Effective Memory Protection Using Dynamic Tainting

James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic
College of Computing
Georgia Institute of Technology
{clause|idoud|orso|milos}@cc.gatech.edu

ABSTRACT

Programs written in languages that provide direct access to memory through pointers often contain memory-related faults, which may cause non-deterministic failures and even security vulnerabilities. In this paper, we present a new technique based on dynamic tainting for protecting programs from illegal memory accesses. When memory is allocated, at runtime, our technique taints both the memory and the corresponding pointer using the same taint mark. Taint marks are then suitably propagated while the program executes and are checked every time a memory address m is accessed through a pointer p; if the taint marks associated with m and p differ, the execution is stopped and the illegal access is reported. To allow for a low-overhead, hardware-assisted implementation of the approach, we make several key technical and engineering decisions in the definition of our technique. In particular, we use a configurable, low number of reusable taint marks instead of a unique mark for each area of memory allocated, which reduces the overhead of the approach without limiting its flexibility and ability to target most memory-related faults and attacks known to date. We also define the technique at the binary level, which lets us handle the (very) common case of applications that use third-party libraries whose source code is unavailable. To investigate the effectiveness and practicality of our approach, we implemented it for heap-allocated memory and performed a preliminary empirical study on a set of programs. Our results show that (1) our technique can identify a large class of memory-related faults, even when using only two unique taint marks, and (2) a hardware-assisted implementation of the technique could achieve overhead in the single digits.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging; C.0 [General]: Hardware/Software Interfaces;

General Terms: Performance, Security

Keywords: Illegal memory accesses, dynamic tainting, hardware support

1. INTRODUCTION

Memory-related faults are a serious problem for languages that allow direct memory access through pointers. An important class of memory-related faults are what we call illegal memory accesses. In languages such as C and C++, when memory allocation is requested, a currently-free area of memory m of the specified size is reserved. After m has been allocated, its initial address can be assigned to a pointer p, either immediately (*e.g.,* in the case of heap allocated memory) or at a later time (*e.g.,* when retrieving and storing the address of a local variable). From that point on, the only legal accesses to m through a pointer are accesses performed through p or through other pointers derived from p. (In Section 3, we clearly define what it means to derive a pointer from another pointer.) All other accesses to m are *Illegal Memory Accesses* (*IMA*s), that is, accesses where a pointer is used to access memory outside the bounds of the memory area with which it was originally associated.

IMAs are especially relevant for several reasons. First, they are caused by typical programming errors, such as array-out-of-bounds accesses and NULL pointer dereferences, and are thus widespread and common. Second, they often result in non-deterministic failures that are hard to identify and diagnose; the specific effects of an IMA depend on several factors, such as memory layout, that may vary between executions. Finally, many security concerns such as viruses, worms, and rootkits use IMAs as their injection vectors.

In this paper, we present a new dynamic technique for protecting programs against IMAs that is effective against most known types of illegal accesses. The basic idea behind the technique is to use dynamic tainting (or dynamic information flow) [8] to keep track of which memory areas can be accessed through which pointers, as follows. At runtime, our technique taints both allocated memory and pointers using taint marks. Dynamic taint propagation, together with a suitable handling of memory-allocation and deallocation operations, ensures that taint marks are appropriately propagated during execution. Every time the program accesses some memory through a pointer, our technique checks whether the access is legal by comparing the taint mark associated with the memory and the taint mark associated with the pointer used to access it. If the marks match, the access is considered legitimate. Otherwise, the execution is stopped and an IMA is reported.

In defining our approach, our final goal is the development of a low-overhead, hardware-assisted tool that is practical and can be used on deployed software. A *hardware-assisted tool* is a tool that leverages the benefits of both hardware and software. Typically, some performance critical aspects are moved to the hardware to achieve maximum efficiency, while software is used to perform operations that would be too complex to implement in hardware.

There are two main characteristics of our approach that were defined to help achieve our goal of a hardware-assisted implementation. The *first characteristic* is that our technique only uses a small, configurable number of reusable taint marks instead of a unique mark for each area of memory allocated. Using a low number of

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

taint marks allows for representing taint marks using only a few bits, which can considerably reduce the amount of memory needed to store taint information, enable a hardware-assisted implementation of the approach, and ultimately reduce both space and time overhead. Although using too few taint marks could result in false negatives for some types of failures or attacks, the vast majority of memory-related faults and attacks known to date would still be detected. Moreover, the approach is flexible enough that it can handle more problematic cases as needed by trading efficiency with effectiveness.

The *second characteristic* is that the technique is defined at the binary, rather than at the source, level. The ability to handle binaries is necessary for IMA detection; all non-trivial applications use external libraries or components whose source code is usually not available. Because memory and pointers can be heavily manipulated by such external code, an approach that does not handle these libraries would typically produce unsound results. Working at the binary level also facilitates a hardware-assisted implementation, which operates at the machine level where there is little or no knowledge of the source code structure.

To evaluate our approach, we developed a software-only prototype that implements the approach for x86 binaries and heap-allocated memory and used it to perform a set of empirical studies. To implement the prototype, we used a generic dynamic-taint analysis framework that we developed in previous work [2]. We also implemented a second prototype using a MIPS-based architecture hardware simulator. This second implementation allowed us to assess the time overhead that would be imposed by a hardware-assisted implementation of the approach.

In the evaluation, we applied our technique to a set of programs that we gathered from different sources and investigated several aspects of our technique. More precisely, we used (1) subjects with known memory-related faults to investigate the effectiveness of our technique in identifying such faults and (2) subjects with no known memory-related faults to investigate the overhead imposed by the technique (and as a sanity check). Overall, the results of the evaluation are encouraging. First, they show that our technique can identify a large class of memory-related faults, even when using only two taint marks. Second, the results provide initial evidence that a hardware-assisted implementation of the technique could achieve time overhead in the single digits, which would make it practical for use on deployed software.

The contributions of this paper are:

- A novel technique for detecting IMAs that is effective and could be efficiently implemented by leveraging hardware support.
- Two prototype implementations of the technique for heap-allocated memory, a software-based one that works on x86 binaries and a hardware-assisted one that works on MIPS binaries.
- A set of empirical studies that provide evidence of the effectiveness and practical applicability of the approach.

2. MOTIVATING EXAMPLE

In this section, we provide a motivating example that we will use in the rest of the paper to illustrate our technique. The code shown in Figure 1 is taken from an on-line C and C++ reference manual [3] and consists of a function that, given an integer n, generates and prints a string of $n - 1$ random characters. We slightly modified the original code by adding the use of a seed for the random number generation, adding a call to a function (`getSeedFromUser`) that reads the seed from the user and returns it in a parameter passed by address. We also introduced a memory-related fault: at line 7, we changed the terminating condition for the `for` loop from “`i < n`” to “`i <= n`”. With the current condition, in the last iteration of the loop, the statement at line 8 writes a random character at position `buffer + n`. Because the address at offset `n` is outside the bounds of the memory area pointed by `buffer`, accessing it through pointer `buffer` represents an IMA.

```
    void prRandStr(int n) {
 1.   int i, seed;
 2.   char *buffer;

 3.   buffer = (char *) malloc(n);
 4.   if (buffer == NULL) return;

 5.   getSeedFromUser(&seed);
 6.   srand(seed);

 7.   for(i = 0; i <= n; i++)        /* fault */
 8.     buffer[i] = rand()%26+'a'; /* IMA */
 9.   buffer[n - 1] = '\0';

10.   printf("Random string: %s\n", buffer);
11.   free(buffer);
    }
```

Figure 1: An example of IMA.

Faults of this kind are fairly common and are a good representative of memory-related faults that can lead to IMAs. As we discussed in the Introduction, one of the problems with IMAs is that they are often non-deterministic, which is also true for our example. Whether the write to address `buffer + n` would result in a failure or not depends on the state of the memory at that address (allocated or free) and on how that memory is used in the rest of the execution. If the memory is not allocated, then the function will appear to behave correctly. Even if the memory is allocated, as long as the value at that address is not used for any further computation in the program (*e.g.,* if the next operation on it is a write), no failure will occur. Conversely, if the value is used, the random value written in it may cause a failure.

3. OUR TECHNIQUE

In this section we present our technique for identifying IMAs. Although the technique is meant to operate on binary code, describing it at that level involves many low-level technical details that would be distracting and would unnecessarily complicate the presentation of the approach. Therefore, for the sake of clarity, we first introduce our technique by describing how it works at the source-code level. For the same reason, we also avoid discussing some specific cases, such as `static` local variables, that our technique handles analogously to the cases we discuss. Finally, in the general discussion of the approach, we assume to have an unlimited number of taint marks available. After presenting our general approach at the source-code level, we then discuss how the approach works when the number of taint marks is limited (Section 3.2) and finally present low-level details of our x86-based implementation (Section 4).

3.1 General Approach

As we discussed in the Introduction, our technique is based on dynamic tainting, which consists of marking and tracking certain data in a program at runtime. In our tainting approach, we instrument a program to mark two kinds of data: memory in the data space and pointers. When a memory area m in the data space is allocated, the technique *taints* the memory area with a mark t. When a pointer p is created with m as its target (*i.e.,* p points to m's initial address), p is tainted with the same taint mark used to taint m. The technique also *propagates* taint marks associated with pointers as the program executes and pointers are used and manipulated.

Finally, when a memory area is accessed though a pointer, the technique *checks* that the memory area and the pointer have the same taint mark. It should be noted that pointers can be stored in memory, so a memory location in our technique actually has storage for two taint marks—one for the taint mark associated with the memory location itself, and the other for pointer taint associated with the value stored in that location.

In rest of this section, we describe in detail the three parts of our technique: tainting, taint propagation, and taint checking.

3.1.1 Tainting

This part of our technique is responsible for initializing taint marks for memory and pointers.

Tainting Memory. Memory can be allocated in two main ways: statically and dynamically.

Static memory allocations occur implicitly, as a consequence of a global or local variable being declared. The memory corresponding to global variables is allocated at program entry, by reserving space in a global data area. For local variables declared in a function f, memory is allocated upon to f, by reserving space on the stack to hold the variable's value. For our example code in Figure 1, assuming a 32 bit word size, 12 bytes of stack space are allocated to store local variables `i`, `seed`, and `buffer` when `prRandStr` is entered.

To taint statically-allocated memory, our technique intercepts function-entry and program-entry events, identifies the memory area used to store each local variable, and taints each individual area with a fresh taint mark. The memory area for a variable can be identified using its starting address and the size needed to store that variable's type. For an integer variable `i`, for instance, the memory area is the range $[\&i, \&i + sizeof(int))$. For statically-allocated arrays, the range is calculated analogously, with the exception that the type's size is multiplied by the number of elements in the array.

To illustrate the technique's handling of static allocations, consider again our example code. When function `prRandStr` is entered, the technique creates three new taint marks, t_1, t_2, and t_3. Mark t_1 is used to mark each of the four bytes in $[\&i, \&i + sizeof(int))$. Analogously, marks t_2 and t_3 are used to mark every byte of `seed` and `buffer`, respectively.

Dynamic memory allocations, unlike their static counterparts, occur explicitly, as a consequence of a call to a memory-allocation function. In C and C++, there is only a small set of memory-allocation functions, and they all fulfill two requirements: they (1) take as input the size of the memory area to allocate and (2) return either the initial address of a contiguous memory area of the requested size or `NULL` if the allocation is unsuccessful.

Therefore, to taint dynamically-allocated memory, our technique intercepts all calls to low-level memory-allocation functions, such as `malloc`. When such a function returns successfully, the technique first identifies the memory area that was allocated as the range $[r, r + size)$, where r is the value returned by the memory-allocation function and $size$ is the amount of memory requested passed as a parameter to the function. Then, it creates a fresh taint mark and uses it to taint the allocated memory. Finally, the technique stores the initial and final address of the tainted memory area (*i.e.,* r and $r + size$) to be able to correctly handle memory deallocations (as discussed in Section 3.1.2).

To illustrate, consider the call to *malloc* at line 3 in our example of Figure 1. Our technique would intercept the call and, if the call is successful, would mark every byte in the range $[buffer, buffer + n)$ with a fresh taint mark.

Tainting Pointers. Pointers can be initialized in several ways, depending on the type of memory they point to.

Pointers to dynamically-allocated memory are initialized, directly or indirectly, using the return value of the allocation function call used to allocate that memory area. Therefore, our technique taints this kind of pointer by, as before, intercepting calls to memory-allocation functions that return successfully. At that point, after tainting the allocated memory area, the technique also taints, using the same mark used for the memory, the pointer p to which the return value of the call is assigned. If other pointers were to be derived from p, p's taint mark would also propagate to those pointers, as we discuss in Section 3.1.2.

For an example of this type of pointer initialization, consider again the call to `malloc` at line 3 of our example. Because the return value of the call to `malloc` is assigned to variable `buffer`, our technique would taint pointer `buffer` with the same taint mark used to taint the dynamically-allocated memory area.

Pointers to statically-allocated memory can be initialized in two ways, depending on whether that memory contains a scalar value or an array. A pointer to a scalar value, such as an integer, can only be initialized using the *address-of* operator (`&`), which returns the starting memory address of the variable to which it is applied. When the *address-of* operator is used on a variable, our technique checks whether the area of memory that is being referenced is tainted and, if so, assigns the same taint mark to the pointer that stores the return value of the *address-of* operator.

For our example code, when the *address-of* operator at line 5 is used to get the address of `seed`, our technique retrieves the taint mark associated with the memory that stores `seed` and associates it with `getSeedFromUser`'s parameter.

The situation is slightly different for pointers to static-allocated arrays. The name of a static-allocated array is for all practical purposes a pointer to the first element of the array. For example, if an array is created as "`int a[10]`", `a` can be used for accessing the array's i^{th} elements using pointer arithmetic, as `*(a + i)`, and for retrieving the initial memory address of the array, as `<pointer to int> = a`. From a logical standpoint, when a statically-allocated array is created (at function or program entry) our technique first associates a taint marking to the memory used to store the array, as described above, and then assigns the same taint mark to the pointer corresponding to the array name. The practical way in which this is done is described in Section 4.

3.1.2 Taint Propagation

One of the key challenges in defining our technique is the definition of a propagation policy for the taint marks. In dynamic tainting, a propagation policy dictates how taint marks flow along data- and control-dependencies as the program executes. In our context, there are no cases where taint marks should propagate through control-flow, so we can disregard control dependences and define our propagation policy for data-flow only.

Our propagation policy treats taint marks associated with memory and taint marks associated with pointers differently. We describe how the policy works in the two cases separately.

Propagation of Memory Taint Marks. Taint marks associated with memory are not actually propagated. They are associated with a memory area when it is allocated and cleared when that memory is deallocated. Therefore, the propagation policy for memory taint marks is relatively straightforward: when an area of memory is being deallocated, our technique must identify the boundaries of the memory area and clear the taint marks associated with that memory area. It is worth noting that, when memory is deallocated and our technique clears that memory's taint marks, the corresponding pointer marks are not cleared. Pointers that are tainted with the same marks as the memory being released remain tainted (unless of course the pointer variable is overwritten with `NULL` or some other value). The fact that a pointer has a taint mark for which there is no corresponding memory area is consistent with the fact that it is a dangling pointer.

According to the above discussion, to suitably handle memory taint marks, our technique must be able to (1) intercept memory deallocations and (2) correctly identify the range of memory being deallocated. We explain how our technique can do this for the two types of allocated memory in a program: statically- and dynamically-allocated.

Dynamically allocated memory remains allocated until it is explicitly deallocated through a direct or indirect call to a memory-deallocation function (*i.e.,* `free`), which takes as a parameter the initial address of the memory to be released. To identify memory deallocations, our technique (1) intercepts calls to `free` from the code or from a library, (2) uses the memory address passed as a parameter to the call to retrieve the previously stored range of the corresponding memory area (see Section 3.1.1), and (3) clears taint marks for all the bytes in that memory area.

For our example in Figure 1, the technique would intercept the call to `free` at line 11 and clear the taint marks for the region $[buffer, buffer + n)$.

Statically allocated memory is deallocated when the function that allocated it returns (for local variables) or at program exit (for variables allocated in the global data area). The latter case is clearly irrelevant for our analysis. To handle deallocation of local variables, our technique intercepts function exits and clears all taint marks associated with the memory corresponding to the exiting function's stack. Because our technique can simply clear the stack at once, for local variables there is no need to store any memory range when they are initially tainted.

In our example code, when `prRandStr` returns, our technique clears taint marks associated with `prRandStr`'s stack, so removing taint marks associated with the memory that stores local variables `i` ($[\&i, \&i{+}sizeof(int))$), `seed` ($[\&seed, \&seed{+}sizeof(int))$), and `buffer` ($[\&buffer, \&buffer + sizeof(char*))$).

Propagation of Pointer Taint Marks. Unlike memory taint marks, taint marks associated with pointers are not just created and cleared. They actually propagate as data flows through the program during execution, which makes the propagation policy for pointer taint marks considerably more complex than the one for memory taint marks. To correctly account for pointer-taint propagation, our technique must suitably handle all pointer arithmetic operations, such as assignments, additions, subtractions, and even bitwise operations. In other words, the propagation policy for pointer taint marks must accurately model all possible operations on a pointer and associate to each such operation a taint-propagation action that assigns to the result of the operation a taint mark determined from the taint marks of the operation's operands.

For some operations, the corresponding taint-propagation action is trivial. For instance, an assignment operation "`p = q`" between two pointers can be easily handled by assigning any taint mark associated with pointer q to pointer p. Unfortunately, not all pointer operations are as simple as assignments. For a simple example, consider line 8 in Figure 1. The expression `buffer[n]` is actually using pointer arithmetic to calculate the memory location of the n^{th} element of `buffer`, that is, it is adding an integer (`n`) to a pointer (`buffer`) and dereferencing the pointer to access a memory location. An accurate taint-propagation policy must make sure that the result of "`buffer + n`" is tainted with the same mark as the original pointer `buffer`.

A superficial analysis of typical pointer arithmetic operations could produce a reasonable, initial propagation policy that accounts for common operations used with their common meaning. For example, additions or subtractions of a pointer p and an integer should produce a pointer with the same taint mark as p; subtractions of two pointers should produce an untainted integer (an offset); operations such as adding or multiplying two pointers or performing logical operations between pointers should be meaningless and simply result in an untainted value. Unfortunately, due to commonly used hand-coded assembly functions and compiler optimizations, a simple propagation policy defined in this way would be highly inaccurate and result in a large number of false negatives.

Even in our preliminary investigation, we encountered dozens of cases where a simple policy would fall short. We report a specific example to provide the reader with some sense of why simple policies are inadequate. The example involves the `strcpy` function of the C library. This is a commonly-used function that copies the contents of a character array (src) to another character array ($dest$) under the assumptions that the two arrays do not overlap. In the version of the C library that we inspected, the `strcpy` function is implemented as follows. It first initializes two pointers s and d that point to the initial address of src and $dest$, respectively. It then calculates the distance $dist$ between s and d by subtracting the two pointers. Finally, it executes a loop that reads the character at position s, copies it to the memory location $s + dist$, and loops, incrementing s by one, until the character copied is the string-termination character.

Using a simple policy like the one described above, this function would always produce false positives. Such an approach would taint the memory areas storing src and $dest$ with taint marks t_{src} and t_{dest}, respectively. When s and d are initialized to point to src and $dest$, our technique would correctly propagate t_{src} to s and t_{dest} to d. Offset $dist$ would be an untainted integer that, when added to s, would result in a pointer tainted with mark t_{src}. An access to any element of $dest$, which has taint t_{dest}, through a pointer resulting from s + $dist$, with taint t_{src}, would result in an IMA being incorrectly reported.

To address this and other issues that we found in our preliminary investigation, we defined a more sophisticated policy based on a series of factors: intuition, knowledge of C, C++, and the underlying machine language, and patterns found in the software subjects that we studied. We present a summary of our pointer-taint propagation policy by discussing how it handles different pointer operations.

Addition and Subtraction ($c = a$ +/- b)**.** For these two operations, we consider different cases based on the taint marks of a and b. If either a or b (but not both) is tainted with a given taint mark t, then the result c is also tainted with mark t. This case accounts for situations where a purely-numeric offset (*e.g.,* an array index) is added to (or subtracted from) a pointer.
If both a and b are tainted with two given marks t_a and t_b, respectively, then c is tainted with taint mark $t_a + t_b$, in the case of an addition, or $t_a - t_b$, for a subtraction. This approach lets us handle a range of situations that may occur in real programs. In particular, it accounts for cases where an offset is added to (resp., subtracted from) a pointer, and the offset was computed by subtracting (resp., adding) two pointers. To illustrate, consider the `strcpy` implementation discussed above. Using this policy, $dist$ would be tainted with mark $t_{dest} - t_{src}$. When $dist$ is later added to pointer s, which has taint t_{src}, the result would be tainted with taint mark $t_{src} + (t_{dest} - t_{src}) = t_{dest}$. The resulting pointer could therefore be used to access the elements of

array $dest$ without our technique generating any false positives. Finally, if neither a nor b are tainted, then c is also not tainted. In this case, the two values are either not used as pointers or they have not been initialized.

In our implementation, “untainted” means that the taint mark is zero, so all of these cases can be implemented by simply adding (subtracting) the operands’ taints whenever data values are added (subtracted) in the program.

Multiplication, Division, and Modulo. Independently from the taint mark of the operands, the result of any multiplication, division, or modulo operation is never tainted.

Bitwise AND **($c = a\&b$).** Handling the bitwise AND operator is problematic because, from the standpoint of pointer-taint propagation, its use can have different meanings depending on the value of its operands. In particular, there are cases where a program may compute an AND between a pointer and an integer to get a base address by masking the lowest bit of an address. For example, the instruction “`c = a & 0xffffff00`” would mask the lowest eight bits of pointer `a`. Unfortunately, the same approach could be used to compute an offset by executing, for instance, the instruction “`c = a & 0x000000000f`”.

To address this issue, we defined our propagation policy as follows. If a and b are either both untainted or both tainted, then c is not tainted. We could not identify any reasonable case where c could still contain useful pointer-related information in these two cases.

If only one of a and b is tainted with a given taint mark t, conversely, we taint c with taint mark t if it points to an address in the same memory area as the tainted operand. The rationale is that the effect of the AND operation is in this case, by definition, to obtain a pointer to a base address from a pointer at a given offset. If the number of taint marks is limited, however, c may point to a different memory area that happens to be tainted with the same mark as the memory area that the tainted operand points to. (We discuss the use of a limited number of operands in detail in Section 3.2.) Therefore, in this case, we apply a heuristic after performing the first check successfully. The heuristic consists of tainting c with taint mark t if the non-tainted operand is a mask that results in preserving the 16 most significant bits of the tainted operand (*i.e.,* the mask is a number that, in binary format, consists of at least 16 ones followed by no more than 16 zeros).

Bitwise NOT (c = $\sim$ a). A bitwise NOT can be used as an alternative way to subtract two values in some special cases. For example, an operation such as “`c = b - a - 1`”, which could be used to compute an offset between two pointers, could be implemented more efficiently as “`c = b + ∼ a`”.[1] To handle this type of situation, if a is tainted with a given taint mark t, our policy taints c with mark $-t$. This approach is consistent with the way our policy handles additions and subtractions that we described above.

Bitwise OR, and XOR. Because we could not identify any reasonable scenario where applying one of these operators to one or two pointers could produce a result that contains useful pointer-related information, these operators produce untainted values in our policy.

It is important to note that any of the above operators could be used in very creative ways and, thus, programmers’ inventiveness (or ingenuity) could result in instruction sequences that are not handled correctly by a specific policy. Therefore, it is unlikely that a propagation policy can be proven to be sound and complete. As far as our policy is concerned, as stated above, we defined it based on domain knowledge and on experience, and we verified that it works correctly for all the software that we studied so far, as discussed in Section 5. If our planned additional experimentation would reveal shortcomings of our policy, we will refine it accordingly.

3.1.3 Checking

This third and last part of the technique is responsible for checking that memory accesses are legal and reporting an IMA otherwise. The way in which our technique performs this check is straightforward. The technique intercepts any memory access, whether it is a read or a write, performed by the application (including libraries) through a pointer. If the memory accessed and the pointer used to access it are tainted with the same taint mark, then the access is considered legitimate. Conversely, if the taint marks are different, the technique considers the access an IMA and reports it. Note that the marks are also considered to be different in cases where either the pointer or the memory is not tainted but the other is. These cases may correspond to (and help identify) some typical types of memory-related failures, such as dereferencing an uninitialized pointer or a dangling pointer.

Considering again the code example in Figure 1, our technique would perform a checking operation when the statements at lines 5, 8, 9, 10, and 11 are executed. These are all locations where memory is accessed through a pointer (directly or through a method call). Other memory accesses, such as the ones at line 7, are not checked because they do not occur through pointers.

Currently, our technique is defined so that it halts program execution when it detects an IMA. However, the technique could also be used to perform different actions upon detection, such as attaching a debugger to the execution or just logging the IMA and allowing the execution to continue. The specific action chosen may depend on the context (*e.g.,* in-house versus in-the-field or friendly versus antagonistic environments).

3.2 Limiting the Number of Taint Marks

Ideally, our technique would use an unlimited number of unique taint marks, which would allow for detecting all IMAs. Realistically, however, the number of distinct taint marks that the technique can use must be limited for the technique to be practical.

Taint marks in our approach are represented as sets of n bits, which limits the number of distinct taint marks to 2^n. Although it is possible to use a large set of bits for taint marks (*e.g.,* 64 bits), which would make the set of unique taint marks virtually unlimited, the storage and manipulation of such large numbers of taint marks would introduce unacceptable overheads. As stated previously, our approach stores two taint marks for every memory location—one mark for the memory-taint, and the other for the pointer-taint of the value stored in that location. Using two 64-bit taint marks per byte of data would result in a 16-fold increase in memory occupation, which is prohibitive for many applications.

Most importantly, the use of a such a large number of taint marks would prevent one of our key goals—a practical hardware-assisted implementation of the approach. There are two primary reasons why a large number of taint marks are incompatible with an hardware-assisted implementation. First, the performance overhead in a hardware-based implementation comes mostly from the competition between data and taint marks for space (*e.g.,* in caches) and bandwidth (*e.g.,* on the system bus). Therefore, it is highly desirable that the number of bits needed to store the taint marks is small relative to the size of the corresponding data. Second, using a large set of bits for taint marks would dramatically affect the design com-

[1] We actually encountered this kind of optimization in real code.

plexity of the hardware. In fact, an ideal solution for a hardware-assisted implementation would use a single taint mark, thus requiring a *single bit* for storage. Solutions that involve a slightly larger number of taint marks, such as 2, 4, or possibly even 8 may still be viable, but they may result in too much additional complexity in the hardware design to be considered for actual implementation.

Based on these considerations, we decided to limit the number of taint marks in our approach (and implementation) to a small, configurable number. The drawback of this approach is that taint marks have to be reused and thus, in some cases, several allocated memory areas may happen to have the same taint mark. In these cases, if a pointer that was intended to point to one of these areas were used to access a different region with the same mark, the resulting IMA would remain undetected. In other words, when using a limited number of taint marks, IMAs are not detected with certainty, but rather *probabilistically*. Assuming that we have 2^n different taint markings, a uniformly-distributed random assignment of taint markings to memory regions, and IMAs such that a pointer accesses a random memory region, the probability of detecting each IMA would be equal to $P = 1 - \frac{1}{2^n}$. This is encouraging: even for a single taint mark, our approach may still find 50% of all IMAs at runtime.

Moreover, this estimate may actually be a lower bound for the effectiveness of our technique, for two reasons. First, using smarter-than-random (re)assignment strategies could reduce the probability of reassigning the same marks to two regions that could be accessed through each other's pointers. For example, in our technique we at least ensure that the same taint mark is never assigned to contiguous regions (when possible). Second, in cases where single faults can result in multiple IMAs, the 50% probability of detection for each IMA may correspond to an even higher probability of detecting the underlying defect(s). According to the above formula, with 2 bits, 4 bits, and 8 bits of storage for taint markings we would expect to detect each IMA with 75%, 94%, and 99.6% probability, respectively.

As we discuss in Section 5, in our empirical evaluation we investigated the effectiveness of our technique when using only two unique taint marks and got encouraging results: we found that the actual IMA-detection results were even better than our expectations based on the analytical computation outlined above. The reason is that most of the common program defects and vulnerabilities result in IMAs where a pointer is used to access a memory region that is adjacent to the one the pointer is intended for, or where a dangling pointer is used to access an unallocated region. Because our approach does not assign the same taint mark to two contiguous regions, it can detect these types of common IMAs even when using only two distinct taint marks. In addition, different and possibly more sophisticated IMAs could still be detected with the estimated probability computed above ($P = 1 - \frac{1}{2^n}$).

We conclude this section by noting that limiting the number of taint marks does not lead to false positives because a pointer and its corresponding memory region would still have the same taint mark. This property gives us the ability to *tune* the number of taint marks used to achieve a desired tradeoff between likelihood of IMA detection and space and performance overhead, without worrying about introducing false positives that would need to be investigated and detract from the practical usability of the technique.

4. IMPLEMENTATION

Although in the previous sections we have kept the discussion at the source-code level, our technique is actually defined to operate (and be implemented) at the binary level. In this section, we discuss the main differences between operating at these two levels and provide some details of the two prototype tools that we developed to perform our empirical evaluation: a software-based implementation and a hardware-based implementation.

4.1 Operating at the Binary Level

Even if operating on binaries, most parts of the technique can be implemented exactly as discussed in Section 3. Memory and pointers can be tainted just as described there. Program entry, function entries, and calls to dynamic memory-allocation routines can be intercepted to taint memory accordingly. Function exits and calls to memory deallocation routines can be intercepted to clear memory taint marks. Assignments of the return value of allocation functions, uses of the *address-of* operator (*i.e.,* its binary equivalent—the load effective address instruction, `lea`), and assignments of the starting address of a static array can be intercepted to taint pointers. Finally, all propagation actions for pointer manipulation, such as arithmetic and bitwise operations, can be implemented by simply mapping the source-level operations to the corresponding hardware instructions.

There is, however, one main challenge when implementing our technique at the binary level, This challenge is related to tainting statically-allocated memory. When operating at the binary level, a great deal of information that is normally available at the source-code level is lost (*e.g.,* the memory ranges associated with local variables or symbol-table information). Without this information, our technique would be unable to individually taint each local variable. Although in many cases such missing information can still be gathered by leveraging debugging information and/or tools like `readelf` and `objdump`,[2] there are also cases where this information cannot be retrieved. This could happen, for instance, when operating on obfuscated or stripped binaries.

In these cases, our technique is still able to operate, albeit with reduced precision; instead of using a unique taint mark for each statically-allocated region of memory, it uses a single mark for the entire range of local variables, which is *always* available in a binary. By doing this, the technique loses the ability to detect IMAs where a pointer to a local variable is used to access a different local variable, but it can still detect IMAs where a pointer to a local variable is used to access a dynamically-allocated memory area or vice versa. In particular, this less precise approach would still be effective in detecting some stack-based attack types, such as stack smashing.[3]

At this stage of the research, our implementation goal is to develop a prototype that lets us investigate feasibility and usefulness of our approach. Moreover, to be able to estimate the performance of a hardware-assisted implementation, we also need to develop a second implementation of our technique, using a simulator. Therefore, instead of implementing fully-fledged tools, we developed prototypes that handle heap-allocated memory only. Note that not considering statically-allocated memory does not affect the effectiveness of the technique when used to detect IMAs targeted at memory allocated on the heap. Note also that IMAs involving the heap are common, so even a partial implementation allows us to perform a meaningful evaluation of the general approach. In practice, this means that our prototypes can detect IMAs resulting from overflows of heap-allocated buffers, integer overflows, and format string attacks that write to heap-allocated memory. The prototypes cannot detect IMAs resulting from writes to statically allocated memory, such as IMAs involving the global offset table, IMAs that overflow stack-allocated arrays, and format string attacks that

[2] `http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html`

[3] `http://insecure.org/stf/smashstack.html`

write to statically-allocated memory. Finally, even though our prototypes only detect heap-related IMAs, both implementations propagate and check taint marks for all instructions, so the performance overhead of our hardware-assisted prototype is likely to be similar to the overhead for a fully-fledged implementation.

4.2 Software-based Implementation

To create our software-based prototype, we leveraged a generic dynamic tainting framework, called DYTAN, that we developed in previous work [2]. To instantiate the framework, we developed custom tainting, propagation, and checking routines that implement our approach and plugged them into DYTAN. DYTAN is in turn built on top of the `Pin` dynamic-instrumentation framework [14], which can instrument binary applications on the fly. Using dynamic binary instrumentation allows our prototype to handle dynamically-loaded shared libraries, which would be difficult for an implementation that operates on source code (unless all shared libraries were recompiled). Although `Pin` allows our implementation to handle shared libraries, it cannot instrument the underlying Operating System (OS), which creates additional challenges for our implementation with respect to handling system calls and signals.

System calls allow an application to request a service from the OS, such as opening a file or checking resource limits. In some instances, system calls can modify the application's memory. The `read` system call, for instance, fills a provided buffer with data read from a file descriptor. Because `Pin` does not instrument system calls, we model their behavior in our prototype. Fortunately, system calls are typically simple to model, so handling them is more tedious than conceptually difficult.

Signals are a mechanism used by the OS to report exceptional situations, such as a division by zero, to an application. Applications can register custom functions to handle a particular type of signal; if an application registers a signal-handler function, the OS invokes the provided function instead of the default action when the corresponding signal occurs. Before invoking the function, however, the OS saves the current state of the execution, including the value of all registers, and creates a clean stack frame for the signal handler. Then, when the handler returns, the OS restores the saved state. All these operations occur within the OS, so our technique would not realize that registers values have changed, and may incorrectly propagate taint marks. Therefore, to handle signals correctly and avoid generating spurious IMAs, our implementation intercepts calls to and returns from signal handlers and suitably saves and restores the correct taint marks.

4.3 Hardware-based Implementation

To implement our hardware-assisted prototype, we used a cycle-accurate simulation approach, which is nearly ubiquitously used in computer architecture research. In particular, we extended SESC [17], an open-source simulator that models the complete execution of code through a modern processor and memory system. Our modifications to SESC extend the simulator to also model, in detail, the operation of the hardware structures needed to implement our technique.

The modifications to the simulator include: (1) extending the hardware registers to hold the taint markings associated with the value in a register and (2) extending the processor core (*e.g.,* arithmetic and floating point units) so that taint markings propagate along with their corresponding data. These modifications allow the hardware to perform, in parallel, both the data propagation operations (*e.g.,* `add`, `sub`, and `and`) and the corresponding taint-propagation actions. This ability to perform taint propagation in parallel with computation on data is one of the key reasons we expect a hardware-based implementation of our approach to have a very low performance overhead.

A naive approach for storing taint markings associated with memory locations would be to simply extend the memory by the number of bits needed to store these marks. For example, per-word single-bit taint markings would result in a two-bit taint added to each 32 bit memory word (one bit for memory taint and one bit for pointer taint). However, from the perspective of commercial adoption, this approach is infeasible. The cost of memory modules (*i.e.,* RAM) is dependent on high volume production. A non-standard (*e.g.,* 34-bit- or 136-bit-wide) module would cost much more than standard 32-bit or 128-bit-wide modules, and would require other expensive changes in the system. For example, such a module would not fit into a standard DIMM slots on a motherboard, so a new format for these slots would be needed.

Instead of assuming that these tainting-specific memory modules and associated changes will become common, our technique adopts a different approach that does not modify the system hardware, except for the changes to the processor core outlined above. To achieve this, we store the taint markings in a packed array, separate from their associated data. When the processor fetches data from memory, it uses a simple index calculation to compute the address of the corresponding taint, and issues a separate fetch request for the taint marks. The steps needed to write to memory are analogous. Despite the additional memory accesses required by this approach, the extensive caching used by modern processors, combined with the favorable ratio between word size and the number of bits needed to store taint marks, results in only a minor increase in the actual number of reads and writes that access main memory through the system bus.

In summary, our hardware-assisted prototype follows well established guidelines for investigating architecture modifications, specifically using a cycle-accurate simulator to model the proposed changes. In addition, we also limit the scope of our architectural modifications to only the processor core, which increases the feasibility of implementing our approach in actual hardware (not a simulator), given today's existing hardware components.

5. EMPIRICAL EVALUATION

The goal of our empirical evaluation is to assess effectiveness and efficiency of our technique. To this end, we used the two prototypes described in Section 4 on a set of real applications gathered from different sources and investigated three research questions:

RQ1: How effective is our technique at detecting IMAs that involve heap-allocated memory when using only a small number of taint marks?

RQ2: Does our technique erroneously report as IMAs any legitimate memory accesses?

RQ3: How much run-time overhead is a hardware-supported implementation of the technique likely to impose?

RQ1 and RQ2 are concerned with the rate of false negatives and false positives, respectively, generated by the technique. RQ3 is concerned with the efficiency of the technique in the case of a hardware-assisted implementation.

Section 5.1 presents the software applications that we used in the study. Sections 5.2, 5.3, and 5.4 present results and discussion for each of our three research questions.

5.1 Experimental Subjects

In our empirical studies, we used two sets of subjects. The first set consists of applications with known illegal heap-memory accesses. We selected both subjects used in related work and subjects selected by looking at online bug databases. More precisely,

Application	IMA location	Type	Detected
bc-1.06	more_arrays: 177	buffer overflow	✓
bc-1.06	lookup: 177	buffer overflow	✓
gnupg-1.4.4	parse_comment: 2095	integer overflow	✓
mutt-1.4.2.li	utf8_to_utf7: 199	buffer overflow	✓
php-5.2.0	string.c: 3152	integer overflow	✓
pine-4.44	rfc822_cat: 260	buffer overflow	✓
squid-2.3	ftpBuildTitleUrl: 1024	buffer overflow	✓

Table 1: Results for RQ1 (effectiveness).

we selected from BugBench [13] the two applications with heap-related IMAs: bc v1.06, an interactive calculator (≈14.4k LoC), and squid v2.3, a web proxy cache server (≈93.5k LoC). bc has two known IMAs and squid has one known IMA. Browsing online bug databases, we identified four additional subjects: pine v4.44, an email and news client (≈211.9k LoC), mutt v1.4.2.li, another email and news client (≈453.6k LoC), gnupg v1.2.2, an implementation of the OpenPGP standard (≈117.3k LoC), and version 5.2.0 of the php language (≈558.2k LoC). Pine, mutt, gnupg, and php all have one known IMA. We used these six subjects to investigate RQ1.

Our second set of subjects consists of the twelve applications from the integer component of SPEC CPU2000 [20]. These applications cover a wide range of possible program behaviors and range in size from ≈3.1k LoC, for 181.mcf, to ≈1312.2k LoC, for 176.gcc.[4] The SPEC benchmarks were created as a standardized set of applications to be used for performance assessment and are close-to-ideal subjects for us for two reasons. First, they are widely used and, thus, thoroughly tested (*i.e.,* we do not expect them to be faulty), so we can use them to address RQ2. Second, they are commonly used to evaluate hardware-based approaches, so they are also a good set of subjects for investigating RQ3.

5.2 RQ1

To address RQ1, we ran the four applications from our first set of subjects while protecting them with our software-based tool and configured our tool so that it used only two taint marks. For each of the considered IMAs, we reran the corresponding application, reproduced the IMA, and checked whether our tool detected it. Because there is a probabilistic component in our technique when working with a limited number of taint marks, we repeated the study five times and obtained consistent results. The results of the study are shown in Table 1. For each IMA, the table shows the *application* containing the IMA, the *IMA location* in the application, the *type* of the illegal access, and whether or not our prototype successfully *detected* the IMA.

As the table shows, our prototype was able to successfully detect all five illegal memory accesses. This result is encouraging because it indicates that even with only a very limited number of unique taint markings (in this case, two), our technique can detect real heap-based IMAs.

5.3 RQ2

To address RQ2, we performed a study similar to the one we performed for RQ1: we protected the applications in the SPEC benchmarks using our software-based tool, run each of them five times against their test-input workload, and checked that no IMA was reported. Because we consider the programs in the benchmark to be virtually bug-free, due to their widespread usage, reporting an IMA would correspond to a false positive. Note that, although our technique should not generate any false positive by construction, we have no formal proof of that. Therefore, this study served as a sanity check for both our technique and our implementation.

```
      boolean Domain_Exit(ft F,lt Z,zz *Status) {
        ...
1259.   fclose(MsgFilePtr);
        ...
        // expanded from macro STAT
1268.   return(Test = *Status == 0 ?
               True :
               PrintErr (F, Z, *Status));
      }

      boolean PrintErr(ft F,lt Z,zz Status) {
        ...
  71.   SendMsg (0, Line);
        ...
      }

      boolean SendMsg (int MsgLevel, char *MsgLine) {
        ...
 278.   if (MsgFilePtr != NULL) {
 279.     fprintf (MsgFilePtr, "%s", MsgLine);
        ...
      }
```

Figure 2: IMA in 255.vortex

Although we observed the expected outcome for eleven of the twelve applications (*i.e.,* no IMAs reported), the prototype reported an IMA for 255.vortex, when function fprintf is called from function Ut_SendMsg. After examining 255.vortex's source code, we discovered that the IMA reported was indeed the consequence of a memory-related fault in the code.

We illustrate the fault in Figure 2. Function Domain_Exit is used to clean up resources before the program exits. In particular, at line 1259, it closes a FILE* pointer called MsgFilePtr. When the expression at line 1268 evaluates to false, the program calls PrintErr, which in turn calls SendMsg. Function SendMsg then attempts to write to the previously closed MsgFilePtr and generates an IMA. The problem occurs because FILE* pointers point to a dynamically allocated area of memory that is allocated through a call to fopen, and deallocated by calling fclose. When 255.vortex is protected by our tool, MsgFilePtr and its associated area of memory are both tainted with the same mark. When fclose is called and frees the memory pointed by MsgFilePtr, that memory is untainted, but MsgFilePtr retains its mark. Therefore, when later on SendMsg attempts to write to MsgFilePtr, our tool intercepts the memory access, detects that the pointer is tainted while the memory is not, and reports an IMA.

After verifying that the problem reported was an actual IMA, we checked the documentation for the SPEC benchmarks and found that it was indeed a known fault in 255.vortex that was corrected in the subsequent release of the benchmarks, which further confirms the relevance of the fault. We also found that it was the only known memory-related fault in that release of the benchmarks.

Overall, the results for RQ2 are fairly positive. Not only our technique did not generate any false positives, but it was also able to detect the only memory-related fault in the whole set of subjects.

5.4 RQ3

For RQ3, we could not use the software-based implementation of our approach. First, we developed our prototype by focusing on functionality rather than efficiency. We used a generic tainting framework that already trades speed for flexibility and imposes approximately a 30x time overhead [2]. In addition, we implemented our tainting, propagation, and checking approach as external functions that are invoked by the framework for every memory access, which results in a considerable additional overhead. As a

[4] Detailed information about the SPEC CPU2000 applications is available at http://www.spec.org/cpu/CINT2000/.

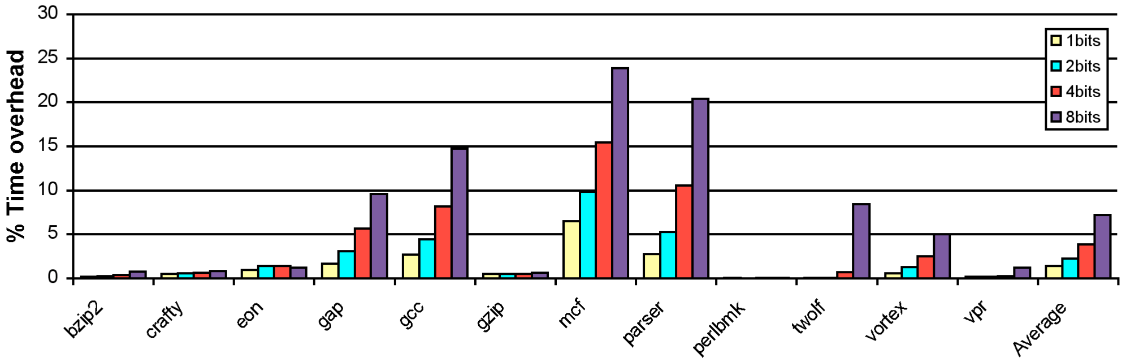

Figure 3: Performance overhead in a hardware-assisted implementation of our approach for different numbers of taint marks.

result, the overall overhead of the software-based implementation varies between 100x and 500x, depending on the application. Second, no software implementation can approach the efficiency of a hardware-assisted implementation due to the intrinsic cost of instrumenting almost every instruction in the code.

Therefore, to investigate this research question, we used the hardware-assisted implementation of our approach described in Section 4.3, which uses the SESC [17] cycle-accurate simulator to model a near-future processor and a memory system. More precisely, we model a 4-issue out-of-order 5GHz processor core with 16KB L1 caches, 2MB L2 caches, and 300-cycle memory latency. Our simulated processor uses the MIPS IV instruction set, which is considerably different from the x86 instruction set used in our software-based implementation. Therefore, our experiments actually evaluated our approach with two different types of binaries.

Similar to what we did for RQ2, we ran all of the SPEC benchmarks against their test-input workload and used the built-in functionality of the simulator to measure the overhead imposed by our technique on the benchmarks. For this study, we ran the technique for different possible numbers of taint markings: 2 (1 bit), 4 (2 bits), 16 (4 bits), and 256 (8 bits). In this way, we were able to get a more thorough assessment of the feasibility of the approach in cases where more than two taint marks could improve the effectiveness of the approach.

The results of the study are shown in Figure 3 using a block diagram. For each application and each number of taint marks considered, the figure shows on the Y-axis the time overhead expressed as a percentage of the base execution time for the application (*i.e.,* the execution time without our technique). The rightmost entry on the X-axis shows the average overhead results across all applications.

As the figure shows, the performance overhead increases as we increase the number of taint bits, as expected. The main reason for the increase is that taint marks and actual data are competing over cache and memory bandwidth. The figure also shows that the overhead varies across applications, which is due to different locality properties and different runtime behaviors. `186.crafty`, for instance, has a small working set that fits inside the data L1 and L2 cache, even after taint marks are added. Therefore, the overhead for `186.crafty` is extremely low. At the other end of the spectrum is `181.mcf`, which has high cache miss rates, exhibits very poor spatial locality in general (*i.e.,* when it accesses a memory location, it does not have a strong tendency to access nearby locations), and behaves even worse when taint marks are added. In this case, the high overhead is mainly caused by the frequent reloads of the cache memory. However, even under these highly unfavorable circumstances, the overhead imposed by our technique on `181.mcf` is lower than 25% (even with 8-bit taint markings).

Overall, the performance overhead results are also encouraging. On average, a hardware-supported implementation of our technique would impose about 1% performance overhead when only two taint marks are used and about 7% overhead when 256 taint marks are used. These numbers, if confirmed in further experimentation, would support the use of the technique on deployed software. They would also make a case for building the needed hardware support into processors, especially considering that many other applications using dynamic tainting are being investigated in various fields, including security and software testing.

5.5 Threats to Validity

The main threats to the external validity of our results are the limited number of applications considered and the focus on only one class of IMAs. Experiments with additional subjects containing other types of IMAs may generate different results. However, the applications we used in our studies are real(istic), representative, and widely used. Moreover, heap-related IMAs are a widespread and relevant class of IMAs [13]. Even if further evaluation were to show that our technique only works with this class of failures, the fact that our technique is able to detect them with high effectiveness and low performance overhead would be a good result in itself.

6. RELATED WORK

There is a large body of existing work, across many disciplines, that attempts to detect IMAs in C and C++ programs. In this section, we discuss the work most closely related to our research.

Program-analysis-based tools (*e.g.,* [7, 9, 11, 23]) attempt to discover IMAs by performing various types of static analysis on an application. Although powerful, these tools may produce a large number of false-positives due to the conservative nature of the analysis they perform, which is likely to alienate users. Language-based approaches, such as Cyclone [12] and CCured [15], are another form of static analysis that attempts to remove the possibility of IMAs by translating unsafe languages into safe variants. Some of these approaches attempt to perform an automated translation, but for large applications they still involve programmer intervention, in the form of rewriting or annotations. Overall, approaches based on static analysis can be considered complementary to dynamic approaches in terms of strengths and weaknesses.

Dynamic approaches for IMA detection instrument a target application to perform run-time monitoring. Instrumentation can either be done at the source-code level (*e.g.,* [6, 18, 24]) or at the binary level (*e.g.,* [10, 19]). Source-code level approaches typically impose less overhead because they can leverage additional information not present at the binary level. However, they have the problem of not being able to track memory allocations or accesses within external black-box libraries and components. Approaches based on dynamic instrumentation, conversely, can instrument code on the fly and handle pre-compiled external code. Among the approaches that work at the binary level, Valgrind [19] is the most similar to our technique, in that it uses a bit to keep track of which memory has been defined and identify illegal accesses to uninitialized memory. Unlike our technique, however, Valgrind cannot detect accesses to memory that has been initialized, but is being accessed through an illegal pointer.

Beside these software-based approaches, there have also been numerous proposals for hardware-assisted detection of IMAs. In particular, SafeMem [16] uses existing memory-error correcting codes to detect accesses to unallocated memory. Our approach is more general than SafeMem, in that we also detect illegal accesses to allocated memory. MemTracker [22] associates a *state* with each memory location and uses a programmable state machine to detect accesses incompatible with the location's current state (*e.g.,* reads from uninitialized locations). MemTracker targets a slightly different problem and cannot distinguish between accesses of the same kind from different pointers. For example, MemTracker can allow (or disallow) all reads from a location, but cannot prevent reads from a given pointer while allowing reads from another one.

There have also been several proposals (*e.g.,* [1, 4, 5, 21]) of techniques that use hardware support to taint data that comes from external inputs, propagate this taint at runtime, and detect when input-derived values are used as jump addresses or fetched as instructions. These mechanisms, as originally proposed, cannot support the taint propagation rules needed for our new IMA-detection technique. However, they demonstrate that hardware support can provide taint propagation with nearly negligible overheads.

7. CONCLUSION

This paper presents a novel dynamic technique for detecting Invalid Memory Accesses (IMAs). Our approach (1) taints a memory region and the pointers that are allowed to point to that region with the same taint mark, (2) propagates taint marks, and (3) checks memory accesses performed through pointers to make sure that the memory and the pointer used to access it have the same taint mark. If this is not the case, it reports an IMA and stops the execution.

Our approach has several key advantages over previously-proposed techniques for dynamic IMA detection. First, it is highly effective; in our empirical evaluation, it was able to identify all of the IMAs in the real programs that we considered. Second, it is amenable to a hardware-assisted implementation. Detailed simulations of a hardware-based implementation show performance overheads below 10%. Finally, unlike previous IMA detection techniques, our technique can easily be tuned to achieve different tradeoffs between performance overhead and probability of detecting IMAs.

In future work, we will extend the implementation of our technique to add support for statically-allocated variables. We will also perform additional empirical evaluation using a broader range of memory faults. Finally, we will investigate additional optimizations and design improvements that could further lower the performance overhead of our hardware-assisted implementation.

Acknowledgments

This work was supported in part by NSF award CCF-0541080 to Georgia Tech and by the Department of Homeland Security and US Air Force under Contract No. FA8750-05-2-0214. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the US Air Force.

8. REFERENCES

[1] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[2] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of The International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 196–206, London, UK, July 2007.

[3] cplusplus.com. Malloc example, June 2007. http://www.cplusplus.com/reference/clibrary/cstdlib/malloc.html.

[4] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[5] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible informatin flow architecture for software security. In *International Symposium on Computer Architecture*, 2007.

[6] 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.

[7] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. *SIGPLAN Not.*, 38(5):155–167, 2003.

[8] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[9] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. *SIGPLAN Not.*, 37(5):69–82, 2002.

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

[11] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. *SIGPLAN Not.*, 38(5):168–181, 2003.

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C, in *2002 USENIX Annual Technical Conference*, pages 275–288, 2002.
, Proceedings of the General Track: , June 10-15, 2002

[13] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 190–200, 2005.

[15] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[16] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[17] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[18] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, 2004.

[19] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[20] Standard Performance Evaluation Corporation. SPEC Benchmarks. *http://www.spec.org*, 2000.

[21] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[22] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, Vol., Iss., Feb. 2007*, pages 273–284, 2007.

[23] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Software Engineering Notes*, 28(5):327–336, 2003.

[24] 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.