Data Space Randomization[⋆]

Sandeep Bhatkar[1] and R. Sekar[2]

[1] Symantec Research Laboratories, Culver City, CA 90230, USA
[2] Stony Brook University, Stony Brook, NY 11794, USA

Abstract. Over the past several years, US-CERT advisories, as well as most critical updates from software vendors, have been due to memory corruption vulnerabilities such as buffer overflows, heap overflows, etc. Several techniques have been developed to defend against the exploitation of these vulnerabilities, with the most promising defenses being based on randomization. Two randomization techniques have been explored so far: address space randomization (ASR) that randomizes the location of objects in virtual memory, and instruction set randomization (ISR) that randomizes the representation of code. We explore a third form of randomization called data space randomization (DSR) that randomizes the representation of data stored in program memory. Unlike ISR, DSR is effective against non-control data attacks as well as code injection attacks. Unlike ASR, it can protect against corruption of non-pointer data as well as pointer-valued data. Moreover, DSR provides a much higher range of randomization (typically 2^{32} for 32-bit data) as compared to ASR. Other interesting aspects of DSR include (a) it does not share a weakness common to randomization-based defenses, namely, susceptibility to information leakage attacks, and (b) it is capable of detecting some exploits that are missed by full bounds-checking techniques, e.g., some of the overflows from one field of a structure to the next field. Our implementation results show that with appropriate design choices, DSR can achieve a performance overhead in the range of 5% to 30% for a range of programs.

Keywords: memory error, buffer overflow, address space randomization

1 Introduction

Memory errors continue to be the principal culprit behind most security vulnerabilities. Most critical security updates from software vendors in the past several years have addressed memory corruption vulnerabilities in C and C++ programs. This factor has fueled a lot of research into defenses against exploitation of these vulnerabilities. Early research targeted specific exploit types such as stack-smashing, but attackers soon discovered alternative ways to exploit memory errors. Subsequently, randomization based defenses emerged as a more systematic solution against these attacks. So far, two main forms of randomization defenses have been explored: *address-space randomization (ASR)* [32,9] that randomizes the locations of data and code objects in memory, and *instruction set randomization (ISR)* [6,27] that randomizes the representation of code.

⋆ This research is supported in part by an ONR grant N000140710928 and an NSF grant CNS-0627687. This work was part of the first author's Ph.D. work [8] completed at Stony Brook University.

Although ASR and ISR have been quite effective in blocking most memory exploits that have been used in the past, new types of exploits continue to emerge that can evade them. As defenses such as ASR begin to get deployed, attackers seek out vulnerabilities and exploits that go beyond them. One class of attacks that can evade coarse-grained ASR is based on corrupting non-control data [13]. In particular, buffer overflows that corrupt non-pointer data are not captured by coarse-grained ASR. Moreover, ASR implementations that are deployed today suffer from the problem of low entropy. This enables brute-force attacks that succeed relatively quickly — with about 128 attempts in the case of Windows Vista, and 32K attempts in the case of PaX [32,35]. Finally, ASR techniques are vulnerable to information leakage attacks that reveal pointer values in the victim program. This can happen due to a bug that sends the contents of an uninitialized buffer to an attacker — such data may contain pointer values that may have been previously stored in the buffer. We therefore develop an alternative approach for randomization, called *data space randomization (DSR)*, that addresses these drawbacks of previous randomization-based techniques.

The basic idea behind DSR is to *randomize the representation of different data objects.* One way to modify data representation is to xor each data object in memory with a unique random mask ("encryption"), and to unmask it before its use ("decryption"). DSR can be implemented using a program transformation that modifies each assignment $x = v$ in the program into $x = m_x \oplus v$, where m_x is a mask associated with the variable x. Similarly, an expression such as $x + y$ will have to be transformed into $(x \oplus m_x) + (y \oplus m_y)$.

To understand how DSR helps defeat memory corruption attacks, consider a buffer overflow attack involving an array variable a that overwrites an adjacent variable b with a value v. As a result of DSR, all values that are written into the variable a will use a mask m_a, and hence the value stored in the memory location corresponding to b would be $v \oplus m_a$. When b is subsequently used, its value will be unmasked using m_b and hence the result will be $(v \oplus m_a) \oplus m_b$, which is different from v as long as we ensure $m_a \neq m_b$. By using different masks for different variables, we can ensure that even if the attacker manages to overwrite b, all she would have accomplished is to write a random value into it, rather than being able to write the intended value v.

Although inspired by PointGuard [17], which proposed masking of all pointer values with a random value, our DSR technique differs from it in many ways.

– First, PointGuard is focused on preventing pointer corruption attacks — otherwise known as *absolute-address-dependent attacks.* In contrast, the primary goal of DSR is to prevent *relative address attacks,* such as those caused by buffer overflows and integer overflows. Consequently, DSR is able to detect non-control data attacks that don't involve pointer corruption, such as attacks that target file names, command names, userids, authentication data, etc. Moreover, since pointer corruption attacks rely on a preceding buffer overflow, absolute-address-dependent attacks are also defeated by DSR.
– Second, DSR randomizes the representation of *all types of data*, as opposed to PointGuard which randomizes only pointer-typed data. (Indeed, as a result of

optimizations, the representation of many pointer variables are left unchanged in DSR.) DSR uses different representations for different data objects in order to prevent buffer overflows on one object from corrupting a nearby object in a predictable way.

- Third, DSR corrects an important problem with PointGuard that can break legitimate C-programs in which pointer and non-pointer data are aliased. For instance, suppose that an integer-type variable is assigned a value of 0, and subsequently, the same location is accessed as a pointer-type. The zero value won't be interpreted as a null value since PointGuard would xor it with a mask m, thus yielding a pointer value m. We note that such aliasing is relatively common due to (a) unions that contain pointer and non-pointer data, (b) use of functions such as `bzero` or `bcopy`, as well as assignments involving structs, and (c) type casts. DSR considers aliasing and hence does not suffer from this drawback.
- Finally, like other previous randomization based defenses, PointGuard is susceptible to *information leakage* attacks that leak the values of encrypted pointers to a remote attacker. Since a simple xor mask is used, leakage of masked data allows the attacker to compute the mask used by PointGuard. She can then mount a successful attack where the appropriate bytes within the attack payload have been masked using this mask. In contrast, DSR is able to discover all instances where masked data is being accessed, and unmask it before use. As a result, an information leakage attack will not reveal masked values.

As compared to ASR, DSR provides a much larger range of randomization. For instance, on 32-bit architectures, we can randomize integers and pointers over a range of 2^{32} values, which is much larger than the range possible with ASR. Moreover, DSR can, in many instances, address the weakness of even the fine-grained ASR techniques [10] concerning their inability to randomize relative distances between certain data items, e.g., between the fields of a struct. Since the C-language definition fixes the distance between struct fields, even bounds-checking techniques do not provide protection from overflows across the fields of a structure. In contrast, DSR has the ability to protect from such overflows as long as there is no aliasing between these fields[3]. (However, this feature is not currently supported due to our use of field-insensitive alias analysis in our implementation.)

A direct implementation of DSR concept can lead to nontrivial runtime overheads due to the need for masking/unmasking after every memory access, and due to the additional memory overheads for accessing mask data. To provide better performance, observe that the first step in memory corruption attacks involve a buffer overflow, i.e., an operation that starts with the base address of an object a in memory, but then accesses a different object b as a result of out-of-bounds subscript or invalid pointer arithmetic operation. Our implementation focuses on disrupting this step. Note that this is possible even without

[3] Typically, aliasing of multiple fields is induced by low-level functions such as `bcopy` and `bzero`. DSR can use different masks for different fields of a struct object if the object is not involved in these operations. In some cases, it is possible to improve this further by incorporating the semantics of these block move operations into the DSR implementation.

masking b, as long as we ensure that a uses a non-zero mask. A static analysis can be used to identify *overflow candidates*, i.e., objects such as a that can serve as a base address in an address arithmetic computation that goes out-of-bounds. In its simplest form, this analysis would identify all arrays, structures containing arrays, and any other object whose address is explicitly taken in the code. This optimization provides significant benefits since most variables accessed in C-programs are simple local variables that can be determined to be non-overflow candidates.

One of the main limitations of the DSR approach is the need to use the same mask for overflow candidate objects that may be aliased. To mitigate the impact of this limitation, our implementation attempts to allocate different objects with the same mask in different memory regions that are separated by unmapped pages. This ensures that even when two objects are assigned the same mask, overflows from one of these objects to the other would be detected since it would cause a memory fault due to the protection memory page in between the objects. However, the number of overflow candidate objects with the same mask may become very large for heap-allocated objects, and hence this approach may not be appropriate for such objects. In those cases, our implementation essentially provides probabilistic protection against overflows involving such objects.

1.1 Paper Organization

In Section 2, we describe the transformations to introduce data space randomization. In Section 3, we describe a prototype implementation of our technique. In section 4, we evaluate performance overhead, and analyze the effectiveness of our technique against different attacks. Related work is covered in Section 5, followed by concluding remarks in Section 6.

2 Transformation Overview

Our transformation approach for DSR is based on a source-to-source transformation of C programs. The basic transformation is quite simple. For each data variable `v`, we introduce another variable `m_v` which stores the mask value to be used for randomizing the data stored in `v` using an exclusive-or operation. The mask is a random number that can be generated at the beginning of program execution for static variables, and at the time of memory allocation for stack and heap variables. The size of `m_v` depends on the size of the data stored in `v`. Ideally, we can store a fixed size (say, word length) random number in the mask variable, and depending on the size of the associated variable, we can generate bigger or smaller masks from the random number. However, for simplicity of notation, we will use mask variables having the same size as that of the variables being masked.

The variables appearing in expressions and statements are transformed as follows. Values assigned to variables are randomized. Thus, after every statement that assigns a value to a variable `v`, we add the statement `v = v ^ m_v` to randomize the value of the variable in the memory. Also, wherever a variable is used, its value is first derandomized. This is done by replacing `v` with `v ^ m_v`.

So far the transformations seem straightforward, but we have not yet considered a case in which variable data is accessed indirectly by dereferencing pointers, as in the following C-code snippet:

```
      int x, y, z, *ptr;
      ...
      ptr = &x;
      ...
      ptr = &y;
      ...
  L1: z = *ptr;
```

In the above code, the expression `*ptr` is an alias for either `x` or `y`. Since `*ptr` is used in the assignment statement at `L1`, we need to unmask it before using its value in the assignment. Therefore, the line should be transformed as:

```
   z = m_z ^ (m_starptr ^ *ptr),
```

where `m_z` and `m_starptr` are respectively masks of `z` and `*ptr`. Unfortunately, statically we cannot determine the mask `m_starptr` to be used for unmasking; it can be the mask of either variable `x` or `y`.

One way to address this problem is to dynamically track the masks to be used for *referents*[4] of all the pointers. This requires storing additional information (henceforth called metadata) about pointers. Similar information is maintained in some of the previous techniques that detect memory errors. In particular, they store metadata using different data structures such as *splay tree* [26] and *fat pointers* [4,30]. These metadata storing techniques lead to either high performance overheads or code compatibility problems. For this reason, we chose to avoid dynamic tracking of masks.

Our solution to the above problem is based on using static analysis[5]. More specifically, we use the same mask for variables that can be pointed by a common pointer. Thus, when the pointer is dereferenced, we know the mask to be used for its referents statically. This scheme requires "points-to" information which can be obtained by using *pointer analysis*, further described in Section 2.1. In the above example, from the results of any conservative pointer analysis technique, we can conclude that both variables `x` and `y` can be pointed by the pointer variable `ptr`. Hence we can use the same mask for both `x` and `y`, and this mask can be then used for unmasking `*ptr`, i.e., `m_x = m_y = m_starptr`. Mask assignment based on the results of pointer analysis is described in Section 2.2.

The principal weakness of the DSR approach arises due to potential aliasing. In particular, if two objects a and b can be aliased, then the same mask will be used for both, which means that overflows from a to b cannot be detected. To address this problem, we allocate objects that share the same mask in different

[4] A *referent* of a pointer is an object that the pointer points to.

[5] A static analysis typically assumes the absence of memory errors. Yet, in our work, we expect that memory errors will occur, and expect the technique to defend against them. In this regard, note that the effect of a memory error is to create additional aliases at runtime — for instance, if p was a pointer to an array a, due to a buffer overflow, it may also end up pointing to an adjacent object b. However, since the static analysis did not report this possible aliasing, we would have assigned different masks for a and b. As a result, the buffer overflow would corrupt b with values that will appear "random," when unmasked using m_b.

memory regions that are separated by an unmapped memory page. In this case, a typical buffer overflow from a to b will attempt to modify data in this inaccessible page, which causes a memory fault. We will revisit this solution in the discussion of optimization later in this section.

2.1 Pointer Analysis

Ideally, we would like to associate a distinct mask with each variable. Unfortunately, the use of pointers in C language potentially forces the assignment of the same mask for different variables. As a result, variables are divided into different equivalence classes. All the variables in a class are assigned the same mask, and those belonging to different classes are assigned different masks. The number of the equivalence classes depends on the precision of pointer analysis. Intuitively, greater the precision, there will be more number of the equivalence classes.

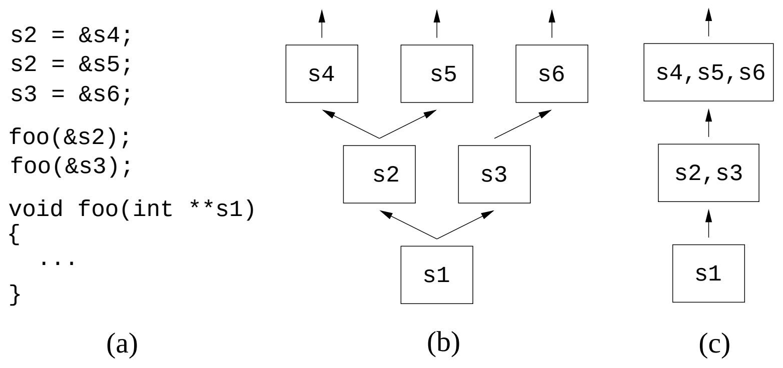
A pointer analysis is, in general, computationally undecidable [33]. As a result, existing pointer analysis algorithms use approximations that provide varying degree of precision and efficiency. The worst-case time complexities of these algorithms range from linear to exponential. We need to consider the time complexity for the analysis to be efficient and scalable. There are several factors that affect precision and efficiency of analysis. Such factors include flow-sensitivity, context-sensitivity, modeling of heap objects, modeling of aggregate objects, and representation of alias information [24]. We need to consider these factors while choosing the analysis.

Algorithms involved in existing flow-sensitive analyses [25] are very expensive in terms of time complexity (high order polynomials). Context-sensitive approaches [21,39] have exponential time complexity in the worst case. We avoid these two types of analyses as they do not scale to large programs. Among the flow-insensitive and context-insensitive algorithms, Andersen's algorithm [3] is considered to be the most precise algorithm. This algorithm has the worst case cubic time complexity, which is still high for it to be used on large programs. On the other hand, Steensgaard's algorithm [37] has linear time complexity, but it gives less precise results. Interestingly, as we shall show in the next section, it turns out that the results of Andersen's and Steensgaard's analyses give us the same equivalence classes of variable masks. Therefore, we implemented Steensgaard's algorithm for our purpose.

Steensgaard's algorithm performs flow-insensitive and context-insensitive inter-procedural points-to analysis that scales to large programs. It computes points-to set over named variables corresponding to local, global and heap objects. We use single logical object to represent all heap objects that are allocated at the same program point. We perform field-insensitive analysis, i.e., we do not distinguish between different fields in the same structure or union. Our implementation is similar to the one described in [37].

2.2 Mask Assignment

Consider points-to graphs computed by Steensgaard's and Andersen's algorithms as shown in Figure 1. A points-to graph captures points-to information in the form of a directed graph, where nodes represent equivalence classes of symbols


```
s2 = &s4;
s2 = &s5;
s3 = &s6;

foo(&s2);
foo(&s3);

void foo(int **s1)
{
  ...
}
```

Fig. 1. Figure (a) above shows a sample C program for which points-to graph is computed. Figures (b) and (c) show the points-to graphs computed by Andersen's algorithm and Steensgaard's algorithm respectively.

and edges represent pointer relationships. Points-to information computed by Andersen's algorithm is more precise than that computed by Steensgaard's algorithm. For instance, according to Steensgaard's graph, `s2` may point to `s6`. However, this relationship appears unlikely if we look at the program. Andersen's graph does not capture this relationship, hence it is more precise. In Steensgaard's analysis, two objects that are pointed by the same pointer are unioned into one node. This may lead to unioning of the points-to sets of formerly distinct objects. This kind of unioning makes the algorithm faster, but results in less precise output as shown in the above example.

Now let us see how we can use the points-to information to determine the equivalence classes of masks for the above example; we do this for Andersen's graph. Objects `s2` and `s3` can be accessed using the pointer dereference `*s1`. This suggests that `s2` and `s3` should have the same mask, and therefore they belong to the same equivalence class. Similarly, pointer dereference `**s1` can be used to access any of the objects pointed by `s2` or `s3`. This implies that the objects pointed by `s2` and `s3` should have the same mask, and hence objects `s4`, `s5` and `s6` should be merged into the same equivalence class. This merging is similar to the unioning operation in Steensgaard's algorithm. Therefore, the equivalence classes of masks will be the same even in the case of Steensgaard's graph. For the above example, the complete set of equivalence classes of masks is {{`s1`}, {`*s1`, `s2`, `s3`}, {`**s1`, `*s2`, `*s3`, `s4`, `s5`, `s6`}}. As Steensgaard's and Andersen's graphs are equivalent from the point of view of determining masks, we use Steensgaard's algorithm for our purpose as it is more efficient than Andersen's algorithm.

Now we formally define the procedure for determining masks using a Steensgaard's points-to graph (refer Figure 2). In general, a points-to graph of a program consists of disconnected components. Hence we consider the procedure only for one component which can be similarly applied to all the graph components. For this, let us first look at the properties of a Steensgaard's points-to graph. The unioning operation in Steensgaard's algorithm enforces following properties

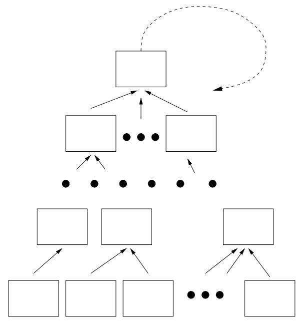


Fig. 2. A Steensgaard's point-to graph

in the points-to graph. A node in the graph has at most one outdegree and zero or more indegree. Owing to this, a connected component in the graph assumes a tree-like structure, where a node can have multiple children corresponding to the indegree edges, but at most one parent depending on the presence of an outdegree edge. However, this does not imply that the component is always a tree. There is a possibility that the root node of the tree-like structure may have an outward edge pointing to any of the nodes in the component, resulting in a cycle. Figure 2 shows such an edge as a dashed line.

We assign a distinct mask to each node of the points-to graph. Note that a node may correspond to multiple variables. The mask of the node is thus used for masking all of its variables.

The mask of an object that is accessed using a pointer dereference is determined as follows. Let `ptr` be the pointer variable. First, the node N corresponding to the pointer variable is located in the points-to graph. For the object `*ptr`, its mask is the mask associated with the parent node $parent(N)$. Similarly, the mask of `**ptr` is the mask associated with $parent(parent(N))$, and so on. Since each node has at most one parent, we can uniquely determine the masks of objects accessed through pointer dereferences. Note that this procedure also works for dereferences of a non-pointer variable that stores an address because the points-to graph captures the points-to relation involved. The procedure for dereferences of pointer expressions involving pointer arithmetic is similar.

Optimization

Indiscriminate introduction of masking/unmasking operations can degrade performance. For instance, many programs make use of a large number of variables that hold integer (or floating-point) values. If we can avoid masking/unmasking for such variables, significant performance gains are possible. At the same time, we want to ensure that this optimization does not have a significant impact on security. We show how this can be achieved by masking only the overflow candidate objects.

There are two types of memory corruption attacks: absolute address-dependent attacks and relative address-dependent attacks. Absolute address-dependent attacks involve corruption of a pointer value. However, mechanisms used for corrupting a pointer value, such as buffer overflows, heap overflows and integer overflows, are in fact relative address-dependent. So if we can defeat relative address-dependent attacks, we get automatic protection for absolute address-dependent attacks. Relative address-dependent attacks involve overflows from overflow candidate objects, and we make these attacks difficult as described below.

All non-overflow candidate objects are allocated in one memory region and we separate memory for this region from the overflow candidate objects with an unmapped memory page. As a result, overflows from overflow candidate objects into non-overflow candidate objects become impossible.

Overflows from an overflow candidate object into another overflow candidate object is possible. To address this problem, first we mask all the overflow candidate objects. Second, we identify objects that may be aliased, and allocate them in disjoint areas of memory that are separated by an unmapped memory page. Now, any attempt to overflow from one of these objects to the other will cause a memory exception, since such a write must also write into the intervening unmapped page[6]. The number of memory regions needed is equal to the number of different objects that use the same mask at runtime. This number can be statically estimated and is small for static data, and hence each such object can be allocated in a disjoint memory area. In typical programs, this number appears to be small for stack-allocated data, so we have been able to allocate such objects across a small number of disjoint stacks. (We call them buffer stacks.) Note that the strategy of removing overflow candidate objects from the main stack has some additional benefits: it removes the possibility of a stack-smashing attack — not only is the return address protected this way, but also other data such as saved registers and temporaries. This is important since, as a source-to-source transformation, we cannot ensure that saved registers and temporaries use a mask. Since this data cannot be corrupted by buffer overflows, we mitigate this weakness. If the number of stack objects with the same mask is large, we can move the objects into the heap. Protection of these objects will then be the same as that of heap objects.

For the heap, however, the number of distinct objects with the same mask may be large, thereby making it difficult to allocate those objects from different memory regions. As a result, our approach is to use a fixed number of memory regions, and cycle the heap allocations through these regions as successive objects with the same masks are allocated. This approach increases the likelihood of successful buffer overflows across two heap blocks, but note that traditional heap overflows, which are based on corrupting metadata stored at the beginning (or end) of heap blocks will fail: an attempt to overwrite metadata value with `x`

[6] The details of this step are somewhat complicated by our choice of implementing this technique using a source-to-source transformation, as opposed to modifying a compiler. With this choice, we cannot control how the memory for objects is allocated. We therefore borrowed a technique from [10] which uses an extra level of indirection for accessing objects. Intuitively, this technique can be viewed as a means to control the layout of objects.

will instead end up writing `x^m_h`, where `m_h` denotes the mask associated with the heap block.

The technique of separating aliased objects with a guard page prevents the common form of buffer overflows, which involve writing a contiguous (or closely spaced) set of locations beyond the end of a buffer. However, there can be buffer overflows that allow an attacker to corrupt memory that is far from the base of the buffer. Such overflows are common in conjunction with integer overflows. The guard-page technique does not protect against this attack. We therefore rely on our relative-address randomization technique [10] as a second line of defense against such attacks.

The technique described so far contains a vulnerability that occurs due to reuse of storage for non-overflow candidate objects. For instance, such a vulnerability may arise in a program that uses an uninitialized pointer variable for which we do not assign any mask. Now, if an attacker can control the previous use of the memory corresponding to this pointer, she can potentially corrupt the pointer with a chosen value. We address this vulnerability by ensuring that all objects are initialized before use, which is any way necessary to prevent information leakage attacks. An information leakage attack targets randomization techniques by exploiting a vulnerability that leaks the random values. An uninitialized non-overflow candidate variable may hold a masked value of a previously used data, and if this value is leaked, it is possible for an attacker to derive the mask. The attacker can then target other variables that share the derived mask. Note that an overflow candidate object is not vulnerable to information leakage attacks because any attempt to read this object will cause the mask associated with the object to be applied. In other words, the attacker receives the plaintext data rather than the masked data.

3 Implementation

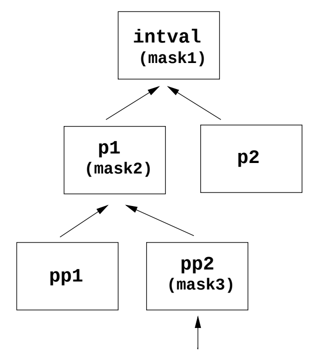
Our transformation approach is applicable to C programs. We use CIL [29] as the front end, and Objective Caml as the implementation language. We describe our implementation approach for a 32-bit x86 architecture and Linux operating system.

As a first step in the transformation of a program, we first perform pointer analysis in order to determine masks associated with different data. Our current implementation supports Steensgaard's pointer analysis. One of the limitation of our current implementation is that it is based on whole program analysis and transformation. The whole program analysis approach requires a merged source file. The CIL toolkit provides an option to automatically generate such a merged file. Sometimes this kind of merging can fail due to type mismatch of variable declarations present in different files. Such cases can be handled by manual changes to the declarations. With some extra effort, our implementation can be extended to a separate file compilation-based transformation approach. Even with the current implementation approach, we have demonstrated its practicality by transforming several "real-world" programs without any manual changes.

```
int *p1, *p2, **pp1, **pp2, intval;
...
int main()
{
   ...
   p1 = &intval;
   pp1 = &p1;

   pp2 = pp1;
   p2 = *pp2;
   ...
   ... = &pp2;
   ...
}
```

(a) A sample C code



(b) Points-to graph for the code

```
static unsigned int mask1, mask2, mask3;
int **p1_ptr, *p2, **pp1, ***pp2_ptr, *intval_ptr, ...;
int main()
{  ...
   (*p1_ptr) = intval_ptr;
   (*p1_ptr) = (int *)((unsigned int)(*p1_ptr) ^ mask2);
   pp1 = p1_ptr;
   (*pp2_ptr) = pp1;
   (*pp2_ptr) = (int **)((unsigned int)(*pp2_ptr) ^ mask3);
   p2 = (int *)((unsigned int)(*((int **)
        ((unsigned int)(*pp2_ptr) ^ mask3))) ^ mask2);
   ...
}
static void (__attribute__((__constructor__)) __dsr_init)()
{  ...
   /* code to allocate memory for intval, p1 and pp2 using their
   pointers intval_ptr, p1_ptr and pp2_ptr respectively. */
   ...
   __dsr_maskassign(mask1); __dsr_maskassign(mask2);
   __dsr_maskassign(mask3);
}
```

(c) Transformed code for the code in (a)

Fig. 3. A sample example illustrating basic DSR transformations.

In the second step, we generate the program's points-to graph, from which we then compute the equivalence classes needed for assigning random masks to data variables. In the third step, we transform the code as per the transformations described in the previous section.

The example shown in Figure 3 illustrates the above transformation steps. In this example, variables `p2` and `pp1` correspond to non-overflow candidate objects, which are not required to be masked due to our optimization. On the other hand, variables `intval`, `p1` and `pp2` correspond to overflow candidate objects because their addresses are taken. So we mask these variables, and for this we respectively introduce variables `mask1`, `mask2`, and `mask3` to store their masks. Each of these mask variables is initialized with a different random value using the macro `__dsr_maskassign` in the constructor function `__dsr_init()` that is automatically invoked before the start of the execution in `main()`. Recall from Section 2 that we need to allocate memory for overflow candidate objects in different memory regions. For this to be possible, we access overflow candidate objects with an extra level of indirection using pointers, e.g., a variable `v` is accessed using `(*v_ptr)`, where `v_ptr` is a pointer to `v`. In this example, we introduce pointers `intval_ptr`, `p1_ptr`, and `pp2_ptr` to access `intval`, `p1`, and `pp2` respectively. The memory for these overflow candidate objects is allocated in the initialization code present in `__dsr_init()`. Since the overflow candidate objects in this example do not share masks, we allocate their memory in the same region, in between two unmapped memory pages. As a result, overflows from overflow candidate objects cannot corrupt non-overflow candidate objects. Moreover, overflows among overflow candidate objects are detected because all of them use different masks.

The statements are transformed as follows. If an overflow candidate variable is assigned a value, the value is first masked and then stored in the memory; if it is used in an expression, its masked value is unmasked before its use.

Now we discuss a few issues concerning the basic implementation approach.

3.1 Handling Overflows Within Structures

According to C language specifications, overflows within structures are not considered as memory errors. However, attackers can potentially exploit such overflows also. For instance, an overflow from an array field inside a structure corrupting adjacent fields in the same structure may lead to an exploitable vulnerability. Thus, it is desirable to have some protection from these overflows. Unfortunately, even bounds-checking detection techniques do not provide defense against these types of overflows. ASR too fails to address this problem due to the inherent limitation of not being able to randomize relative distances between fields of a structure because of language semantics. DSR can be used to provide some level of protection in this case. The basic idea is to use field-sensitive points-to analysis so that we can assign different masks to different fields of the same structure. However, our current implementation does not support field-sensitive points-to analysis. As a part of future enhancement, we plan to implement Steensgaard's points-to analysis [36] to handle field-sensitivity. The time complexity of this analysis, as reported in [36], is likely to be close to linear in the size of the program in practice. Hence, this enhancement would not affect the scalability of our approach. Moreover, it does not increase runtime performance overhead.

Library functions such as `memcpy`, `memset` and `bzero`, which operate on entire structures, need a special transformation. For instance, we cannot allow `bzero` to zero out all the fields of a structure. Instead it should assign each field a value corresponding to its mask. This would require computing points-to set for pointer arguments of these functions in a context-sensitive way (as if the functions are inlined at their invocation point). As a result, the pointer arguments would most likely point to specific type of data including structures and arrays. So if the data pointed by an argument is a structure, we would use corresponding masks for the individual fields of the structures using summarization functions.

3.2 Handling Variable Argument Functions

Variable argument functions need special handling. In effect, we treat them as if they take an array (with some maximum size limit) as a parameter. This effectively means that the same mask is assigned to all the parameters, and if some of these parameters happen to be pointers, then all their targets get assigned the same mask, and so on. However, the imprecision in resulting pointer analysis can be addressed by analyzing such functions in a context-sensitive manner. Our implementation currently does not support this.

3.3 Transformation of Libraries

A source-code based approach such as ours requires the source code for the program as well as the libraries, as all of them need the transformation.

A few extra steps are required for handling shared libraries. Using the steps described in Section 2, we would obtain points-to graphs for all the shared libraries and the main executable. Since these graphs could be partial, we need to compute the global points-to graph. This could potentially lead to merging of some equivalence classes of masks, which in turn can make an object in a shared library an alias of another object from the executable or other shared libraries. In such situation, mask values are needed to be shared across the executable and the libraries.

A natural way to implement the above steps is to enhance the functionality of the dynamic linker. For this, each binary object (an executable or a shared library) needs to maintain dynamic symbol information about the points-to graph, which is a general yet an important piece of information that could be useful to many program analysis and transformation techniques. In addition, the binary objects need storage for mask variables and also dynamic symbol information about them. Using this information, at link-time, the dynamic linker can compute the global points-to graph, and resolve the mask variables just like it resolves other dynamic symbols. Additionally, it needs to initialize the mask variables with random values.

At times, source code may not be available for some libraries. Such libraries cannot be directly used with our DSR technique. The standard approach for dealing with this problem is to rely on summarization functions that capture the effect of such external library functions.

Given the prototype nature of our implementation, we did not transform shared libraries, and instead used the approach of summarization functions. For the test programs used in our experiments, we needed to provide summarizations for 52 `glibc` functions. In addition, we do not mask external variables, (i.e., shared library variables) and any internal variable that gets aliased with an external variable, so as to make our technique work with untransformed libraries.

4 Evaluation

4.1 Functionality

We have implemented DSR technique as described in the previous section. The implementation is robust enough to handle several "real-world" programs shown in Figure 4. We verified that these programs worked correctly after the transformation. We also manually inspected the source code to ensure that the masking and unmasking operations were performed on data accesses, and that variables were grouped into regions guarded by unmapped pages as described earlier.

4.2 Runtime Overheads

Figure 4 shows the runtime overheads, when the original and the transformed programs were compiled using `gcc-3.2.2` with optimization flag `-O2`, and run on a desktop running RedHat Linux 9.0 with 1.7 GHz Pentium IV processor and 512 MB RAM. Execution times were averaged over 10 runs.

Program	Workload	% Overhead
`patch-1.06`	Apply a 2 MB patch-file on a 9 MB file	4
`tar-1.13.25`	Create a tar file of a directory of size 141 MB	5
`grep-2.5.1`	Search a pattern in files of combined size 108 MB	7
`ctags-5.6`	Generate a tag file for a 17511-line C source code	11
`gzip-1.1.3`	Compress a 12 MB file	24
`bc-1.06`	Find factorial of 600	27
`bison-1.35`	Parse C++ grammar file	28
	Average	15

Fig. 4. Runtime performance overhead introduced by transformations for DSR.

For DSR transformations, the runtime overhead depends mainly on memory accesses that result in masking and unmasking operations. In I/O-intensive programs, such as `tar` and `patch`, most of the execution time is spent in I/O operations, and hence we see low overheads for such programs. On the other hand, CPU-intensive programs are likely to spend substantial part of the execution time in performing memory accesses. That is why we observe higher overheads for CPU-intensive programs. The average overhead is around 15%, which is a bit higher than the overheads for ASR techniques. Nonetheless, DSR technique is still practical and provides a stronger level of protection.

4.3 Analysis of Effectiveness Against Different Attacks

Effectiveness can be evaluated experimentally or analytically. Experimental evaluation involves running a set of well-known exploits against vulnerable programs, and showing that our transformation stops these exploits. Instead, we have relied on an analytical evaluation for the following reasons. First, exploits are usually very fragile, and any small modification to the code, even if it they are not designed for attack protection, will cause the exploit to fail. Clearly, with our implementation, which moves objects around, the attacks would fail even if we used a zero-valued mask in all cases. Modifying the exploit so that it works in this base case is quite time-consuming, so we did not attempt this. Instead, we rely on an analytical evaluation that argues why certain classes of existing exploitation techniques will fail against DSR; and estimate the success probability of other attacks.

Stack buffer overflows. Memory for all the overflow candidate local variables is allocated in a buffer stack or the heap. Typical buffer overflow attacks on the stack target the data on the main stack, such as the return address and the saved base pointer. These attacks will fail deterministically, since the buffer and the target are in different memory regions, with guard pages in between. Similarly, all attacks that attempt to corrupt non-aggregate local data, such as integer or floating-point valued local variables, saved registers, and temporaries will also fail.

Attacks that corrupt data stored in an overflow candidate variable by overflowing another overflow candidate variable is possible if they are both in the same memory region. However, such attacks have low probability of success (2^{-32}) because we ensure that objects with the same mask are allocated in different memory region. As mentioned before, in our implementation, we could do this without having to maintain a large number of buffer stacks. If this assumption did not hold for some programs, then we could resort to moving those overflow candidate variables to the heap, and falling back on the technique (and analysis of effectiveness) as overflows in the heap.

Static buffer overflows. Static overflow candidate objects are separated from static non-overflow candidate objects with inaccessible pages. So overflows in static memory cannot be used for corrupting non-overflow candidate static data.

Overflows from a static overflow candidate object into another overflow candidate object (not necessarily a static object) that is allocated in a different memory region are impossible due to the use of guard pages in between regions. However, overflows within the same static data region are possible. For such overflows, since our implementation ensures that the masks for different variables within each region will be different, the probability of a successful data corruption attack is reduced to 2^{-32}. In our experiments, we needed less than 150 distinct memory regions in the static area.

Heap overflows. Heap overflows involve overwriting heap control data consisting of two pointer-values appearing at the end of the target heap block (or at the beginning of a subsequent block). This sort of overwrite is possible, but since our technique would be using a mask for the contents of the heap block (which, as pointed out earlier, is an overflow candidate data), whereas the heap-control related pointer values would be in cleartext. As a result, the corruption has only 2^{-32} probability of succeeding.

Overflows from one heap block to the next are possible. If the two heap blocks are masked with different masks, then the attack success probability is 2^{-32}. However, heap objects tend be large in numbers, and moreover, possible aliasing may force us to assign the same mask to a large number of them. An important point to be noted in this case is that the number of different memory regions required for heap objects is a property of input to the program, rather than the program itself. Hence we use a probabilistic approach, and distribute heap objects randomly over a bounded number of different memory regions. Moreover, it should be noted that inter-heap-block overflows corrupt the heap control data in between, so the program may crash (due to the use of this corrupted data) before the corrupted data is used by the program. Nevertheless, it is clear that the success probability can be larger than 2^{-32}. In practice, this is hard because (a) heap allocations tend to be unpredictable as they are function of previous computations performed and inputs processed, (b) the control data will also be corrupted, and so it will likely be detected.

Format string attacks. Traditional format-string attacks make use the `%n` directive to write the number of bytes printed so far, and require the attacker to be able to specify the location for this write. Note that the attacker has control only over the format string, which being an overflow candidate object does not reside on the main stack. The location for the write is going to be a parameter, which will reside on the main stack. Thus the attacker cannot control the target into which the write will be performed, thus defeating traditional format-string attacks.

Other ways of exploiting format string attacks may be possible. The attacker may use `%n` to refer to a pointer value or an attacker-controlled integer value that is already on the stack. Also, the attacker can use other format specifiers such as `%x and %d` to print out the contents of the stack.

We point out that this weakness is shared with most existing defenses against memory error exploits. Nonetheless, the best way to deal with format-string attacks is to combine DSR with an efficient technique such as FormatGuard[16].

Relative address attacks based on integer overflows. Note that the base address used in any relative address attack must correspond to an overflow candidate variable. There is a small probability that the target location overwritten by the attack will have been assigned the same mask as the one corresponding to the base address. It is hard to predict this probability independent of the program, as it is the same as the probability of possible aliasing between the base address and the target address of the attack. In any case, relative-address randomization of overflow candidate objects in DSR provides probabilistic protection against these attacks.

Attacks Targeting DSR. We discuss possible attacks targeted at DSR.

- **Information leakage attack.** Randomization based defenses are usually vulnerable to information leakage attacks that leak the random values. For instance, ASR is vulnerable to attack that leaks the values of pointers that are stored on the stack or the heap. Interestingly, DSR is not susceptible to this attack. This is because any attempt to read a pointer value will automatically cause the mask associated with the pointer to be applied, i.e., the result of the read will be in plaintext rather than being in encrypted form. Thus, information regarding the masks is not leaked.
- **Brute force and guessing attacks.** The probability calculations in the previous sections indicate the difficulty of these attacks.
- **Partial pointer overwrites.** These attacks involve corruption of the lower byte(s) of a pointer. Partial pointer overflows can decrease the attacker's work

because there are only 256 possibilities for the LS byte. But these vulnerabilities are difficult to find and exploit. Even when exploitable, the target usually must be on the stack. In our implementation, since the main stack does not contain overflow candidate variables, it becomes impossible to use buffer overflows to effect partial overflow attack on stack-resident data.

5 Related Work

Runtime Guarding. These techniques transform a program to prevent corruption of specific targets such as return addresses, function pointers, or other control data. Techniques such as StackGuard [18], RAD [15], StackShield [5], Libverify [5] and Libsafe [5], in one way or another, prevent undetected corruption of the return address on the stack. ProPolice [22] additionally guards against corruption of non-aggregate local data. FormatGuard [16] transforms source code to provide protection from format-string attacks.

As above techniques provide only an attack-specific protection, attackers find it very easy to discover other attack mechanisms for bypassing the protection.

Runtime Enforcement of Static Analysis Results. Static analysis based intrusion detection techniques such as [38] were based on using a static analysis to compute program control-flow, and enforcing this at runtime. However, since enforcement was performed only on system call invocations, control-flow hijack attacks were still possible. Control-flow integrity (CFI) [1] addressed this weakness by monitoring all control-flow transfers, and ensuring that they were to locations predicted by a static analysis. As a result, control-flow hijack attacks are detected, but the technique does not detect data corruption attacks. Data-flow integrity [12] addresses this weakness by enforcing statically analyzed *def-use* relationships at runtime. However, it incurs much higher performance overheads than CFI as well as DSR.

Write-integrity testing (WIT) [2] proposes a faster way to perform runtime checking of validity of memory updates. Specifically, they use a static analysis to identify all memory locations that can be written by an instruction, and assign the same "color" to all these locations. This color is encoded into the program text as a constant. At runtime, a global array is used to record the color associated with each memory location. Before a memory write, WIT ensures that the color associated with the write instruction matches the color of the location that is written. Although developed independent of our work [8], the techniques behind WIT share some similarities with DSR. In particular, their color assignment algorithm is also based on alias analysis.

WIT reports lower overheads than DSR, but this is achieved by checking only the write operations. In contrast, DSR addresses reads as well as writes, and hence can address memory corruption attacks that may be based on out-of-bounds reads. In terms of strength of protection, DSR and WIT are comparable in the context of buffer overflows where the source and target objects are *not* aliased. If they are aliased, then WIT can still offer deterministic protection against typical buffer overflows that involve writing a contiguous (or closely spaced) set of locations beyond the end of a buffer. DSR can also provide deterministic protection in such cases, but its implementation technique for achieving this, namely, the use of unwritable memory pages, does not scale well for heap objects. However, WIT fails to provide any protection in the case of buffer overflows where the source and target objects are aliased and are far apart — this happens often in the case of integer overflows. In contrast, DSR offers probabilistic protection in this case due to its use of relative address randomization as a second line of defense.

Runtime Bounds and Pointer Checking. Several techniques have been developed that maintain, at runtime, metadata related to memory allocations and pointers [4,26,30,34,41,20]. Pointer dereferences are then checked for validity against this metadata. While the techniques differ in terms of the range of memory errors that they are able to detect, they all cover a broad enough range to protect against most memory error exploits. As compared to the techniques described in previous paragraph, these techniques tend to be more precise since they rely on runtime techniques (rather than static analysis) for metadata-tracking. However, this also translates to significant additional overheads.

Randomization Techniques. Our DSR technique is an instance of the broader idea of introducing diversity in nonfunctional aspects of software, an idea first suggested by Forrest et al [23]. The basic idea is that the diversified programs maintain the same functionality, but differ in their processing of erroneous inputs. As a result, different variants exhibit different behaviors when exposed to the same exploit. In the context of memory error vulnerabilities, several recent works have demonstrated the usefulness of introducing automated diversity in the low level implementation details, such as memory layout [32,40,23,28], system calls [14], and instruction sets [27,6]. System call and instruction set randomization techniques only protect against injected code attacks, but not from return-to-libc (aka existing code) or data corruption attacks. On the other hand, the techniques which randomize memory layout, popularly known as address space randomization (ASR) techniques, provide protection against injected code as well as data corruption attacks. ASR techniques that only perform absolute address randomization [32,9] (AAR) don't directly address buffer overflows, but are still able to defeat most attacks as they rely on pointer corruption. ASR techniques that augment AAR with relative address randomization (RAR) [10] are effective against all buffer overflows, including those not involving pointer corruption. DieHard [7] and DieFast [31] approaches provide randomization-based defense against memory corruption attacks involving heap objects.

Randomization techniques with relatively small range of randomization, e.g., PaX with its 16-bits of randomness in some memory regions, can be defeated relatively quickly using guessing attacks [35]. As mentioned earlier, they are also susceptible to information leakage attacks. Cox et al. [19] and Bruschi et al. [11] have shown how process replication can be used to address these deficiencies, and hence provide deterministic (rather than probabilistic) defense against certain classes of memory exploits. However, they come with a significant overhead due to the need to run two copies of the protected process. In contrast, DSR incurs only modest overheads to mitigate guessing attacks (using a much larger range of randomization) as well as information leakage attacks.

6 Conclusion

In this paper, we introduced a new randomization-based defense against memory error exploits. Unlike previous defenses such as instruction set randomization that was effective only against injected code attacks, our DSR technique can defend against emerging attacks that target security-critical data. Unlike address space randomization, which is primarily effective against pointer corruption attacks, DSR provides a high degree of protection from attacks that corrupt non-pointer data. Moreover, it is not vulnerable to information leakage attacks. Finally, it provides much higher entropy than existing ASR implementations, thus providing an effective defense from brute-force attacks.

We described the design and implementation of DSR in this paper. Our results show that the technique has relatively low overheads. In addition to reducing the likelihood of successful attacks to 2^{-32} in most cases, the technique also provides deterministic protection against attacks such as stack-smashing, and traditional format-string attacks. In future work, we expect to address some of the limitations of current prototype, such as the inability to address intra-field overflows.

References

1. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity - principles, implementations, and applications. In *ACM conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.
2. P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *IEEE Symposium on Security and Privacy*, May 2008.
3. L. O. Andersen. Program analysis and specialization for the C programming language. PhD Thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.
4. T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, June 1994.
5. A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, pages 251–262, Berkeley, CA, June 2000.
6. E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM conference on Computer and Communications Security (CCS)*, Washington, DC, October 2003.
7. E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, Ottawa, Canada, June 2006.
8. S. Bhatkar. Defeating memory error exploits using automated software diversity. Ph.D. Thesis, Stony Brook University, September 2007. Available at http://seclab.cs.sunysb.edu/seclab/pubs/thesis/sandeep.pdf.
9. 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*, August 2003.
10. S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, Baltimore, MD, August 2005.
11. D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified process replicae for defeating memory error exploits. In *International Workshop on Information Assurance (WIA)*, April 2007.
12. M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 2006.
13. S. Chen, J. Xu, and E. C. Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security Symposium*, 2005.

14. M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.
15. T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *IEEE International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001.
16. C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from `printf` format string vulnerabilities. In *USENIX Security Symposium*, 2001.
17. C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2003.
18. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.
19. B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, August 2006.
20. D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *International Conference on Software Engineering*, 2006.
21. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
22. H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.
23. S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.
24. M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.
25. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, July 1999.
26. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *International Workshop on Automated and Algorithmic Debugging*, pages 13–26, 1997.
27. G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 272–280, Washington, DC, October 2003.
28. L. Li, J. Just, and R. Sekar. Address-space randomization for windows systems. In *Annual Computer Security Applications Conference (ACSAC)*, December 2006.
29. S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, 2002.
30. G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2002.
31. G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, San Diego, CA, June 2007.
32. PaX. Published on World-Wide Web at URL http://pax.grsecurity.net, 2001.
33. G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, September 1994.
34. O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*, pages 159–169, San Diego, CA, February 2004.
35. H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 298–307, Washington, DC, October 2004.
36. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Conference on Compiler Construction*, LNCS 1060, pages 136–150, April 1996.
37. B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 32–41, January 1996.
38. D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, May 2001.
39. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.
40. J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 2003.
41. W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Newport Beach, CA, November 2004.