack smashes [25] to heap overflows [5], `printf` format vulnerabilities [6], multiple free errors [1, 13] etc. which bypass existing buffer overflow defenses such as non-executable memory segments [14, 12], StackGuard [9] and libsafe [2].

All of these classes of attack work to corrupt *pointers*: Sometimes code pointers (function pointers and longjmp buffers) and sometimes data pointers [4]. In principle, an attacker can use overflows to corrupt arbitrary program objects, but in practice corrupting pointers is by far the most desirable attacker target. The reason is that the attacker is seeking *total* control of the victim process, i.e. they want the process to execute *payload* code [25] so they can get to a root privileged shell.

Thus we sought an effective defense for pointers. *PointGuard* defends against pointer corruption by encrypting pointer values while they are in memory, and decrypt them only immediately before dereferencing, i.e. just as they are loaded into registers. Attackers attempting to corrupt pointers in memory in any way can *destroy* a pointer value, but cannot produce a *predictable* pointer value in memory because they do not have the decryption key. We modify a C compiler (GCC) to emit code that encrypts pointers for storage in memory and decrypts them for dereferencing.

The rest of this paper is organized as follows.Section 2 elaborates on pointer corruption vulnerabilities. Section 3 presents the PointGuard design and implementation. Section 4 presents our compatibility testing, showing that PointGuard imposes minimal compatibility issues on commonly used software. Section 5 presents our security testing against known pointer corruption vulnerabilities in actively used software. Section 6 presents our performance testing, showing that the performance costs of PointGuard protection are minimal. Section 7 describes related work in defending against pointer corruption. Section 8 presents our conclusions.

2 Pointer Corruption Vulnerabilities

An attacker's goal in attacking a vulnerable program is to obtain that program's privileges. While the attacker could manipulate the program into directly performing

This work supported in part by DARPA contract N66001-00-C-8032.

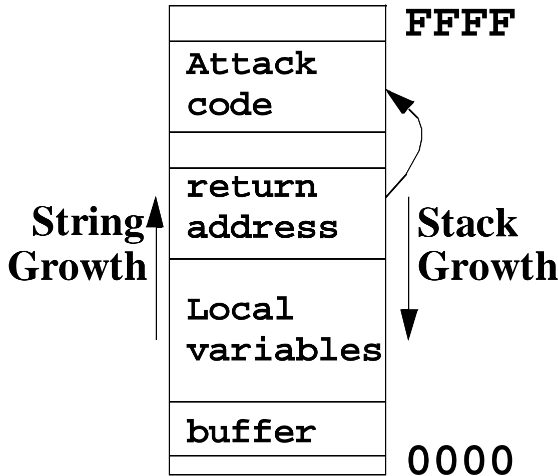

Figure 1: "Stack Smash" Attack Against Activation Record

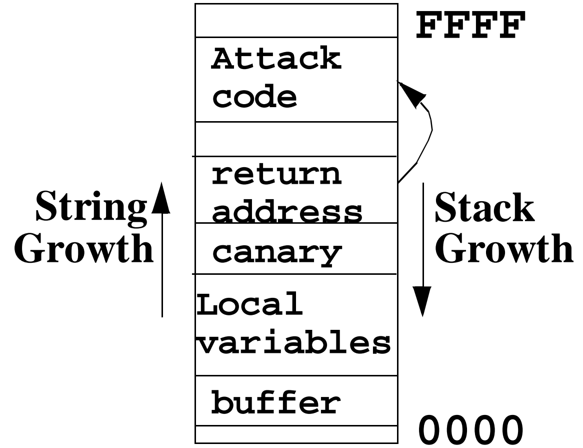

Figure 2: StackGuard Defense Against Stack Smashing Attack

some action, it is much more convenient for the attacker to get a shell prompt with the program's privileges. For these reasons, the attacker most desires to get the victim program to *execute arbitrary code*, colloquially referred to as "shell code" because the common code performs the semantic equivalent of `exec(/bin/sh)`.

To get the program to execute shell code, the attacker wants to modify a *code pointer* (a pointer that the program expects to point to legitimate code) such that the code pointer points to code the attacker would like executed. Once the code pointer has been thus modified, at some later point in execution the program dereferences the corrupted code pointer, and instead of jumping to the intended code, the program jumps to the attacker's shell code.

The classic form of buffer overflow attack is the "stack smash" described by Aleph One [25]. In this attack, buffers (character arrays) that are allocated on the stack (*automatic* variables [21] declared within the body of a C function) are overflowed with the goal of corrupting the function's return address within the activation record, as shown in Figure 1. In previous work, we provided the StackGuard defense [9] against stack smashing attacks, which detect corrupted return address values in activation records by ornamenting the activation record with a *canary* word, as shown in Figure 2. When a stack smash occurs, the overflow necessarily corrupts the canary word.

The general case of buffer overflows is to overflow buffers allocated *anywhere* (stack, heap, or static data area) corrupting whatever important state is adjacent to the overflowable buffer. Figure 3 shows a simple program subject to a static buffer overflow, where excess input intended for `statbuf` corrupts the `statptr` pointer. Figure 4 shows a similar program subject to heap overflows. In addition to threatening the adjacent buffer, this overflow also has the potential to corrupt the malloc data structures.

In principle, this would allow the attacker to induce arbitrary behavior in the victim program. In practice, the state adjacent to overflowable buffers are determined by the vagaries of data layout within the program, and depending on circumstances, may have limited semantics of use to the attacker. Thus attackers are most inter-

```
char statbuf[100];
char * statptr; // vulnerable to statbuf

myfunc() {
      statptr = malloc(1000);
      gets(statbuf);
      gets(statptr);
}
```

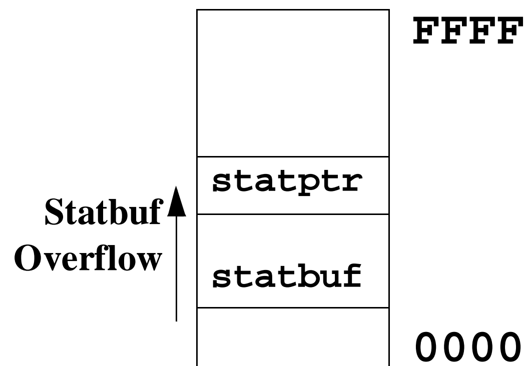

Figure 3 Vulnerable to Static Overflow

ested in corrupting *pointers*, because they give the most leverage.

Conover et al [5] describe a variety of methods to use buffer overflows against buffers located in heap and static data areas to corrupt adjacent pointers. In some cases, they directly corrupt *code pointers* (function pointers and `lonjmp` buffers) to directly seize control of the victim program. In other cases, they *indirectly* use overflows to corrupt *data pointers* to point to unintended locations, and from there use those data pointers to corrupt code pointers. Thus the PointGuard defense is designed to protect all pointers from corruption.

3 PointGuard Defense Against Pointer Corruption

The PointGuard defense against pointer corruption consists of encrypting pointer values in memory and only decrypting the pointers when they are loaded into CPU registers. Section 3.1 describes basic PointGuard operations. Section 3.2 describes PointGuard encryption. Section 3.3describes our implementation of the Pointguard defense. Section 3.4 elaborates on special implementation issues that Pointguard imposes on the compiler and on developers.

3.1 PointGuard Operation

Pointers are vulnerable in memory, where they can be corrupted using various buffer overflow and printf format string attacks. Encryption protects pointers, because the attacker cannot corrupt pointers such that they will *decrypt* to a predictable value. Conversely, pointers are safe in registers, because registers are not addressable via computed addresses, and thus not subject to overflow attacks.

This scheme critically depends on pointers always being loaded into registers prior to being dereferenced. Older CPU instruction sets support various forms of *memory indirection* [23] where a pointer could be dereferenced without using a register. More recent RISC instruction set architectures dispensed with memory indirection as being too slow, adopting instead a *load/store architecture* [28] in which *all* values are loaded into registers before operating on them. Even legacy CISC instruction set architectures [17] found load/store instruction sequences to be most efficient, and compilers for these CPUs were subsequently changed to prefer load/store instruction sequences. PointGuard critically depends on a load/store instruction discipline.

Figure5 through Figure8 illustrate how PointGuard works to defend pointers. Figure 5 shows a normal pointer dereference. Figure 6 shows a normal pointer dereference under attack, where the attacker used a buffer overflow or related means to corrupt a pointer to point to a different location. Figure 7 shows a PointGuard dereference, decrypting the pointer value as it is loaded into the CPU register. Figure 8 shows a PointGuard dereference under attack. The attack fails because the attacker's corrupted value is passed through the PointGuard decryption process, producing a *random* address reference, and for reasonably sparse address spaces, probably causing the program to crash Crashing is the objective: to cause the victim program to *fail-stop*, rather than hand control over to the attacker.

Notably critical to this scheme is that the attacker cannot know or predict the encryption key. This key is a relatively easy secret to keep, because it is never shared. The key is generated at the time the process starts, using some suitable source of entropy such as reading a value from `/dev/random`. This key is then never shared with any entity outside the process's address space. To obtain the key, the attacker would either have to already have permission to manipulate the process with debug-

```
myfunc() {
       char *buf1 = (char *)malloc(BUFSIZE);
       char *buf2 = (char *)malloc(BUFSIZE);

       gets(buf1);
       gets(buf2);
}
```

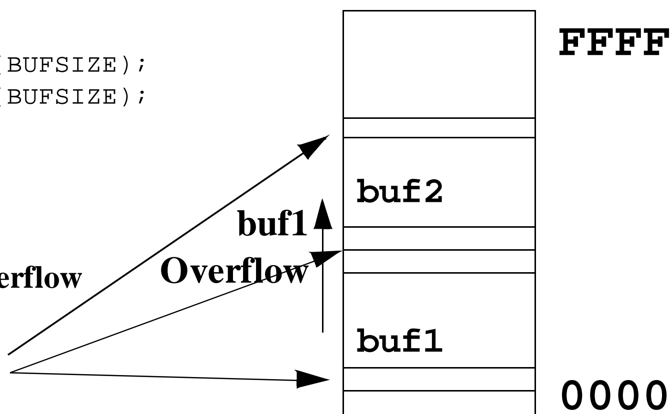

Figure 4 Vulnerable to Heap Overflow

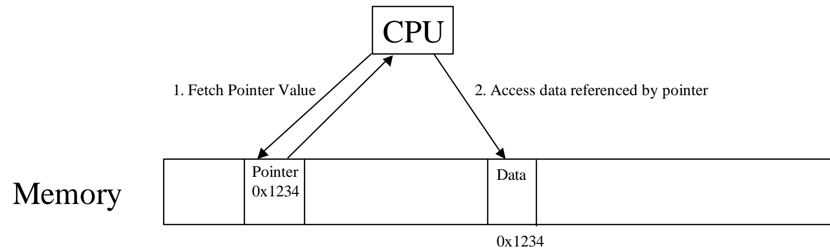

Figure 5 Normal Pointer Dereference

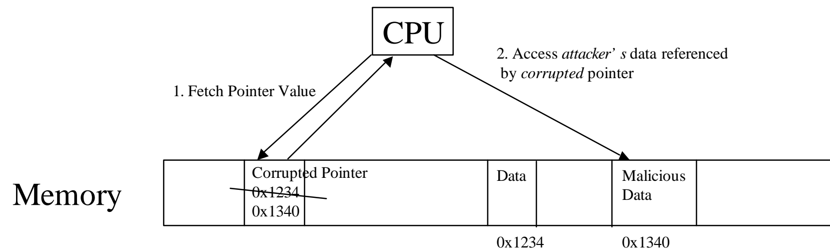

Figure 6 Normal Pointer Dereference Under Attack

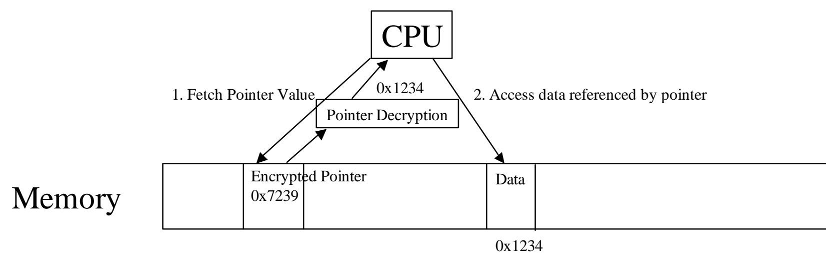

Figure 7 PointGuard Pointer Dereference

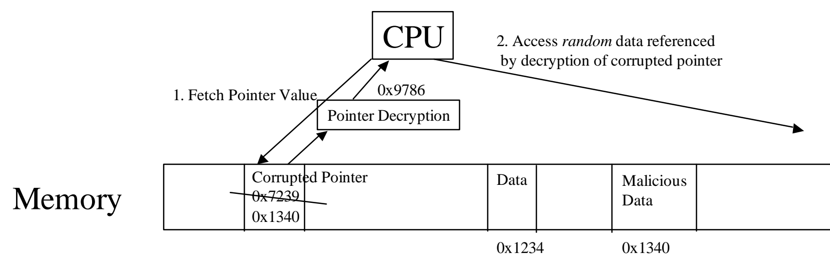

Figure 8 PointGuard Pointer Dereference Under Attack

ging tools (e.g. `ptrace`) or would have to have *already* successfully perpetrated a buffer overflow attack against the process.

The key is available to processes that share memory, but such processes are already effectively sharing enough to be fully vulnerable to each other. In fact, the PointGuard key is significantly *less* sensitive than some other, more durable objects that processes may share across a shared-memory boundary, such as file descriptors.

3.2 PointGuard Encryption

PointGuard takes the odd position of using encryption to provide integrity. PointGuard seeks to provide integrity for pointers, so that pointers cannot be modified in ways the programmer did not intend. *Encryption* provides for

confidentiality, but is cryptographically weak providing integrity, and so PointGuard would seem to be cryptographically weak.

However, PointGuard never gives the attacker a look at the ciphertext. The key is chosen anew at process exec() time, and in the absence of *already* having some way to bypass PointGuard, attackers have no way of determining what that key is, or of reading any sample encrypted pointer values. Thus the cryptographic weakness of using encryption for integrity is irrelevant, and it suffices to make it merely improbable to be able to spoof integrity in pointer values *on the first try*. PointGuard makes pointers *brittle* with respect to corruption: attackers cannot use pointer corruption to read or write any *particular* data structure.

Conversely, because PointGuard sits between the CPU level 1 cache and registers, it is very important that PointGuard be fast. There is often a few cycles of load delay slots between loading a pointer into a register value and dereferencing the pointer value, so if decryption can be done in a few cycles, it can be nearly free. Therefore, it is appropriate to use very fast and simple "encryption" techniques, which we elaborate on in Section 3.3.2.

3.3 PointGuard Implementation

PointGuard is implemented as a C compiler enhancement. PointGuard defense mechanisms are integrated into the binary programs that compiler emits. The compiler mist *consistently* perform encryption and decryption of pointers be *consistent*, i.e. that pointers are consistently encrypted at such places that they are always decrypted before use, and that only encrypted pointers are decrypted.

It is important to the security value of PointGuard that pointers are encrypted when stored in memory, i.e. storage that is addressable, and thus vulnerable to attack due to un-typed memory access. CPU registers are notably *not* addressable, and thus the ideal method is to store pointers in the clear in registers, and encrypted in memory. There are several *potential* PointGuard implementation strategies, described in Section 3.3.1. Section 3.3.2 describes our actual PointGuard implementation.

3.3.1 Potential Implementation Strategies

There are many possible places in the compiler to put the encryption/decryption of pointers. These options must all satisfy the "consistency" requirement, and trade off the security value against ease of implementation.

In the Preprocessor: It is possible to use a preprocessor (possibly C's conventional CPP preprocessor) to do a source->source translation of programs, so that all pointer expressions are transformed to include the encryption upon setting pointer values, and decryption on reading pointer values. The advantage of this approach is ease of implementation. The disadvantage is that C macro expansion has a propensity to evaluate terms in an expression multiple times, which could cause problems if it results in pointer values being decrypted *twice*. There is also a substantial risk that transient subexpressions performing the PointGuard encryption/decryption might result in temporary values containing clear text pointers being left in memory.

In the Intermediate Representation: Most compilers first transform the source code into an intermediate representation (commonly known as an *abstract syntax tree*) to perform architecture-independent manipulations on the intermediate representation, and then emit architecture-dependent instructions from the modified intermediate representation. One of the manipulations performed on the intermediate representation can be to insert code to encrypt and decrypt pointer values when they are set and read, respectively. The advantage to this method vs. the preprocessor is that it avoids the duplicate expression problem, and can be more efficient. The disadvantage to this method is that it may leave decrypted pointer values in the form of temporary terms in main memory.

In the Architecture-Dependent Representation: As above, maximal security protection is provided if pointer values are only decrypted while in the CPU's registers. However, the CPU's registers are only visible to the compiler in the architecture-dependent representation. Compilers are capable of manipulating architecture-dependent representations (e.g. for "peephole optimization") but at the cost of having to do the compiler work over again for each CPU architecture to be targeted (as in GCC, which seeks to target many CPUs, including the Intel x86, the IBM/Motorola PPC, and the Sun SPARC processors). This particular transformation would transform instructions to load pointer values from memory into registers to add the decryption, and would transform the saving of pointer values from registers to memory to add encryption.

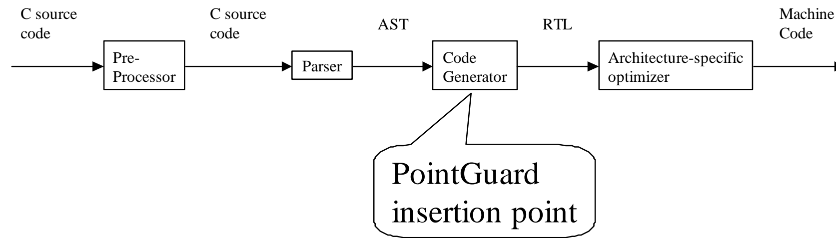

Figure 9 PointGuard in GCC

3.3.2 Actual PointGuard Implementation

Among the three implementation alternatives, we chose to implement PointGuard in the intermediate representation (AST in GCC) as shown in Figure 9. We wanted to implement as *late* as possible in the compiler, to maximize confidence that the PointGuard defenses are not optimized away. AST is the last stage in the GCC compiler where type information is clearly visible, making it possible to distinguish between pointers and other data.

One might be tempted at this point to consider PointGuard-style encryption on *all* loads and stores between memory and CPU registers. However, this defeats the purpose of PointGuard: both legitimate loads and stores and buffer overflows alike would be passed through the PointGuard encryption/decryption, and the buffer overflow attacks would start working again. It is similar to running rot13 encryption *twice*.

The PointGuard encryption method is very basic. The key is a machine-word size (e.g. 32 bits on x86) word, initialized from a suitable source of random keys (in our Linux implementation, `/dev/urandom`) one per process address space. Pointers are encrypted by XOR'ing against the key, and decrypted by XOR'ing again against the same key. This method is simple & efficient; in many cases the single cycle XOR operation will fit neatly into a memory load delay slot (whether implicit or explicit).

This method is cryptographically secure, because cracking it requires guessing the random 32-bit value. Brute force guessing is impractical, because wrong guesses force the victim process to exit, and the new process will have a different key. Massive numbers of processes dying and re-starting is sufficiently "noisy" that conventional intrusion detection methods can detect the attempt at brute force.

Nor can the key be extracted by looking at ciphertext, because the ciphertext is never actually shared with anyone. To obtain a sample of ciphertext, the attacker would have to coerce the victim program into exposing internal pointer values to the attacker. Attackers seeking the privileges of the victim process normally do not have read access to the processes address space. Programs do not normally dump data structures containing pointers outside of their address space, because such pointers lose any meaning outside of the address space.

Thus we cannot identify any feasible means by which the attacker can obtain the PointGuard key.

3.4 PointGuard Special Issues

Apart from the basic problem of building a compiler that correctly emits code implementing the PointGuard defense, Pointguard raises the following special issues. Statically initialized data is described in Section 3.4.1. Protecting the PointGuard key is described in Section 3.4.2. Preventing leaks of clear text pointers is described in Section 3.4.3. Mixed mode code requires programmer intervention, described in Section 3.4.4

3.4.1 Statically Initialized Data

The C language includes support for static initialization of data, including pointers. Normally, these static values are computed at compile time and initialized as the program loads. However, because PointGuard chooses the encryption key at `exec` time, statically initialized pointers cannot be properly initialized until after the program has begun running.

The solution to this is straightforward: we modify the initialization code emitted by the compiler (stuff that runs before `main()`) to re-initialize statically initialized pointers with values encrypted with the current process's key.

3.4.2 Securely Initializing and Protecting the PointGuard Key

Section 3.3.2 described how the attacker cannot feasibly obtain the PointGuard key. But it is also necessary that the attacker not be able to *set* the PointGuard key to a chosen value.

To defend against such an attack, the PointGuard key is stored on its own page when initialized. That page is then made read-only (using mprotect) so that the attacker cannot subsequently use a buffer overflow to change the key value. To make the key page writable again, the attacker would have to execute malicious code, which requires them to bypass PointGuard protection.

3.4.3 Preventing Clear Text Leaks

PointGuard being implemented at the AST level (see Section 3.3.1) the actual CPU registers are not visible. Thus it is possible for the compiler to emit *register spill* instructions that store register contents to the stack in clear text form to make registers available for other purposes. Such register spills are a security threat to PointGuard, because an attacker could potentially use a buffer overflow to corrupt a pointer value stored in clear text form from a register spill, which is subsequently dereferenced by the program when the register values are restored.

To defend against this, a future implementation of PointGuard will *flag* the AST expressions containing PointGuard-relevant expressions such that the flag marks are passed through to the RTL layer. The RTL layer can then notices that PointGuard values are about to be spilled to memory, and insert PointGuard encryption/decryption instructions along with the register spill & restore instructions.

3.4.4 Mixed-mode Code

Because pointers are encrypted, mixed mode code (some PointGuard code, some not) are not compatible unless there is an interface that is aware of the difference, and performs appropriate encryption and decryption. Ideally, this situation should be minimized by compiling all code with PointGuard, as is possible when building an all open source Linux distribution, or in embedded systems.

But in practice, binary-only applications and libraries are sometimes necessary, and so this issue must be handled. There are two cases that need to be handled: PG code calling non-PG code, and non-PG code calling PG code. Both cases are handled with enhancements to function declarations.

```
__std_ptr_mode_on__
#include <stdio.h>
__std_ptr_mode_off__

main() {
      printf("Hello, world\n");
}
```

Figure 10 `hello.c` calling non-PointGuard library functions

To handle PG code calling non-PG code, we specially mark all external function prototype declarations for non-PG code. This is made convenient by the compiler directives `__std_ptr_mode_on__` and `__std_ptr_mode_off__`. Functions declared *between* these directives are marked as needing special marshalling to decrypt arguments before the functions are called. Thus the "hello world" program shown in Figure 10 that is calling out to a non-PG stdio library can make its call to the `printf` library function.

These declarations can be used at finer granularity as storage classes, which is useful for declaring arguments being passed in from non-PG code to PG code. For instance, declaring "`char * __std_ptr_mode_on__ x`" says that x is a *non-encrypted* pointer, and PointGuard will then omit the decryption code when reading this argument. A shorthand notation for this is "`char @x`".

Because the conversion will be handled by the function itself, it does not matter what kind of function prototype the calling code presumed. So for instance, the `printenv` program shown in Figure 11 declares that its second argument is of type "`char @@`" rather than of type "`char **`". The @ in place of * denotes a pointer that is *not* encrypted, and thus needs to be handled specially.

It should be further noted that the @ type declarations need to be carried deeper into the PG code, to the extent that the data structure being passed in has depth. Abstractly, if the non-PG code passes in a pointer to pointer to integer, and the PG code wants to dereference the first pointer and then pass pointer to integer to a second internal PG function, then the internal function needs to also understand that the argument is not encrypted. Concretely, the example shown in Figure 11 handles the `envp` argument, which is of type pointer to pointer to character array, which passes the `envp` to an internal function `print_env()` expecting a similar

```
__std_ptr_mode_on__
#include <stdio.h>
__std_ptr_mode_off__

void
print_env (char @*envp)
{
  int i;

  for (i = 0; envp[i] != NULL; i++)
        printf ("%s\n", envp[i]);
}

int main (int argc, char @@ argv, char @@ env)
{
  print_env (env);
  return 0;
}
```

Figure 11 `printenv.c` being called by non-PointGuard code

pointer to pointer to a character array, which then has to expect an argument of type `char @*`. to accommodate the fact that `main()` has decrypted the outer pointer, but not the inner pointer.

There are also directives `__hashed_ptr_mode_on__` and `__hashed_ptr_mode_off__`. These directives nest within the `__std_ptr_mode_on__` and `__std_ptr_mode_off__` directives (and vice versa) so that, for instance, within a large header file declared standard, selected data structures can be declared to be hashed.

Our current implementation supports this syntax, but specially exempts `varargs` pointer arguments and does not encrypt them. Future implementations will encrypt `varargs` pointer arguments as well, by carrying the pointer's mode (encrypted or not) as part of the function's type signature within the compiler.

Finally, it should be reiterated that the PointGuard specific syntax is necessary *only* to support mixed mode code. Where *all* code is to be compiled with PointGuard, no syntax changes should be required.

4 Compatibility Testing

PointGuard is basically functional, and can compile & run fairly elaborate applications. Unfortunately, for the reasons explained in Section 3.4.4, code needs to be modified to support interoperation with non-PointGuard code. There are two ways to approach this:

Modify the Application: One could modify the application to use `__std_ptr_mode_on__` when ever it makes a call to a library function or kernel system call. This would be a lot of work, and have to be repeated for each application.

Modify the System Libraries: A more cost-effective approach is to modify the system libraries. Libraries are the middleware for applications to interact with the kernel. As such, they are the natural place to insert PointGuard wrappers to encrypt and decrypt pointer data.

Unfortunately, thorough wrapping of system libraries is a lot of work, and we are not yet done. Similar problems with StackGuard compiling system libraries resulted in approximately 6 months delay between a compiler that could compile applications [9] and a compiler that could compile entire systems [7].

5 Security Testing

Our first security test is against the straw man program shown in Figure12. For illustration, this program places a function pointer (`funcptr`) directly adjacent to an overflowable buffer (`buf`) and then accepts user input that can overflow the buffer and corrupt the function pointer. Exploits for this program without PointGuard are trivial to construct [5].

When the same exploits are tested against a version protected with PointGuard, the victim program crashes with a segmentation fault when it tries to call

```
__std_ptr_mode_on__
   #include <stdio.h>
   #include <stdlib.h>
   #include <unistd.h>
   #include <string.h>
__std_ptr_mode_off__

   #define ERROR -1
   #define BUFSIZE 64

   int goodfunc(const char *str); /* funcptr starts out as this */

   int main(int argc, char @@argv)
   {
      static char buf[BUFSIZE];
      static int (*funcptr)(const char *str);

      if (argc <= 2)
      {
         fprintf(stderr, "Usage: %s <buf> <goodfunc arg>\n", argv[0]);
         exit(ERROR);
      }

      setuid(0);
      funcptr = (int (*)(const char *str))goodfunc;
      memset(buf, 0, sizeof(buf));
      strncpy(buf, argv[1], strlen(argv[1]));
      (void)(*funcptr)(argv[2]);
      return 0;
   }
```

Figure 12 Straw man vulnerability overflowing an adjacent function pointer

`funcptr`. To ensure that the crash is not due to simple variation in program layout, we constructed a *special* version of PointGuard that uses a value of 0 for the encryption key (which has no encryption effect under XOR) and created exploits that actually work against the null-key PointGuard program. These same exploits again produce repeated segmentation faults against the victim program protected with PointGuard.

There are ample live examples of heap and static data overflows that PointGuard is intended to block, such as telnetd [26], WU-FTPd [27], CVS [15], and sudo [20]. Unfortunately, these exploits almost all attack `malloc` data structures, which are part of `glibc`. Until we have a PointGuard version of `glibc`, effective security testing of these programs against live exploits is problematic.

That PointGuard might stop real exploits from penetrating real vulnerabilities in real programs would be comforting, but does not assure that PointGuard is non-bypassable. This is important, because simple obscurity tricks are sufficient to cause many exploits to fail, but these exploits would soon succeed if simply re-tuned to bypass the obscurity defenses.

Unfortunately, it is not possible to use testing and experimentation to show non-bypassability: testing can only show bypassability. Non-bypassability must be established by inspection. Our argument is that:

1. Bypassing PointGuard is defined as hijacking a program by corrupting one or more pointers.
2. *Usefully* corrupting a pointer requires pointing it at a *specific* location.
3. Under PointGuard protection, a pointer cannot be corrupted to point to a specific location without knowing the secret key.
4. Learning the secret key requires either obtaining the secret key directly, or cryptanalysis against a sample pointer value.

5. Obtaining the secret key directly would require corrupting a pointer precisely, which begs the question (see Section 3.4.2).
6. Obtaining a sample of ciphertext (an encrypted pointer) would require either corrupting a pointer precisely (which begs the question) or a program that leaks pointer values (which is highly unusual).

Thus it should be difficult to bypass PointGuard and control a program by corrupting a pointer. However, it is possible for attackers to "bypass" PointGuard by attacking non-pointer objects, such as overflowing one character array to change the string value of an adjacent character array. Such an attack could, e.g. cause a program to change which program the victim program is about to `exec()`. This form of attack is out of scope for PointGuard's protection.

6 Performance Testing

We measure the overhead imposed by PointGuard using microbenchmarks described in Section 6.1 and macrobenchmarks described in Section 6.2. All benchmarks were run on a 1.6 GHz P4-M with 512 KB of L2 cache and 512 MB of DRAM, with the compiler set to schedule for i686:

6.1 Microbenchmarks

The microbenchmarks comprised three tests:

Read: exercise reading pointers by following a linked list.

Write: stores values into an array of pointers.

Read/Write: copies values from one array of pointers to another array of pointers.

Tests were also partitioned into *cachable* and *large*. The cachable tests fit in the L2 cache, while the large tests did not. The rationale being that the cachable case presents smaller/fewer load delay slots (the load delay slot being where PointGuard decryption happens) and thus should expose greater PointGuard overhead.

The cachable microbenchmark results are shown in Table 1, and non-cachable in Table 2. Counter-intuitively, PointGuard imposed less overhead on the cachable case than on the non-cachable case, and in fact PointGuard provided substantial performance *improvements* vs. the standard GCC in many cases.

We believe these performance improvements to be the result of PointGuard imposing heavier use of registers for dereferencing pointers. We do not claim that PointGuard itself is actually enhancing performance, but rather that the PointGuard implementation has exposed the possibility that GCC could provide greater performance if it made more aggressive use of registers.

Table 1: Microbenchmark Results for Cachable

	Non-Optimized	-O2	-O3, -fomit-frame-ptrs
Read	-1.47%	0.94%	-0.79%
Write	2.78%	-11.45%	-11%
RW	-1.01%	-9.93%	-9.83%

Table 2: Microbenchmark Results for Non-cachable

	Non-Optimized	-O2	-O3, -fomit-frame-ptrs
Read	-0.34%	-0.14%	-0.08%
Write	4.02%	3.46%	2.44%
RW	-0.85%	-1.45%	-0.75%

Table 3: SciMark Results for Cachable

	Non-Optimized	Optimized
FFT	-2.12%	-6.09%
SOR	0.20%	-0.21%
Monte	0.63%	0.33%
Sparse	0.26%	-0.16%
LU	-2.95%	-0.76%
Composite	-0.95%	-1.13%

6.2 Macrobenchmarks

Our first macrobenchmark is the SciMark benchmark suite [29], a compute-intensive benchmark for scientific computing. Again, this benchmark was partitioned into cachable and non-cachable data sets.

Our cachable SciMark results are shown in Table 3, and non-cachable results in Table 4. Results are similar to the microbenchmarks, but the performance gains if any are smaller, and performance losses are higher.

Table 4: SciMark Results for Non-cachable

	Non-Optimized	Optimized
FFT	5.43%	4.16%
SOR	-0.06%	0.06%
Monte	0.28%	0.06%
Sparse	0.00%	-0.07%
LU	-0.22%	-0.63%
Composite	0.08%	-0.05%

Table 5: OpenSSL Speed Throughput

Cipher	Normal	PG	% Overhead
MD5 8KB	86,794	88,989	-2.5%
SHA1 8KB	56,180	58,119	-3.5%
DES CBC 8KB	10,619	9336	12%
DES EDE3 8KB	3902	3639	6.7%
Blowfish CBC 8KB	49,987	39,316	21%
RSA 1Kbits	1094	892	18%
DSA 2Kbits	23	19	17%

Our second macrobenchmark is the OpenSSL Speed, a part of the OpenSSL package [11]. This is a benchmark included with the SSL package to measure throughput on various ciphers, in terms of how many units of work can be done in a 10 second period.

Our results for OpenSSL Speed are shown in Table 5. As in the microbenchmarks, performance varies from a nominal speedup of 3.5% to a slowdown of 21%. The variation is due to instruction scheduling and register usage: PointGuard increases register pressure, and consumes load delay slots in CPU scheduling. Increasing register pressure improves performance where registers are available, and decreases performance if registers were already fully occupied. Increased use of delay slots is free if there were empty delay slots, and induces overhead if delay slots were already full.

7 Related Work

Previously we surveyed buffer overflow attacks and defenses [10]. Attacks were classified according to how the malicious code was injected, and how the victim program is coerced to jump to the malicious code. Correspondingly, defenses were classified according to how they stopped these effects.

The related work presented here is somewhat updated, including results produced since our previous survey. A summary is shown in Table 6. "Class" indicates which of the families of technologies a defense can be grouped with. "Coverage" indicates the classes of threats that the technology addresses. "Bypassable" indicates whether the attacker can, with some effort, craft an attack that will bypass the defense and exploit a vulnerability anyway. "Cost" indicates the cost in either performance or software developer effort.

Note that many of these technologies have distinct areas of coverage, and can be combined to achieve greater coverage. Section 7.1 describes Bounds Checking. Section 7.2 describes various non-executable buffers. Section 7.3 describes address space randomization. Section 7.4 describes pointer protection.

7.1 Bounds Checking

Bounds checking provides "perfect" protection against buffer overflows, but at a substantial cost in compatibility, performance, or both.

Jones & Kelly [19] and Brugge [31] provide full bounds checking for C code while maintaining `sizeof(void *) == sizeof(int)`, which is important for preserving code compatibility with legacy systems. This compiler does so by using associative lookup on each pointer reference to an array descriptor that stores base and bounds. Performance penalties are high, approximately 10X to 30X slowdown.

The Bounded Pointers project [22] also provides full bounds checking, but changes pointers from a single word into a tuple that incorporates base and bounds. This improves performance by eliminating the associative lookup in Jones&Kelly, but costs compatibility because pointers no longer fit in a single word. Performance penalties are still high at approximately 5X slowdown.

More general than bounds checking, *type safety* subsumes bounds checking and protects against all manner of data type chicanery, not just buffer overflows. Classically, strongly typed languages such as Java and ML provide strong static type safety, built in to the program-

ming language semantics. Because the programming languages were designed for type safety, efficiency is high, and compatibility is not an issue.

More recently, hybrid languages designed to be *safer* versions of the C programming language such as CCured [24] and Cyclone [18] have appeared. These are *dialects* of C, removing some unsafe constructs, and adding others. This provides a safe programming environment with good performance, with stronger security than PointGuard, but also imposes a different order of magnitude on the developer to achieve that safety. A developer can port an application to these safer dialects in a few hours or days, where as PointGuard was designed to allow a developer to compile & protect *millions* of lines of code in a few hours or day.

7.2 Non-Executable Buffers

Making various pieces of memory non-executable restricts where attack payload can be injected. This is good because it is fast, transparent, and works on binary-only applications. The main limitation is that this defense can be bypassed, because suitable attack payload code (effectively "`exec(sh)`")) is almost always resident in victim program address spaces, and so pointer corruption is all that is necessary for the determined attacker to succeed.

Non-executable stack segments [12, 14] was one of the first general purpose defenses against buffer overflows. Zero performance cost and near-zero compatibility cost, but can be bypassed.

The PAX project provides non-executable heap, making it more difficult to bypass, but still can be bypassed. Performance cost is substantial at 10% overall.

7.3 Address Space Randomization

PAX also incorporates ASLR (Address Space Layout Randomization) which can be viewed as the dual of PointGuard: rather than randomizing pointers, ASLR randomizes the location of key memory objects. Benefits are similar to PointGuard, but because objects that are randomly located are coarser, there is residual risk of attackers exploiting adjacency and approximate memory location.

Sekar et al [3] have a new implementation of this concept that randomizes more elements of the address space

Table 6: Buffer Overflow Defenses

Class	Technology	Coverage	Bypassable	Cost
Bounds Checking	Jones&Kelly	complete	no	10X to 30X
	Bounded Pointers	complete	no	3-5X
	Safe Languages	complete	no	complete rewrite
	Safer C Dialects	complete	no	port software
Non-executable Buffers	Non-executable stack	stack buffers	yes	0
	Non-executable heap	heap buffers	yes	10-30%
Address Space Randomization	PAX/ASLR	buffer overflows that don't depend on adjacency	probably	~0
	Sekar et al	buffer overflows that don't depend on adjacency	maybe	0-18%
Pointer Protection	StackGuard	activation records	no	~0
	Libsafe	library string functions attacking activation records	yes	~0
	PointGuard	pointers	maybe	0-20%

layout, which may make it harder to bypass than PAX/ASLR.

7.4 Pointer Protection

Libsafe [2] provides plausibility checks on the arguments to the "big 7" string manipulation functions in the standard C library. Libsafe imposes low overhead, and compatibility is excellent, providing protection to binary-only applications. Protection, however, is limited only to vulnerabilities involving the protected 7 functions.

Snarskii [30] introduced pointer integrity checking with a `libc` library for FreeBSD that checked the integrity of activation records created within the library. StackGuard generalized this technique from a single protected library into a compiler, at first using the Random and Terminator Canaries [8]. In 1999 we introduced the XOR canary to address Emsi's Attack [4]. The XOR canary helped improve the PointGuard design by changing PointGuard from using adjacent canaries to directly encrypting pointers. StackGhost [16] provides a similar degree of protection to StackGuard, using the SPARC CPU's register spill detection hardware.

8 Conclusions

PointGuard provides protection against all vulnerabilities related to pointer corruption, which includes most current and anticipated buffer overflow, as well as related attacks such as printf format bugs and multiple free errors. PointGuard imposes minimal performance overhead, compatibility and performance overhead.

9 Availability

When PointGuard is complete, it will be released under the terms of the GPL from `http://immunix.com/`

10 Acknowledgments

We would like to thank the developers of the open source GCC compiler who made this work possible, DARPA for funding this work, Chris Wright and Seth Arnold for hard labor helping to debug, and Jane-Ellen Long of USENIX for help in final preparation of the paper.

References

[1] Anonymous. Once upon a free()... *Phrack*, 11(57), August 2001.

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *2000 USENIX Annual Technical Conference*, San Diego, CA, June 18-23 2000.

[3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Approach to Combat Buffer Overflows, Format-String Attacks, and More. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[4] "Bulba" and "Kil3r". Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[5] Matt Conover. w00w00 on Heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt, 1999.

[6] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.

[7] Crispin Cowan, Steve Beattie, RyanFinnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, Raleigh, NC, May 1999.

[8] Crispin Cowan, Tim Chen, Calton Pu, and Perry Wagle. StackGuard 1.1: Stack Smashing Protection for Shared Libraries. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998. Brief presentation and poster session.

[9] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[10] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000. Also presented as an invited talk at SANS 2000, March 23-26, 2000, Orlando, FL, http://schafercorp-ballston.com/discex}.

[11] Mark J. Cox, Ralf S. Engelschall, Stephen Henson, and Ben Laurie. openSSL. http://www.openssl.org/, March 2001.

[12] "Solar Designer". Non-Executable User Stack.

http://www.openwall.com/linux/.

[13] "Solar Designer". JPEG COM Marker Processing Vulnerability in Netscape Browsers. http://online.securityfocus.com/archive/1/71598, July 25 2000. Bugtraq.

[14] Casper Dik. Non-Executable Stack for Solaris. Posting to `comp.security.unix`, http://x10.dejanews.com/getdoc.xp?AN=207344316&CONTEXT=890082637.156735%9211&hitnum=69&AH=1}, January 2 1997.

[15] Igor Dobrovitski. Exploit for CVS double free() for Linux pserver. http://online.securityfocus.com/archive/1/309913, February 2 2003. Bugtraq.

[16] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, Washington, DC, August 2001.

[17] Intel. *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, 2002.

[18] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of USENIX Annual Technical Conference*, Monteray, CA, June 2002.

[19] Richard Jones and Paul Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html, July 1995.

[20] Michel Kaempf. [synnergy] - Sudo Vudo. http://online.securityfocus.com/archive/1/189037, June 6 2001. Bugtraq.

[21] BrianW. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[22] Greg McGary. Bounds Checking in C & C++ Using Bounded Pointers. http://gcc.gnu.org/projects/bp/main.html, 2000.

[23] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual*, second edition, 1984.

[24] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, London, England, January 2002. Also available at http://raw.cs.berkeley.edu/Papers/ccured_popl02.pdf.

[25] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[26] "Zen Parse". ADV/EXP: netkit less than 0.17 in.telnetd remote buffer overflow. http://online.securityfocus.com/archive/1/203000, August 10 2001. Bugtraq.

[27] "Zen Parse". Re: exploiting wu-ftpd. http://online.securityfocus.com/archive/82/244938, December 12 2001. Bugtraq.

[28] DavidA. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, first edition, 1990.

[29] Roldan Pozo and Bruce Miller. SciMark 2.0. http://math.nist.gov/scimark, June 20 2000.

[30] Alexander Snarskii. FreeBSD Stack Integrity Patch. ftp://ftp.lucky.net/pub/unix/local/libc-letter, 1997.

[31] Herman ten Brugge. Bounds Checking C Compiler. http://web.inter.NL.net/hcc/Haj.Ten.Brugge/, 1998.