Data Randomization

Cristian Cadar
Microsoft Research
Cambridge, UK
cristic@stanford.edu

Periklis Akritidis
Microsoft Research
Cambridge, UK
pa280@cl.cam.ac.uk

Manuel Costa
Microsoft Research
Cambridge, UK
manuelc@microsoft.com

Jean-Phillipe Martin
Microsoft Research
Cambridge, UK
jpmartin@microsoft.com

Miguel Castro
Microsoft Research
Cambridge, UK
mcastro@microsoft.com

Abstract

Attacks that exploit memory errors are still a serious problem. We present *data randomization*, a new technique that provides probabilistic protection against these attacks by xoring data with random masks. Data randomization uses static analysis to partition instruction operands into equivalence classes: it places two operands in the same class if they may refer to the same object in an execution that does not violate memory safety. Then it assigns a random mask to each class and it generates code instrumented to xor data read from or written to memory with the mask of the memory operand's class. Therefore, attacks that violate the results of the static analysis have unpredictable results. We implemented a data randomization prototype that compiles programs without modifications and can prevent many attacks with low overhead. Our prototype prevents all the attacks in our benchmarks while introducing an average runtime overhead of 11% (0% to 27%) and an average space overhead below 1%.

1 Introduction

Programs written in unsafe languages like C and C++ are vulnerable to attacks that exploit memory errors, for example buffer overflows and underflows [6, 43], dangling pointers [8], and double frees [33]. Attackers routinely exploit these errors to gain control over the execution of vulnerable programs or to force vulnerable programs to disclose confidential information.

We present *data randomization*, a new technique that provides probabilistic protection against these attacks by xoring data with random masks. Data randomization uses static analysis to partition instruction operands into equivalence classes according to the objects they may refer to. It then assigns a random mask to each class, and instruments code to xor data read from and written to memory with the operand's mask. This provides a probabilistic version of *write integrity* [9] and *read integrity*: writes and reads have unpredictable results when they access objects that they were not intended to (according to the analysis). Data randomization can be applied to C and C++ programs without modifications, it has high coverage with no false positives, and it has low space and time overhead.

We implemented data randomization by modifying a C compiler to run the static analysis and to generate instrumented code. To compute the equivalence classes, the compiler runs a points-to analysis [29] to determine the set of objects that each instruction operand may refer to in executions that do not violate memory safety. Then it places operands that may refer to the same object, according to the points-to analysis, in the same class. Each class is assigned a random mask that is used at runtime to xor all objects that are accessed through operands in the class.

The data randomization compiler adds instrumentation to *encrypt* values written to memory: it inserts an instruction before a write that xors the value being written with the mask of the destination operand. It also adds instrumentation to *decrypt* values read from memory: it inserts an instruction after a read that xors the value read with the mask of the source operand. This ensures that data is stored "encrypted" in memory where it is vulnerable to attacks, and that registers store a "plaintext" version of the data to enable processing with unmodified CPUs.

We generate new random masks at load time and patch the loaded binary to use the new masks. The compiler outputs a file with the locations of the masks in the binary to enable efficient patching.

Data randomization provides probabilistic read and write integrity. Attackers that exploit memory errors to write to objects in the wrong class cannot predict the values written because they are xored with a random unknown mask. Similarly, attackers cannot obtain confidential information by reading data from objects in the wrong class because the data is xored with a random un-

known mask.

It is interesting to compare data randomization with previously proposed randomization techniques. For example, instruction set randomization [34, 12] prevents code injection attacks by xoring the program text of the vulnerable program with a key unknown to the attacker. It requires hardware support to achieve low overhead. Data randomization can provide similar protection with low overhead. It ensures that code injected into a buffer cannot be decoded correctly by the processor because the instructions are xored with a key unknown to the attacker.

PointGuard [24] is superficially similar to data randomization but it only xors pointers and uses the same mask to xor all pointers. Therefore, it cannot prevent attacks that exploit memory errors to access non-pointer data and leaking any pointer value compromises the entire system. It may also fail to work on programs where pointers may be aliased with non-pointer data.

Another technique, which is widely used, is address space layout randomization (ASLR) [3, 13, 30, 38, 54]. ASLR randomizes the memory locations of data and code. ASLR is vulnerable to attacks that exploit relative offsets to overwrite memory locations and to attacks that place many copies of data chosen by the attacker in the address space of the vulnerable program. For example, heap spraying [46] places many copies of shell code in the heap of the target program. This ensures that when the attacker overwrites a code pointer there is a high probability of executing the shell code. Data randomization can reduce the probability of success of these attacks.

We evaluated the coverage of data randomization using a suite of attacks to test buffer overflow prevention techniques [53] and four real attacks on SQL server, ghttpd, nullhttpd, and stunnel. Data randomization was able to prevent all these attacks. We also evaluated the overhead introduced by data randomization using SPEC CPU and Olden benchmarks. Data randomization has low runtime overhead and very low space overhead: it had an average runtime overhead of 11% (0% to 27%) and an average space overhead below 1% in our benchmarks. On the nullhttpd web server instrumented with data randomization, peak throughput while serving static content from the SPECweb benchmark decreased by only 6%.

The rest of the paper is organized as follows. Section 2 presents an overview of data randomization. Section 3 discusses the static analysis that we use to compute equivalence classes and a safety analysis to avoid unnecessary instrumentation. Section 4 describes how the data randomization compiler instruments code. Section 5 discusses the runtime environment used by data randomization. Section 6 presents our experimental evaluation of data randomization, including its performance overhead and its effectiveness at blocking attacks. Section 7 discusses related work, and we conclude in Section 8.

2 Overview

Data randomization involves four components: static analysis, compile-time instrumentation, load-time instrumentation, and run-time randomization. We will use the example in Figure 1 to illustrate how all the components work. The example is a simplified remote shell server with a buffer overflow vulnerability that can be exploited with non-control-data attacks [18]. The example is inspired by a real attack on an SSH server [4].

```
1:  void ProcessConnection(connection *c) {
2:    cred_t user;
3:    char message[1024];
4:    int i = 0;
5:
6:    auth_user(&user, c);
8:    while (!end_of_message(c)) {
9:       message[i] = get_next_char(c);
10:      i++;
11:   }
12:   seteuid(user.user_id);
13:   ExecuteRequest(message);
14: }
```

Figure 1: Example code: simplified remote shell server with a buffer overflow vulnerability.

The function in Figure 1 is called when the server receives a new connection request. The function starts by calling `auth_user` to authenticate the user and to store the user credentials in `user`. Next, the function enters a loop that receives characters from the connection and stores them in the `message` buffer. After receiving the message, the program calls `seteuid` to impersonate the remote user and it calls `ExecuteRequest` to execute the command with the user's privileges. This function has a buffer overflow vulnerability in lines 8 – 10: by supplying a long message, an attacker can overflow the `message` buffer and overwrite the `user` variable. The attacker can thus supply an arbitrary user id (e.g., root) and the server will execute commands with the corresponding user's privileges. This is a non-control-data attack [18], since it does not force any unintended control-flow transfer in the program.

Data randomization uses static analysis to partition objects into equivalence classes such that objects that can be accessed through the same pointer are placed in the same class. For example, in Figure 1 the variables `message` and `user` are placed in separate classes. After the analysis, we assign a random mask to each equivalence class. Section 3 describes the static analysis and random mask assignment.

The compile-time component instruments instructions that write to or read from an object in memory to xor the

object with the corresponding mask. If the instruction is a write, the extra xor instruction effectively *encrypts* the written value. If it is a read, it *decrypts* the read value. The compiler also records the offsets of masks in the code. We describe this component in detail in Section 4.

The load-time component generates a new random mask for each equivalence class every time the program is loaded. Then it patches the loaded binary to use the new masks. This component simply reads the offsets recorded by the compiler and overwrites old mask values with the new ones.

In the example of Figure 1, accesses to the variables `message` and `user` are xored with distinct random masks. The instruction at line 9 encrypts the character it stores in `message` with the corresponding mask, and the instruction at line 12 decrypts the `user.user_id` value it reads with a different random mask. This does not prevent attackers from overflowing the `message` buffer and overwriting `user.user_id` but attackers can no longer write a value of their choice to `user.user_id`. Doing so would require attackers to know the xor of the random masks used by the program to write to the `message` buffer and to read from the `user.user_id` object.

This example illustrates the power of data randomization relative to other randomization techniques. Since there is no code injection, this attack would not be prevented by instruction set randomization techniques [12, 34]. PointGuard [24], which xors pointers with a random mask, would also fail to prevent this attack because no pointer is overwritten. The ASLR techniques that are widely deployed [3, 30] only randomize the base addresses of heap, stack, static data, and text areas. Therefore, they would not prevent this attack either. Even the comprehensive randomization technique described in [14] would likely fail to prevent this attack. This technique uses two stacks to segregate buffers from other variables but `message` and `user` are both buffers that would be placed in the same stack frame. Data randomization can prevent attacks like this and it can be used in production systems because it has low space and time overhead.

3 Static analysis

We used the Phoenix compiler framework [39] to implement the static analysis that computes equivalence classes for data randomization. The analysis operates on Phoenix's medium level intermediate representation (MIR), which is still independent of the target processor. Figure 2 shows the MIR for the vulnerable C code in Figure 1.

```
     _i  = ASSIGN 0
     CALL &_auth_user, &_user, _c
$L6: t274 = CALL &_end_of_message, _c
     t275 = COMPARE(NE) t274, 0
     CONDITIONALBRANCH(True) t275, $L7, $L8
$L8: t278 = CALL &_get_next_char, _c
     t277 = ADD &_message, _i
     [t277] = ASSIGN t278
     _i = ADD _i, 1
     GOTO $L6
$L7: CALL &_seteuid, _user+4
     CALL &_ExecuteRequest, &_message
```

Figure 2: Example vulnerable code in medium level intermediate representation (MIR).

3.1 Computing equivalence classes

We use an inter-procedural points-to analysis due to Andersen [10] that is flow and context insensitive but scales to large programs. It computes a points-to set for each pointer operand, which is the set of logical objects the pointer may refer to. The analysis is conservative: it includes all objects that the pointer may refer to at runtime but it may include additional objects. Our implementation is similar to the one described in [29] but it is field-insensitive rather than field-based (i.e., it does not distinguish between the different fields in a structure, union, or class). WIT [9] uses the same points-to analysis.

The points-to analysis makes a global pass over all source files to collect *subset constraints*. For example, each assignment $x = y$ results in a subset constraint $x \supseteq y$, which means that the set of possible values of x contains the set of possible values of y. We use Phoenix to compile each source file to MIR and write all subset constraints in the MIR to a file. After this global pass, the analysis reads the constraints file and computes the points-to sets by iterating over all the constraints until it reaches a fixed point. Then, it stores the points-to sets in a file. In the example in Figure 2, there is only one pointer operand `[t277]`. The points-to analysis determines that it points to the `message` array. We use `[p]` to denote a dereference of `p`.

We use the points-to sets to partition instruction operands into equivalence classes. We place two operands in the same equivalence class if they can refer to the same object at runtime according to the points-to analysis. This constraint ensures that our instrumentation does not change program behavior in executions that do not violate memory safety because all reads and writes to an object are xored with the same mask. Under this constraint, we maximize the number of equivalence classes to increase the number of attacks that we can prevent.

We compute equivalence classes using an iterative process. Initially, there is a separate equivalence class for each points-to set: the initial equivalence class for a points-to set $p \rightarrow \{o_1, ..., o_n\}$ is $\{[p], o_1, ..., o_n\}$. Then we merge equivalence classes that intersect until we

reach a fixed point. We use an union-find data structure [20] to compute the classes efficiently. After processing the points-to sets, we iterate over all objects that are not referenced by any pointer. These are variables that are never accessed indirectly, for example, `_i` in Figure 2. We place each of these objects in a separate class. For example, {`_i`} and {`[t277], message`} are two of the equivalent classes computed for the code in Figure 2.

This analysis assumes that correct programs do not use pointer arithmetic to navigate between independent objects in memory. For example in Figure 2, the analysis assumes that correct programs will not use `t277`, which is a pointer into the `message` array, to write to `user`. Existing compilers already make this assumption when implementing several standard optimizations. Therefore, this assumption applies to the vast majority of programs. However, it is precisely this assumption that is violated by most attacks that exploit memory errors. Data randomization can prevent attacks that violate this assumption without false positives.

3.2 Avoiding instrumentation with safety analysis

We also run a safety analysis at compile time to identify equivalence classes that we do not need to instrument. This is an important performance optimization.

The safety analysis classifies instruction operands as safe or unsafe: an operand is safe if runtime accesses to the operand can never violate memory safety. The analysis marks safe all temporary, local variables, or global operands in the MIR. These operands are safe because they always refer to registers or to a constant number of bytes starting at a constant offset from the frame pointer or the data segment. In the example in Figure 2, all operands are safe except `[t277]`.

In addition, the safety analysis runs a simple intra-procedural pointer-range analysis to compute writes and reads through pointers that are always in bounds. These pointer operands are marked safe. Our pointer-range analysis is a simplified version of the one described in [57]. It collects sizes of aggregate objects (e.g., structs) and arrays that are known statically. Then it uses symbolic execution to compute the minimum size of the objects each pointer can refer to and the maximum offset of the pointer into these objects. When the analysis cannot compute this information or the offset can be negative, it conservatively assumes a minimum size of zero. Our current implementation can track constant offsets and offsets that can be bound using Phoenix's built-in value range information for numeric variables. Given information about the minimum sizes, the maximum offsets, and the size of the intended accesses, the analysis checks if accesses through the pointer are always in bounds. If they are, the corresponding pointer operand is marked safe. We used a similar safety analysis to improve the performance of WIT [9].

We do not instrument reads and writes to objects in an equivalence class when all the instruction operands in the class are safe. We say that the objects referred to by these operands are safe. These objects are stored in "plaintext" in memory. But we still ensure that accesses that violate read or write integrity have unpredictable results because accesses to safe operands cannot violate memory safety and accesses to unsafe operands are instrumented using random masks. In the example in Figure 2, all objects are safe except for `message` and `user` (because of an unsafe access inside `_auth_user` not shown in the figure).

3.3 Assigning masks to classes

We need to select masks for classes carefully to ensure that every byte in an object is consistently xored with same byte mask. The issue is that there may be operands of different sizes in the same equivalence class and the accesses to objects in the class may have different alignments at runtime. For example, an integer array may be accessed using a `char*` variable.

We assign random masks of different sizes to equivalence classes. The size of the mask for a class C is the minimum size of an operand in C. For example, if C has two elements `[p]` and `[q]` with size two and four bytes, the mask size for class C is two bytes. We use a maximum mask size of four bytes for masks. In order to compute the mask size of each class, we record the size of operands during the pass that collects points-to constraints. In our example, in Figure 2, the mask size for the `[t277]`'s class is one byte and the mask size for `_user+4` is four bytes.

After computing the mask sizes, we generate a random mask with the right size for each class. Reads and writes to objects in the class are xored with this mask. If the size of an operand is greater than the mask size of its class, the mask is extended by replicating it up to operand size. Going back to our previous example, if `[q]` has size four bytes and the mask for its class is `0x3210` of size two bytes, the extended mask is `0x32103210`. This ensures that the bytes in an object that can be accessed through `[q]` or `[p]` are consistently encrypted and decrypted.

Provided memory accesses are aligned, this assignment of masks to classes ensures that we can determine the masks to use for instrumentation statically. Portable programs satisfy this assumption because many architectures raise exceptions when unaligned accesses are issued or incur significant performance penalties for unaligned accesses.

We also experimented with a version of data randomization that assigns four byte masks to each class. In this

case, we can still determine statically the masks to use for access sizes of four or more bytes. However, the instrumentation for access sizes of one and two bytes must use the alignment of the target address to determine dynamically the mask to use. For example, a one byte access would be xored with the i-th least significant byte in the mask, where i is the value of the two least significant bits of the address being accessed. To make the example more concrete, if the mask for a class is `0xDDCCBBAA`, a one byte access to address `0x00200000` would be xored with byte `0xAA` and a one byte access to address `0x0020002` would be xored with byte `0xCC`. This version is more secure because it ensures a minimum size of 32-bits for the masks the attacker must guess. In most of our applications, the overhead of the two versions is the same but in some applications the version that always uses four byte masks introduces a significant overhead. We focus on the first version in the rest of the paper.

4 Instrumentation

After computing the equivalence classes and their masks, the data randomization compiler generates code with instrumentation to encrypt and decrypt memory accesses. We implemented a Phoenix [39] plug-in to insert the instrumentation. Since the static analysis works on MIR, we instrument the code by transforming MIR. This avoids the complexity of mapping instruction operands between different code representations. Transforming a lower level intermediate representation would provide more control over the generated code, but the current version of the static analysis does not work on lower level intermediate representations.

We start by presenting the code transformation that we use in the general case. Then we describe the transformation for function calls. We end by describing the instrumentation for the version of data randomization that assigns four byte masks to all equivalence classes.

4.1 General case

The compiler adds instrumentation to decrypt values read from memory and to encrypt values written to memory. It inserts instructions that xor a value that was read from memory with the mask of its source operand and instructions that xor a value that is about to be written to memory with the mask of the destination operand. For example, it transforms an MIR instruction `o1 = OPERATION o2,o3` into:

```
t2    = BITXOR o2, m2
t3    = BITXOR o3, m3
t1    = OPERATION t2, t3
o1    = BITXOR t1, m1
```

where `o1`,`o2`, and `o3` are unsafe operands, `t1`,`t2`, and `t3` are new temporaries, and `m1`,`m2`, and `m3` are constants with the mask values for the operands. The machine model for the Phoenix MIR provides an infinite number of temporaries that are assigned to registers in a later compilation stage. If any of the operands is safe, we can remove instrumentation for that operand. For example, if `o1` and `o2` are safe the instrumented code is:

```
t3    = BITXOR o3, m3
o1    = OPERATION o2, t3
```

This instrumentation ensures that operations are performed on plaintext copies of the objects and that memory copies of unsafe objects are encryted.

Some of the temporaries that we insert during the instrumentation may be spilled to memory by the compiler. This is not a problem for data randomization because the memory accesses generated by the compiler are safe and values written by an unsafe access to a spilled temporary are encrypted with a mask unknown to the attacker. This was a problem for PointGuard [24].

Usually Phoenix can generate efficient code for the transformed MIR but we developed a number of optimized transformations for common cases. These transformations achieve significant speedups by reducing the number of extra temporaries or instructions. For example, we avoid adding an extra temporary in the common case of loads from memory. We transform `t1 = ASSIGN [t2]` into:

```
t1    = ASSIGN [t2]
t1    = BITXOR t1, m2
```

In some cases, we do not need extra temporaries or instructions because we can modify the value of a constant operand. For example, we instrument `[t1] = ASSIGN c` by replacing the constant `c` by the result of xoring `c` with `[t1]`'s mask. We can use the same transformation for instructions that compare whether an unsafe object is equal to a constant.

We cannot use the general transformation directly to instrument floating point operands and structure operands larger than eight bytes because Phoenix's `BITXOR` operation does not support these operand types. We allocate new local variables to hold copies of source operands of these types instead of temporaries. Then we call a function to xor the memory copies of these variables or the destination operand. This function is inlined for speed and we can avoid copying operands to local variables in some cases.

4.2 Function calls

Instrumentation of a function call has several steps. First, we insert instructions to decrypt unsafe actual arguments before we call the function and to encrypt the return value if it is stored in

an unsafe object. For example, the instruction `o1 = CALL &_function, o2, o3` is transformed into:

```
t2    = BITXOR o2, m2
t3    = BITXOR o3, m3
t1    = CALL &_function, t2, t3
o1    = BITXOR t1, m1
```

where `o1`,`o2`, and `o3` are unsafe operands, `t1`,`t2`, and `t3` are new temporaries, and `m1`,`m2`, and `m3` are constants with the mask values for the operands. We also insert instructions to encrypt unsafe formal arguments at the beginning of each function and to decrypt return values at function exit.

This transformation allows us to decouple the instrumentation at the caller and the callee. It enables instrumentation of indirect calls without constraining the masks assigned to argument operands. Another important advantage is that it simplifies interaction with uninstrumented code because arguments and return values are passed unencrypted. We can invoke uninstrumented functions that do not take pointer arguments or return pointer values.

4.3 Instrumentation with fixed size masks

As discussed in Section 3.3, we can improve the security of data randomization by assigning four byte masks to all classes. The instrumentation for this version is identical for operands with size greater than or equal to four bytes. But it requires complex instrumentation for accesses of one and two bytes. For example, `t2 = BITXOR o2, m2` becomes:

```
t21   = ASSIGN &o2
t21   = BITAND t21, 0x3
t21   = SHIFTLEFT t21, 0x3
t22   = ASSIGN m2
t22   = SHIFTRIGHT t22, t21
t2    = CONVERT t22
t2    = BITXOR t2, o2
```

where the first five instructions compute which byte of the mask to use based on the alignment of the byte being read. This complex instrumentation does not directly add extra memory accesses to data or branches but it requires two extra registers. The increased register pressure can indirectly cause poor performance due to extra memory accesses.

In applications without many byte accesses, the more secure version of data randomization has good performance. However, it can have high overhead in applications with many byte accesses. We expect the overhead to decrease in architectures with more registers like the new 64-bit extensions of Intel processors. Additionally, it should be possible to instrument loops efficiently by rotating a mask at each iteration rather than computing the alignment for each runtime access.

4.4 Load-time instrumentation

We generate new random masks when a program is loaded. To enable efficient re-assignment of masks to classes, the compiler emits a file with the byte offset, size of each immediate operand containing a mask, and the mask used. The loader uses this information to patch the loaded binary: it reads the old immediate value of a mask, looks up the corresponding new value, and overwrites the old value with the new one in the binary.

4.5 Example

Figure 3 shows our example vulnerable code with instrumentation. The out-of-bounds writes to the `message` array are xored with the random mask `0xF3` and the value read from the user identifier field of the `_user` structure is xored with the random mask `0xACFB4711`. Therefore, to write a chosen 32-bit user identifier, attackers must guess 32 random bits. They must guess the xor of the random masks `0xACFB4711` and `0xF3F3F3F3`. If user identifiers are small positive integers, attackers may choose to overwrite only the least significant bytes of the user identifier field to reduce the number of random bits they must guess. In the worst case, the attacker must still guess eight random bits.

```
     _i  = ASSIGN 0
     CALL &_auth_user, &_user, _c
$L6: t274 = CALL &_end_of_message, _c
     t275 = COMPARE(NE) t274, 0
     CONDITIONALBRANCH(True) t275, $L7, $L8
$L8: t278 = CALL &_get_next_char, _c
     t277 = ADD &_message, _i
     t300 = BITXOR t278, 0xF3
     [t277] = ASSIGN t300
     _i = ADD _i, 1
     GOTO $L6
$L7: t301 = BITXOR _user+4, 0xACFB4711
     CALL &_seteuid, t301
     CALL &_ExecuteRequest, &_message
```

Figure 3: Example vulnerable code in medium level intermediate representation (MIR) with instrumentation.

This instrumentation also makes debugging hard but we believe that it would be possible to modify a debugger to use the appropriate masks when vieweing or changing the values of variables.

5 Runtime

The runtime environment for data randomization provides an initialization function and wrappers for library

functions and operating system calls. Our compiler inserts a call to the initialization function at the beginning of `main`. This function xors global variables and the arguments to `main` with the approriate masks.

Many attacks make use of libraries when exploiting vulnerabilities. For example, string manipulation functions are notorious for their use in exploits of buffer overflow vulnerabilities. To increase data randomization's coverage, we provide wrappers for C library functions and operating system calls that receive or return pointers. We have written wrappers for all the library functions used in our test cases to ensure that all accesses to unsafe objects are instrumented.

To implement a wrapper for a library function, one must write a wrapper function and describe the subset constraints that calling the function adds to the points-to analysis (if there are any). We instrument the code to call the wrapper instead of the original function and to supply the wrapper with the masks for the objects that the function reads and writes. In most cases, the wrapper simply xors the objects before they are read by the library function, calls the function, and then xors objects written by the function before returning. For efficiency, we provide our own implementation for some library functions. Wrappers for functions in the standard libraries can be implemented once and then can be reused by any new application without further modifications.

Figure 4 shows a wrapper for `strchr` that decrypts the string buffer on the fly. This wrapper takes three arguments: a mask `mask`, a string `s`, and a character `c`. The wrapper iterates over the contents of string `s`, decrypting each character with the given mask, and comparing the decrypted value with the character `c`.

```
char* strchr_DataRand(uint mask, uchar* s, uint c)
{
  uchar cmask =
    (uchar)((mask >>((((uint)s)%4)*8)) & 0xff);

  while ((*s^cmask) && (*s^cmask) != (char)c)
  {
    s++;
    cmask =
      (uchar)((mask >>((((uint)s)%4)*8)) & 0xff);
  }

  if ((*s ^ cmask) == (char)c)
    return (char *)s;

  return NULL;
}
```

Figure 4: Example wrapper for strchr.

There are two interesting things to note about this wrappper. First, we add context-sensitive subset constraints for calls to wrapped library functions. For example, a call of the form `x = strchr(s, c)` is treated as the assignment `x = s` by the points-to analysis. This improves the precision of our analysis. A context-insensitive treatment of library functions would put all string arguments to `strchr` in the same class. Second, the wrapper uses the address of the character being read to select the mask byte used to decrypt it. All our wrappers do this. This allows us to disregard access sizes inside the libraries when computing the mask sizes for equivalence classes (as described in Section 3.3). Additionally, we can use a mask size of 4-bytes for strings that are exclusively manipulated through library functions. Since many strings are exclusively manipulated through library functions, this improves security by increasing the number of bits attackers must guess to launch a successful attack.

Providing wrappers for all library and system calls is important to improve coverage but it is not strictly necessary. When a program calls a library function for which we have no source code and no wrapper, we do not instrument accesses to objects that are reachable from a pointer that is passed to or received from this library function. We determine these objects by running a reachability analysis on the output of our points-to analysis and we assign mask zero to the equivalence class that contains these objects.

6 Evaluation

We ran experiments to evaluate the overhead of our implementation of data randomization and its effectiveness at preventing a range of real and synthetic attacks. This section presents our results. Data randomization prevents all the attacks in our tests and its CPU and memory overhead are low for all the applications tested.

6.1 Performance overhead

In our first experiment, we measured the overhead added by data randomization to seven programs from the SPEC CPU 2000 benchmark suite [50] (gzip, vpr, mcf, crafty, parser, bzip2 and twolf), and to nine programs from the Olden [15] benchmark suite (bh, bisort, em3d, health, mst, perimeter, power, treeadd, and tsp). We chose these programs to facilitate comparison with other techniques that have been evaluated using the same benchmark suites.

We compared the running time and peak physical memory usage of the programs compiled using Phoenix [39] with and without data randomization. We compiled the programs with options -O2 (maximize speed), -fp:fast (fast floating point model), and -GS- (no stack guards). When building binaries with data randomization, we linked with our runtime (see Section 5). We ran the experiments on Windows Vista Enterprise, on an idle Dell Optilex 745 Workstation with a 2.46GHz Intel

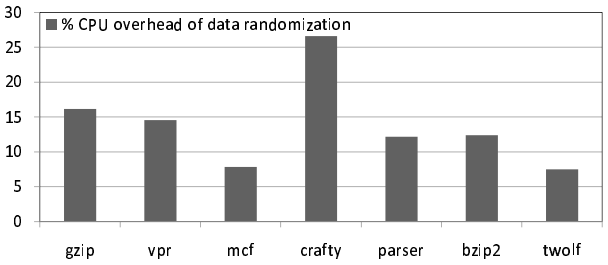

Figure 5: Execution time overhead added by data randomization for SPEC CPU (relative to the execution time without instrumentation).

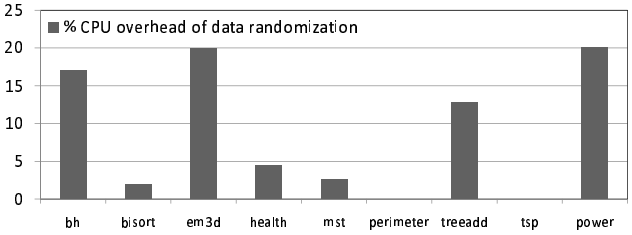

Figure 6: Execution time overhead added by data randomization for Olden (relative to the execution time without instrumentation).

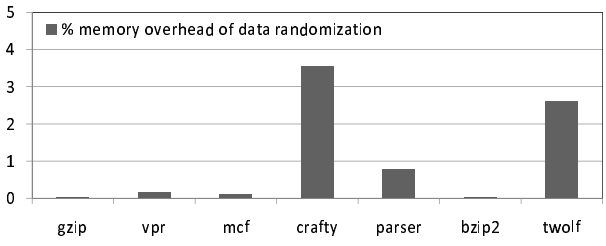

Figure 7: Memory overhead of data randomization for SPEC applications (relative to the memory used without instrumentation).

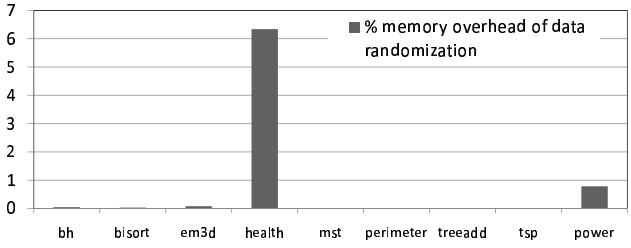

Figure 8: Memory overhead of data randomization for Olden applications (relative to the memory used without instrumentation).

Core 2 processor and 2GB of memory. For each experiment, we present the median of 3 runs. The variance in the results was negligible.

Figures 5 and 6 show the CPU overhead on SPEC and Olden applications with data randomization. For SPEC, the average overhead is 14% and the maximum is 26%. For Olden, the average overhead is 8% and the maximum is 20%. To put this overhead in perspective, WIT [9] achieves a similar overhead: an average overhead of 10% and a maximum of 23% in SPEC, and an average overhead of 5% and a maximum of 17% in Olden. WIT provides deterministic write and control flow integrity but does not provide read integrity. The bounds checking technique described in [26], which is the fastest we know, has an average overhead of 15% and a maximum overhead of 69% in the Olden benchmarks [1]. The comprehensive ASLR in [14] reports an overhead of 17% for gzip whereas data randomization has a similar overhead of 16%.

We also ran this experiment with the more secure version of data randomization that assigns four byte masks to every class (as discussed in Sections 3.3 and 4.3). The average runtime overhead across all the benchmarks increased from 11% to 18%. The runtime overhead did not change for the 11 benchmarks that do not have many one and two byte accesses but it increased for the other five.

Figures 7 and 8 show the memory overhead introduced by data randomization on SPEC and Olden applications. The overhead is very low for all applications. For SPEC, the average memory overhead is 1% and the maximum is 3.5%. For Olden, the average overhead is 1% and the maximum is 6.3%. This was expected because data randomization does not introduce additional data structures or padding. In contrast, WIT has an average space overhead of 14% in SPEC and 12% in Olden.

We observed an interesting anomaly in this experiment. In our initial measurements, mcf had a memory overhead of 23%. We found that this was because of a large memory allocation using `calloc`. Data randomization was touching all the memory to xor it with the appropriate mask. Whereas the version without the instrumentation did not access all the memory. We expected the C runtime to zero the allocated memory but it relies on the operating system to zero allocated pages when they are first accessed. This is easy to fix by xoring pages when they are first accessed.

We also measured the increase in the size of SPEC and Olden binaries compiled with data randomization. Figures 9 and 10 show the results. Instrumented SPEC binaries are 16% larger on average, while Olden binaries are 28% larger on average. This is a small increase in code size and it is similar to the increase in code size introduced by WIT.

We also measured the peak throughput of the nullhttpd web server compiled with data randomization. The server ran on the same machine as the previous experiments and we loaded it with static requests for a 1KB

[1] These results are computed relative to the optimized baseline in [26]. We believe this is more appropriate than reporting the 12% overhead relative to the unoptimized baseline.

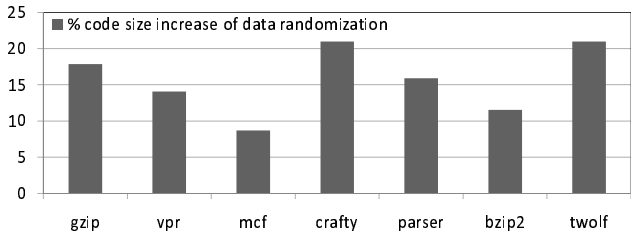

Figure 9: Code size increase of SPEC executables compiled with data randomization (relative to the size without instrumentation).

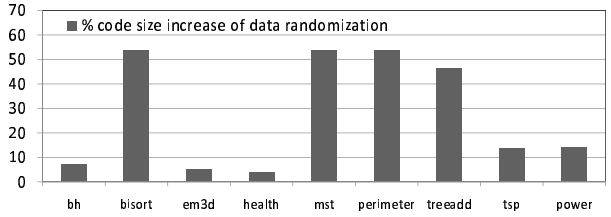

Figure 10: Code size increase of Olden executables compiled with data randomization (relative to the size without instrumentation).

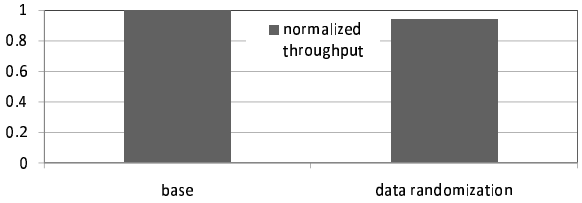

Figure 11: Peak throughput of the nullhttpd web server when serving a 1KB file from SPECweb.

file from the SPECweb [50] benchmark. We used a small static file request to avoid masking our overhead with the effects of disk access or process creation on the server. The load was generated from simulated clients on a HPxw4600 workstation with a 2.66GHz Intel Core2 Duo CPU and 4GB of memory, running Windows Vista Enterprise, over a DLink 100Mbps Ethernet switch. The results in Figure 11 show that peak throughput decreases only by 6%.

6.2 Effectiveness against attacks

To evaluate the effectiveness of data randomization at preventing attacks, we used a benchmark with synthetic exploits [53] and several exploits of real vulnerabilities in existing programs. This section describes the programs and the vulnerabilities. Then it presents an analysis of the security afforded by data randomization.

6.2.1 Synthetic exploits

We ran the benchmark described in [53] that has 18 control-data attacks that exploit buffer overflow vulnerabilities. The attacks are classified according to the technique they use to overwrite control-data, the location of the buffer they overflow, and the control-data they target. There are two techniques to overwrite control-data. The first overflows a buffer until the control-data is overwritten. The second overflows a buffer until a pointer is overwritten, and uses an assignment through the pointer to overwrite the control-data. The attacks can overflow buffers located in the stack or in the data segment, and they can target four types of control-data: the return address on the stack, the old base pointer on the stack, and function pointers and longjmp buffers in the stack or in the data segment. Table 1 shows that data randomization can prevent all the attacks in the benchmark.

Attack	Target data structure
direct overwrite on stack	parameter function pointer parameter longjmp buffer return address old base pointer function pointer longjmp buffer
direct overwrite on data segment	function pointer longjmp buffer
overwrite through stack pointer	parameter function pointer parameter longjmp buffer return address old base pointer function pointer longjmp buffer
overwrite through data segment pointer	return address old base pointer function pointer longjmp buffer

Table 1: Synthetic attacks prevented by data randomization.

6.2.2 Real vulnerabilities

In our final experiment, we tested data randomization's ability to prevent attacks with a set of real vulnerabilities in real applications: `SQL server`, `Ghttpd`, `Nullhttpd`, and `Stunnel`.

`SQL server` is a relational database from Microsoft that was infected by the infamous Slammer [40] worm. The vulnerability exploited by Slammer causes `sprintf` to overflow a stack buffer. Data randomization prevents the attack because the wrapper for `sprintf` randomizes the data that overwrites the current stack frame, including the return address. This causes the server to exit when freeing a local variable that was overwritten. Should the return instruction be reached, the server would jump to an invalid program location and crash.

`Ghttpd` is an HTTP server with several vulnerabilities [1]. The vulnerability that we chose is a stack buffer overflow when logging `GET` requests inside a call to `vsprintf`. Data randomization prevents the attack

because the wrapper for `vsprintf` randomizes the value written by the attacker into the return address, causing the server to crash when the return address is used.

`Nullhttpd` is another HTTP server. This server has a heap overflow vulnerability that can be exploited by sending HTTP `POST` requests with a negative content length field [2]. These requests cause the server to allocate a heap buffer that is too small to hold the data in the request. While calling `recv` to read the `POST` data into the buffer, the server overwrites the heap management data structures maintained by the C library. This vulnerability can be exploited to overwrite arbitrary words in memory. We attacked `NullHttpd` using the technique described in [18]. The attack works by corrupting the CGI-BIN configuration string. This string identifies a directory holding programs that may be executed while processing HTTP requests. Therefore, by corrupting it, the attacker can force `NullHttpd` to run arbitrary programs. This is a non-control-data attack because the attacker does not subvert the intended control-flow in the server. Data randomization prevents the attack because the wrapper for `recv` randomizes the values written over the heap management data structures. This causes the server to crash when the values are used.

`Stunnel` is a generic tunnelling service that encrypts TCP connections using SSL. We studied a format string vulnerability in the code that establishes a tunnel for SMTP [5]. An attacker can overflow a stack buffer by sending a message that is passed as a format string to the `vsprintf` function. Data randomization prevents the attack because the wrapper for `vsprintf` randomizes the value written by the attacker into the return address.

6.2.3 Security analysis

This section showed that data randomization can stop existing real exploits with low runtime overhead. We now present a discussion of possible attacks against data randomization. We assume that attackers know the code of a vulnerable program, and that they can supply arbitrary inputs to the program. We assume they know the equivalence classes but do not know the masks used to randomize memory accesses. These masks are generated each time the program starts and we assume that attackers do not have access to the operating system process running the program. These assumptions are a good fit for a network service, for example.

Since data randomization does not remove memory errors from the program, the attacker can craft inputs that cause an instruction to write or read memory locations unintended by the programmer. Unlike ASLR, data randomization does not prevent attacks by making it hard for attackers to access a chosen memory location. Instead, it makes the result of accessing that location unpredictable. This unpredictability depends on two factors.

First, if the the read and write accesses used by the attacker to access the target location are in the same equivalence class, the attack succeeds. Figures 12 and 13 show the number of distinct equivalence classes with unsafe accesses for SPEC and Olden applications, respectively. All SPEC applications have a reasonably high number of distinct classes, except mcf. Olden applications have a small number of classes because they are small benchmarks. We could increase the number of classes by using more precise points-to analysis [37]. But our current analysis was sufficiently precise to thwart all the attacks we tested, i.e., the read and write accesses in an attack were in different classes.

It is important to note that objects that should never be accessed by unsafe instructions (according to the analysis) are always in a separate class. Therefore they are protected regardless of the precision of the points-to analysis. This includes important attack targets like return addresses, exception handler pointers, and dynamic linking data structures, as well as most local variables.

Second, the unpredictability of an access to a target location depends on the size of the random masks used to access the location. If the attack must write a chosen value to the target location or read the value stored at the chosen target location, the attack succeed with probability $2^{-min(o,max(w,r))}$, where o is the number of bits in the target location, w is the bit size of the mask used to xor the write access performed by the attacker, and r is the bit size of the mask used by correct accesses to the target location. Figures 14 and 15 show the distributions of mask sizes used in SPEC and Olden applications, respectively. The results show that the majority of accesses use 4-byte masks. Furthermore, using more precise points-to analysis [37] would increase the average mask size. We could also guarantee that all accesses use 4-byte masks with the version of the instrumentation described in Section 4.3 but the overhead would increase.

Some attacks perform partial overwrites of pointers, for example, they overwrite the least significant byte of a pointer to defeat ASLR. Attacks that only need to write eight bit values have a probability of success of 2^{-8} regardless of the mask sizes. This probability is still lower than the probability of success with deployed ASLR techniques. It is interesting to note that data randomization can provide this protection even for direct overwrites of non-pointer data that is security critical. ASLR [3, 13, 54] does not provide this protection.

Some attacks do not require complete control on the value written to a target location. For instance, heap spraying [46] attacks create many copies of shellcode [6] in the heap of the target program and overwrite a code pointer, e.g., a function pointer. If there are enough copies of the shellcode, the attack will succeed for most

Application	Vulnerability	Exploit
NullHttpd	heap-based buffer overflow	overwrite cgi-bin configuration data
SQL Server	stack-based buffer overflow	overwrite return address
STunnel	format string	overwrite return address
Ghttpd	stack-based buffer overflow	overwrite return address

Table 2: Real attacks detected by data randomization.

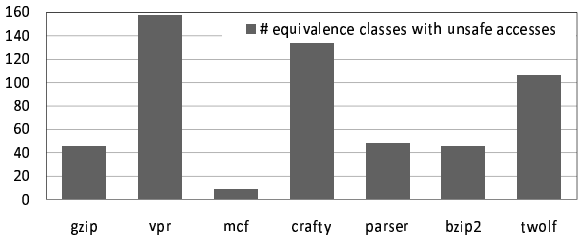

Figure 12: Number of equivalence classes with unsafe accesses in SPEC applications.

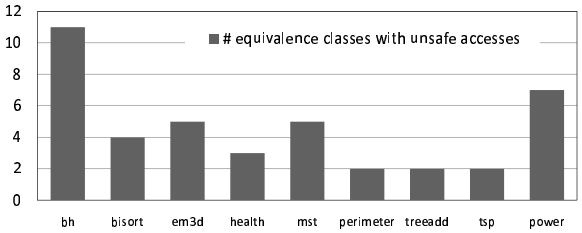

Figure 13: Number of equivalence classes with unsafe accesses in Olden applications.

values written to the pointer. Therefore, this attack can bypass ASLR. But data randomization prevents the attack because the shell code is xored with a random mask unknown to the attacker. Data randomization provides the same protection as instruction set randomization [34, 12] in this case but with low overhead and without hardware support.

Information leakage attacks [27, 48] are also of particular concern for randomization approaches. It is important not to leak the masks used to randomize memory accesses. It is hard to evaluate the probability of successfully exploiting a vulnerability to leak information about the randomization masks, but ASLR implementations have been subjected to this type of attack [27].

Data randomization may also be subjected to brute force guessing attacks as described in [45]. In most cases, we believe that the number of random bits the attacker must guess is large enough to prevent these attacks. Moreover, brute force attacks often cause visible anomalies, such as crashing the vulnerable applications; when these anomalies are observed, countermeasures such as automatic filtering [21] can be deployed to thwart the brute force attack.

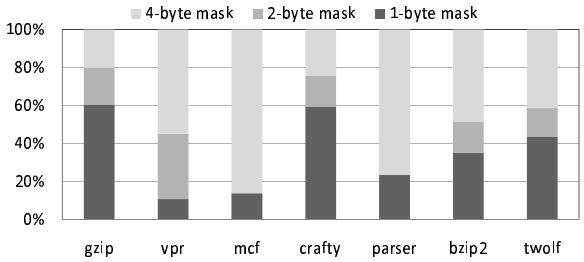

Figure 14: Fraction of static memory accesses with 4-byte, 2-byte and 1-byte random masks, in SPEC applications.

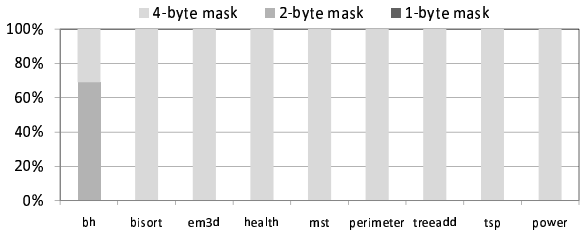

Figure 15: Fraction of static memory accesses with 4-byte, 2-byte and 1-byte random masks, in Olden applications.

7 Related work

Many techniques have been proposed to protect C and C++ programs from memory error exploits. Several tools find vulnerabilities by analyzing the source code of applications [51, 36, 17]. These tools have been very successful at removing vulnerabilities from software before it ships, but they are not sufficient because they are imprecise: they can miss vulnerabilities and they raise false alarms.

Memory safe dialects of C, such as CCured [41] and Cyclone [31] can prevent all memory errors but they require significant changes to the source code of applications, and they require major changes to the C runtime, e.g., they require a garbage collector.

Other techniques can be applied to C and C++ programs without modifications. Some techniques defend from attacks that overwrite specific targets, such as return addresses or other control data (e.g., [49, 11, 19, 25, 47]), or that exploit specific vulnerabilities, such as format string vulnerabilities (e.g., [23]). These techniques have

low overhead but there are memory error exploits that they cannot catch [53]. In particular, techniques inspired by StackGuard [25] are widely used, but they provide no protection from overflows of heap and static variables [43].

Other techniques provided higher coverage at the cost of additional overhead. Several systems detect attacks using dynamic taint analysis, e.g., [42, 22], which can prevent many attacks that exploit memory errors and other types of attacks. They work with binaries and do not require source code. However, their overhead is several orders of magnitude larger than data randomization's. Xu *et al* [55] describe a dynamic taint analysis technique that is implemented as a source-to-source transformation on C programs. Their overheads are an order of magnitude lower than previous techniques but they are still above 100% when preventing memory error exploits on CPU-intensive benchmarks.

There are several bounds checkers for C. For example, the Jones and Kelly [32] bounds checker does not require changes to the pointer format. It instruments pointer arithmetic to ensure that the result and original pointers point to the same object. To find the target object of a pointer, it uses a splay tree that keeps track of the base address and size of heap, stack, and global objects. CRED [44] is similar but provides support for some common uses of out-of-bounds pointers in existing C programs. These techniques have high overhead, for example, CRED can slow down applications by up to a factor of 12. Xu *et al* [56] describe a technique that improves the coverage of the previous bounds checkers and reduces their overhead. The technique of Dhurjati *et al* [26] is similar to CRED but introduces several optimizations that reduce runtime overhead dramatically. This technique has an average overhead of 15% and a maximum overhead of 69% in the Olden benchmarks.

The concept of control-flow integrity was introduced in [7, 35]. However, attackers can exploit memory errors to execute arbitrary code without violating control-flow integrity. There are examples of several attacks of this type in [18]. CFI [7] and Program Shepherding [35] cannot detect this type of attack. Data randomization provides probabilistic protection from these attacks and it has lower overhead. Data-flow integrity (DFI) [16] protects programs by enforcing data-flow relations between CPU instructions. For each instruction that reads a value, it uses static analysis to compute the instructions that are allowed to write the value. Then it instruments writes and reads to ensure that the values read at runtime were written by allowed instructions. Its coverage is similar to data randomization's but its average overhead on the SPECint benchmarks is 104%. In a companion paper [9] we propose *Write Integrity Testing*(WIT) a technique that is based on the same static analysis as data randomization. WIT has high coverage and low overhead, but it can't prevent some attacks detected by data randomization because it doesn't instrument reads. Furthermore, WIT's memory overhead is higher than data randomization's.

Several techniques are based on the idea of randomizing different aspects of computer programs [28]. PointGuard [24] randomizes pointer values in a manner similar to data randomization, but doesn't protect non-pointer data. In addition, PointGuard uses a single mask for all pointers, so leaking any pointer value compromises the entire system. Address space layout randomization (ASLR) randomizes the locations of code and data in memory to make it harder for attackers to target specific objects [3, 13, 54]. The best such technique that we know [14] randomizes absolute and relative locations of all memory-resident objects and combines ASLR with other buffer overflow mitigation techniques. It has an overhead similar to data randomization's. Data randomization can prevent some attacks that can bypass ASLR, such as direct overwrites of security-critical data and heap spraying [46]. Data randomization can also be combined with ASLR to make it more resistant to attacks [27, 45, 52]. Other techniques do instruction set randomization (ISR) [12, 34] but they have high overhead without hardware support, and they cannot prevent some attacks prevented by data randomization.

8 Conclusion

We presented data randomization, a new technique that provides probabilistic protection from memory error exploits. Data randomization uses static analysis to partition memory accesses into different classes according to the objects that they may access. Data randomization then assigns a distinct random mask to each class and, at runtime, xors the data read/written from/to memory with the corresponding mask. Therefore, memory accesses that violate the results of the analysis, i.e., that access unintended objects, have unpredictable results. Our results show that data randomization can block a broad range of attacks, while introducing an average runtime overhead of 11% and an average space overhead below 1%.

References

[1] GHttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960.

[2] Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774.

[3] PaX. `http://www.grsecurity.net/`.

[4] SSH CRC-32 Compensation Attack Detector Vulnerability. http://www.securityfocus.com/bid/2347.

[5] STunnel Client Negotiation Protocol Format String Vulnerability. http://www.securityfocus.com/bid/3748.

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

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity: Principles, implementations, and applications. In *ACM CCS*, Nov. 2005.

[8] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit, Aug. 2007.

[9] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, 2008.

[10] L. Andersen. Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, 1994.

[11] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Usenix Annual Technical Conference*, June 2000.

[12] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks. In *10th ACM Conference on Computer and Communication Security (CCS)*, Oct. 2003.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Usenix Security Symposium*, Aug. 2003.

[14] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[15] M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, 1996.

[16] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 11–11, Berkeley, CA, USA, 2006. USENIX Association.

[17] K. Chen and D. Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 75–84, New York, NY, USA, 2007. ACM Press.

[18] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, July 2005.

[19] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS*, Apr. 2001.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[21] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, Oct. 2007.

[22] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *SOSP*, Oct. 2005.

[23] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proceedings of the 10th USENIX Security Symposium*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[24] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, Aug. 2003.

[25] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Wadpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *USENIX Security Symposium*, Jan. 1998.

[26] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 162–171, New York, NY, USA, 2006. ACM Press.

[27] T. Durden. Bypassing PaX ASLR protection. *Phrack*, (59), July 2002.

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

[29] N. Heintze and O. Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *PLDI*, June 2001.

[30] M. Howard. ASLR features in Windows Vista. `http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx`, 2006.

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

[32] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the Third International Workshop on Automated Debugging*, pages 13–26, May 1997.

[33] jp. Advanced doug lea's malloc exploits. *Phrack*, (61), Sep. 2003.

[34] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM CCS*, Oct. 2003.

[35] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, Aug. 2002.

[36] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Aug. 2001.

[37] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, June 2007.

[38] L. Li, J. E. Just, and R. Sekar. Address-space randomization for Windows systems. In *ACSAC*, 2006.

[39] Microsoft. Phoenix compiler framework. `http://research.microsoft.com/phoenix/phoenixrdk.aspx`.

[40] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4), July 2003.

[41] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[42] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *NDSS*, Feb. 2005.

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

[44] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *NDSS*, Feb. 2004.

[45] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*, Oct. 2004.

[46] Skylined. Heap spraying. `http://www.edup.tudelft.nl/~bjwever/`, 2004.

[47] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *NDSS*, Feb. 2005.

[48] N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? The effectiveness of instruction set randomization. In *Usenix Security Symposium*, Aug. 2005.

[49] SPEC. Olden benchmark suite. `http://www.cs.princeton.edu/~mcc/olden.html`.

[50] SPEC. Spec Benchmarks. `http://www.spec.org`.

[51] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

[52] O. Whitehouse. An analysis of address space randomization on Windows Vista. *Symantec Advanced Threat Research*, March 2007.

[53] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Network and Distributed System Security Symposium (NDSS)*, pages 149–162, San Diego, California, February 2003.

[54] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *SRDS*, Oct. 2003.

[55] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.

[56] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *SIGSOFT Softw. Eng. Notes*, 29(6):117–126, 2004.

[57] S. H. Yong and S. Horwitz. Pointer-range analysis. In *Proceedings of Static Analysis Symposium (SAS)*, 2004.