Appears in IEEE Transactions on Computers (ToC), Pages 87-100, Vol. 61, Issue 1, January 2012

Effective and Efficient Memory Protection Using Dynamic Tainting

Ioannis Doudalis, *Student Member, IEEE,* James Clause, *Member, IEEE,*
Guru Venkataramani, *Member, IEEE,* Milos Prvulovic, *Senior Member, IEEE,*
and Alessandro Orso, *Member, IEEE,*

Abstract—Programs written in languages allowing direct access to memory through pointers often contain memory-related faults, which cause non-deterministic failures and security vulnerabilities. We present a new dynamic tainting technique to detect illegal memory accesses. When memory is allocated, at runtime, we taint both the memory and the corresponding pointer using the same taint mark. Taint marks are then propagated and checked every time a memory address m is accessed through a pointer p; if the associated taint marks differ, an illegal access is reported. To allow always-on checking using a low-overhead, hardware-assisted implementation, we make several key technical decisions. We use a configurable, low number of reusable taint marks instead of a unique mark for each allocated area of memory, reducing the performance overhead without losing the ability to target most memory-related faults. We also define the technique at the binary level, which helps handle applications using third-party libraries whose source code is unavailable. We created a software-only prototype of our technique and simulated a hardware-assisted implementation. Our results show that (1) it identifies a large class of memory-related faults, even when using only two unique taint marks, and (2) a hardware-assisted implementation can achieve performance overheads in single-digit percentages.

Index Terms—C Computer Systems Organization, C.0.b Hardware/-software interfaces, C.1 Processor Architectures, D.2.5.g Monitors

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 is allocated, a currently-free area of memory m of the required size is reserved. After m is allocated, its starting 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). An access to m is legal only if 1) it uses pointer p or a pointer derived from p and 2) if the access occurs during the interval when p is valid, (i.e. between the allocation and deallocation of m). All other accesses to m are *illegal memory accesses* (IMAs), where a pointer is used to access memory outside the bounds of the memory area with which it was originally associated, or outside the time period during which the pointer is valid.

IMAs are especially relevant for several reasons. First, they are caused by typical programming errors, such as array-out-of-bounds accesses and stale 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 injection vectors.

This paper is an extended version of our previous work [3], that presents a new dynamic technique for protecting programs against most known types of IMAs. The basic idea behind the technique is to use dynamic tainting, also known as dynamic information flow tracking (DIFT) [11], to link memory areas with their valid pointers. Every time memory is accessed through a pointer, our technique checks if 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. The access is considered legitimate if the taint marks match. Otherwise, an IMA is reported.

Because our technique is intended for efficient hardware-assisted implementation, one of the key goals in our design is to allow *runtime decisions* about the tradeoff between application performance and IMA detection probability. Whereas a software-only tool can select among any number of schemes that offer different tradeoffs, in a hardware-assisted tool the hardware cost would be the sum of hardware costs of all supported schemes. In effect, hardware support for each distinct scheme would be included in the hardware cost of a system even if that particular system never actually uses that scheme. For this reason, our technique should be *parametrized* such that the same scheme can be used to achieve different points in the performance-accuracy tradeoff. We achieve this parametrization by using a configurable number of taint marks, instead of using a distinct taint mark for each memory al-

- *I.Doudalis, M.Prvulovic and A.Orso are with the College of Computing, Georgia Institute of Technology, Klaus Advanced Computing Building, 266 Ferst Drive, Atlanta, GA 30332-0765. E-mail: {idoud,milos,orso}@cc.gatech.edu.*
- *J.Clause is with the University of Delaware, University of Delaware 18 Amstel Ave 439 Smith Hall Newark DE, 19711. Email: clause@udel.edu.*
- *G.Venkataramani is with Department of Electical and Computer Engg, The George Washington University, 801 22nd St Suite 624D Washington DC 20052. E-mail: guruv@gwu.edu.*

location. Limiting the number of taint marks can result in false negatives, because different memory regions and their pointers can have the same taint mark and an IMA where the address and the memory region happen to have the same taint mark would be undetected. Thus, the probability of IMA detection depends on how many taint marks can be used. The hardware-assisted performance of the scheme also depends on the number of taint marks – the number of bits needed to encode each taint mark determines how much extra capacity and bandwidth are used by the memory subsystem, and also how much extra latency is added by taint propagation circuitry. Overall, the number of taint marks can be used to select the desired point in the performance-accuracy tradeoff.

Our evaluation has dual goals: evaluating the ability of our technique to detect IMAs, and determining its effect on program performance. To assess IMA detection, we developed a software-only prototype that implements the approach for x86 64-bit binaries and protects stack, heap and global allocated memory and was used to perform a set of empirical studies. This prototype instruments the application's code using LLVM [16] and its runtime component is built within DYTAN [4], a generic dynamic-taint analysis framework. To determine the performance impact of the hardware-assisted implementation, we implemented another prototype within the SESC [23] computer architecture simulator that uses MIPS binaries. This two-pronged evaluation approach is needed because hardware simulation is extremely time-consuming, making start-to-finish simulations of real large programs with known IMAs infeasible. Instead, we use a software-only prototype to run these programs to evaluate our technique's IMA detection ability, but determine expected overheads using a benchmarking methodology traditionally used in computer architecture research – simulation of smaller applications and using only a representative fraction of the entire run.

Our experiments show that our proposed technique can identify a large number of IMAs, even when using only one-bit taint marks (only two unique taint marks). They also show that a hardware-assisted implementation imposes low time overheads, typically a few percent for a single taint mark, that grow moderately as the number of taint marks increases. These low overheads should make our scheme practical for use on deployed software.

The contributions of this paper are:

- A new technique for detecting IMAs that is effective and amenable to hardware-supported implementation.
- A design space analysis for hardware implementation of our technique.
- Two prototype implementations of the technique: a software-only one that works on x86_64 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 introduce an example that we use to illustrate our technique. The code shown in Fig. 1 is taken from a reference manual [5] and consists of a function that,

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

Fig. 1. An example IMA.

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 and 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 two memory-related faults. First, at line 7 we changed the terminating condition for the `for` loop from "`i < n`" to "`i <= n`", which causes the statement at line 8 to write a random character at position `buffer + n`. Because the address at offset `n` is outside the bounds of the memory area pointed to by `buffer`, accessing it through `buffer` is an IMA. The second IMA we introduced is that `buffer` is freed in line 10, so at line 11 the user-level library code in `printf` accesses memory that is no longer allocated.

The first IMA in this example is a *spatial* IMA – the access is illegal because a pointer accesses memory outside of the range that is valid for that pointer. The second IMA in our example is a *temporal* IMA – a previously valid pointer-memory association is no longer valid at the time of the access.

3 Our Technique

We first outline our technique at the source code level using an unlimited number of taint marks. Sections 3.2 and 4 then discuss how the technique works when the number of taint marks is limited and when operating at the level of binaries.

3.1 General Approach

Our technique is based on dynamic tainting, which is a technique for marking and tracking certain data at runtime. Our approach instruments programs to mark two kinds of data: memory in the data space and pointers. When a memory area m is allocated, the technique *taints* m with a taint mark t. When a pointer p is created with m as its referent (*i.e.,* p points to m's starting address), p is tainted with the same taint mark used to taint m. The technique *propagates* taint marks associated with pointers as the program executes. Finally, when memory is accessed using a pointer, the technique *checks* that the memory and the pointer have the same taint mark. Because pointers can be stored in memory, our technique actually stores two taint marks for each memory location, one associated with the memory location itself and the other for a pointer that may be stored in that location.

The rest of this section describes 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. There are four cases that our technique must handle: static memory allocations, dynamic memory allocations, pointers to statically-allocated memory, and pointers to dynamically-allocated memory.

Static memory allocations occur implicitly wherever a global or local variable is declared. The memory for 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 entry to f, by reserving space on the stack. For our example code in Fig. 1, assuming a 32-bit word size, 12 bytes of stack space are allocated (for `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 and global variable, and taints each individual area with a fresh taint mark. The memory area for a variable is identified using its starting address and the size needed to store that variable's type. In our example code, when `prRandStr` is entered, three fresh taint marks (*e.g.,* t_1, t_2, ad t_3) are created, using the first to mark the memory range $[\&i, \&i + sizeof(int))$, the second taint mark for $[\&seed, \&seed + sizeof(int))$, and the third for the $[\&buffer, \&buffer + sizeof(char*))$ range. 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.

Pointers to statically-allocated memory can be initialized in two ways. For a scalar variable, the *address-of* operator (`&`) 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 taints the pointer with the same taint mark that was used for the variable's memory. In our example, when the *address-of* operator at line 5 produces the starting address of `seed`, our technique retrieves the taint mark t_2 associated with `seed` and associates it with the address passed to `getSeedFromUser`. For statically allocated arrays, the name of the array is, for all practical purposes, a pointer to the first element of the array. In either case, we can create a *shadow pointer* that corresponds to each pointed-to region, and taint this shadow pointer with the region's taint mark (more detail on this is provided in Section 4).

Dynamic memory allocations, occur explicitly, as a consequence of a call to a memory-allocation function. In C and C++, there are only a few memory-allocation functions, and they all 1) take as input the size of the memory area to allocate and 2) return either the beginning address of a contiguous memory area of the requested size or `NULL` if the allocation is unsuccessful.

To taint dynamically-allocated memory, our technique intercepts calls to memory-allocation functions, (*e.g.,* `malloc`). When such a function is about to return successfully, the technique identifies the range of the allocated memory as $[r, r + size)$, 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, and taints the memory in this range with a fresh taint mark. In our example (Fig. 1), the call to *malloc* at line 3 would taint the range $[buffer, buffer + n)$ with a fresh taint mark (*e.g.,* t_4).

Pointers to dynamically-allocated memory are created either directly (as a return value of an allocation function) or indirectly (from another pointer). When our technique intercepts a memory allocation function that returns successfully and taints the allocated memory area with a fresh taint mark, it taints the function's return value (pointer) with the same taint mark. If other pointers are derived from that pointer, their taint mark is propagated to them as discussed in Section 3.1.2. In our example, the call to `malloc` returns a value tainted with t_4, then (as a result of the assignment at line 3) this taint is propagated to the `buffer` pointer.

3.1.2 Taint Propagation

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 define our propagation policy for data-flow only. Our propagation policy treats taint marks associated with memory and taint marks associated with pointers differently.

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 removed when it is deallocated. This removal of taint marks upon deallocation is implemented by intercepting memory deallocation and clearing (*e.g.,* setting to zero) the taint marks associated with that memory area. Note that pointers that were tainted with the memory area's taint mark can remain tainted. If such a pointer is used to access memory after its deallocation, the taint marks of the pointer and the memory location are different and an IMA is still detected.

Dynamically allocated memory is deallocated by calling a memory-deallocation function (*e.g.,* `free`), which is intercepted by our technique to clear the taint marks of the deallocated memory range. For the example in Fig. 1, the call to `free` at line 11 is intercepted and the taint marks for the memory region $[buffer, buffer + n)$ are cleared.

Statically allocated memory is deallocated when the function that allocated it returns (for local variables) or at program exit (for global variables). The latter case is irrelevant for our technique. To handle deallocation of local variables, our technique intercepts function exits and clears taint marks of the memory that corresponds to the function's stack frame. In our example code, when `prRandStr` returns, our technique clears taint marks associated with `prRandStr`'s stack, thus removing taint marks associated with memory that stores local variables `i`, `seed`, and `buffer`.

Propagation of Pointer Taint Marks: Unlike taint marks associated with memory, taint marks associated with pointers are propagated to derived pointers. To correctly propagate these taint marks, our technique must accurately model all possible operations on pointers and associate, for each operation, a propagation action that assigns to the result of the operation the correct pointer taint mark (using a zero taint to denote “untaintedness” of non-pointer values).

A superficial analysis of typical pointer operations can produce a reasonable initial propagation policy. 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 propagation policy defined in this way would be highly inaccurate and result in a large number of false negatives. In our preliminary investigation, we encountered dozens of cases where a simple policy falls short. To illustrate why simple policies are inadequate, we use the `strcpy` function of the C library as an example. This is a commonly-used function that copies the contents of a character array (src) to another character array ($dest$) under the assumption 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 to 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 copies the character at position s, to the memory location $s{+}dist$, incrementing s in each iteration until a string-termination character (zero in C/C++) is copied.

With the simple policy described above, this function always produces false positives – if src and $dest$ have taint marks t_{src} and t_{dest}, when offset $dist$ is computed it is an untainted integer which, when added to s, results in a pointer that has the taint mark t_{src} but points to the memory area of the $dest$ string. As a result, an IMA is reported for each write to the $dest$ string in the `strcpy` function.

To address this and other issues, we defined a more sophisticated policy based on our intuition and knowledge of the underlying assembly instructions and patterns found in the software subjects that we studied. We present a summary of our more final policy by discussing how it handles different operations:

Add, Subtract: **(**$c = a$ +/- b**).** If a and b are tainted with t_a and t_b, respectively, then c is tainted with $t_a + t_b$ (for addition) or $t_a - t_b$ (for subtraction). This accounts for a range of situations, such as adding to (or subtracting from) a pointer an offset computed by subtracting two pointers. In the `strcpy` code discussed above, $dist$ is now tainted with $t_{dest} - t_{src}$. When $dist$ is added to s, the result is now tainted with $t_{src} + (t_{dest} - t_{src}) = t_{dest}$, the correct taint for accessing the memory that contains the $dest$ string. This policy also produces correct t_c for the cases where a and b are untainted (t_a and t_b are zero, so t_c is also zero) and where one of them is tainted (pointer) and the other is not (*e.g.,* when t_a is non-zero and t_b is zero, t_c will be equal to t_a).

Multiply, Divide, Modulo, Bitwise OR and XOR: Independently from the taint mark of the operands, the result of any of these operations is never tainted.

Bitwise AND: **(**$c = a\&b$**).** Bitwise AND can have different semantics depending on the value of its operands. In particular, a program may AND a pointer and an integer to get a base address (mask out the lower bits of the pointer). For example, “`c = a & 0xffffff00`” masks out the lowest eight bits of pointer `a`. Bitwise AND can also be used to compute an offset (*e.g.,* “`c = a & 0x0000000 0f`”). 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, c is tainted with t if the result preserves the most-significant bits of the tainted value; the rationale for this is that the operation is (or might be) computing a base address for some memory range.

Bitwise NOT: (c = ∼ a). Bitwise NOT can be used as an alternative to subtraction, *e.g.,* “`c = b - a - 1`” could be optimized into “`c = b + ∼ a`”. Thus, our taint propagation rule for bitwise NOT is $t_c = -t_a$. Note that this also correctly handles bitwise NOT of untainted values (t_a is zero).

It is important to note that clever or unusual combinations of operators can result in sequences that are not handled correctly by any specific policy, and similar problematic sequences can be created for any IMA detection scheme that is defined at the binary level. Therefore, it is unlikely that any propagation policy can be proven to be sound and complete. However, our policy works correctly for all the software that we studied so far, as discussed in Section 5. If additional experimentation reveals shortcomings of our policy, we will refine it accordingly.

3.1.3 Checking

To check legality of memory accesses, our technique intercepts all memory accesses and compares the taint mark of the pointer used to access memory with the taint mark of the accessed memory location. These taint marks are equal for legitimate memory accesses, so the access is considered an IMA when the taint marks are different (including the case where one is tainted and the other is not). Currently, our technique halts program execution when it detects an IMA. However, it could also perform different actions, such as attaching a debugger or 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, friendly versus antagonistic environments, etc.).

3.2 Limiting the Number of Taint Marks

Ideally, our technique would use an unlimited number of unique taint marks because it would allow 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.

In our implementations, each taint mark is represented with n bits, limiting the number of distinct taint marks to 2^n. Although it is possible to use a large number of bits (*e.g.,* 64 bits) for a virtually unlimited number of taint marks, the storage and manipulation of large taints would introduce unacceptable overheads. As stated previously, our approach stores two taint marks for every memory location—one for the memory location, and the other for the pointer stored in that location. Two 64-bit taint marks per byte of data would result in a 16-fold memory increase, which is prohibitive for many applications.

Most importantly, large taints would prevent a practical hardware-assisted implementation. There are two primary reasons for this. 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 to keep the size of taint marks small relative to the size of the corresponding data. Second, large taints would dramatically affect the design complexity of the hardware.

Based on these considerations, a key feature of our scheme is that the number of taint marks can be small while still 1) avoiding false positives and 2) retaining the ability to detect all types of IMAs. The drawback of using fewer taint marks is that they have to be reused, so several allocated memory areas may have the same taint mark. When a pointer intended to point to one area is used to access another that has the same mark, the resulting IMA would be undetected. In other words, when using a limited number of taint marks, IMAs are detected *probabilistically*. Assuming n-bit taint marks, a uniformly-distributed random assignment of taint marks to memory regions, and IMAs where a pointer accesses a random memory region, the probability of detecting each IMA is equal to $P = 1 - \frac{1}{2^n}$. This is encouraging: even for a single taint mark (1 bit), our technique may still find 50% of all IMA events; with 2-, 4-, and 8-bit taints we would expect to detect each IMA with 75%, 94%, and 99.6% probability, respectively.

Moreover, these estimates may actually be low for two reasons. First, many IMAs occur because a pointer is used to access memory slightly outside the bounds of its intended referent. Ensuring that abutting regions get different taint marks would mean that these IMAs will be caught. Second, in cases where a single defect can result in multiple IMAs, even a 50% probability of detection for each IMA event results in a much higher probability of revealing the underlying defect by detecting at least one IMA caused by it.

Even though limiting the number of taint marks makes the technique probabilistic, it does not lead to false positives. A pointer and its corresponding memory region will always have the same taint mark. This gives us the ability to *tune* the number of taint marks 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 previous sections present the technique at the source-code level, our prototype implementations actually operate at the binary level. This approach allows our technique to work correctly even when no source code is available for parts of the application, such as when using dynamically loaded off-the-shelf modules and libraries. Working at the binary level also facilitates a hardware-assisted implementation, where the hardware accelerator operates at the machine level with little or no knowledge of the source code structure. The role of this hardware accelerator in our technique is to propagate taints, while the software is responsible for correctly initializing the taint information using the ISA (similar to FlexiTaint [30]) extensions provided by the hardware. In this section, we provide details of the two implementations (software-based and hardware-assisted) and discuss the main differences between them.

4.1 Operating at the Binary Level

When operating at the binary level, our technique can still use the same taint propagation rules described in Section 3, but loses the necessary information for initializing taints correctly. To retain that information, the runtime library needs to be modified to intercept heap allocations and deallocations, and the compiler must instrument the application's code to appropriately initialize memory and pointer taints for global- and stack-allocated memory.

Handling of dynamically allocated objects requires that we modify the heap allocation library to appropriately initialize the taint as described in Section 3.1.1. We note that use of custom memory allocators by the application, even if they are not changed to initialize memory and pointer taints, does not result in false positives in our technique. Assume that an application allocates several MBytes through `malloc` and internally partitions them. The allocated memory area will have a single memory taint, and all the pointers pointing to its partitions will have the same pointer taint as the original pointer to the array (return value of the `malloc` function) because they have all been derived from it. This eliminates false positives, but does prevent identification of IMAs within that area (*e.g.,* using a pointer to one partition to access another). To enable IMA detection in this case, the author of the memory allocator should modify the custom allocator to insert appropriate taint initialization and clearing code.

For the case of statically allocated objects on the stack, *e.g.,* a stack-based buffer, the address of the object is determined at runtime and it is relative to the stack pointer of the program. The compiler instruments the code to initialize the memory taint of the buffer *e.g.,* using the ISA extensions. However, subsequent accesses to the buffer are ordinarily often computed by adding a constant offset to the stack pointer, which has zero pointer taint in order to access the non-tainted stack variables, and would result in false positives, because the resulting pointer will not have the "correct" taint. In our prototype implementation, we overcome this problem by introducing a `shadow` pointer for every tainted area (*e.g.,* a buffer) in the stack. The prologue of each function initializes these shadow pointers to point to the corresponding stack-allocated memory areas, and initializes the memory taint of each memory area and the pointer taint of its shadow pointer. The body of the function is then changed to no longer use the stack pointer to access these memory areas, but use the shadow pointers instead. Similarly, the *address-of* operator now simply copies the shadow pointer, propagating it to the newly created pointer. Finally, the epilogue of each function clears the memory taints of the stack-based objects that are freed at that point. The same approach is also used for global statically allocated memory – the only difference is that initialization of the memory taint occurs when global constructors and initializers are being called. Finally, the memory taint of the shadow pointers themselves is zero, protecting them from any accidental or malicious overwrites. Similarly to the shadow pointers the memory taint of the return address is also zero to prevent possible attacks.

These changes ensure that each statically allocated object is now accessed only through its shadow pointer or pointers copied or derived from it, just like a dynamically-allocated object is only accessed through the pointer returned by an allocation function (*e.g.,* malloc) or pointers copied or derived from it. Furthermore, clearing of memory taints when we leave the object's scope makes the object's shadow pointer no longer valid for accessing it, also just like pointers to dynamically allocated memory stop being valid because deallocation clears memory taints of the memory that is being deallocated.

Finally, our technique operates without false positives inside shared libraries and other binaries for which the source code is not available. Inside such code, our technique will not detect IMAs that only involve stack-based objects from within the uninstrumented code, but will still detect heap-based IMAs and IMAs where a pointer from the instrumented part of the code is used to access memory that belongs to the uninstrumented code, or vice versa. It should be noted that this is neither a fundamental limitation in our scheme nor is it endemic to our scheme: any IMA detection scheme would be unable to detect IMAs in code whose pointer-memory associations cannot be conveyed to the scheme, and dynamic recompilation at the binary level would be able to overcome this problem (at least for stack-based objects) by instrumenting function entry and exit code even when no source code is available.

4.2 Software-based Implementation

To create our software-based prototype, we added a pass in LLVM 2.6 [16] to taint all stack and global defined arrays using the shadow-pointer approach described in Section 4.1. The taint propagation is implemented using DYTAN [4], a generic dynamic tainting framework which is itself built on top of the `Pin` dynamic-instrumentation framework [18]. Taint initialization requests from the instrumented application code are conveyed to our Dytan-based runtime implementation via function calls that are intercepted by `Pin`.

Although `Pin` allows our implementation to handle shared libraries, it cannot instrument the underlying Operating System (OS), so our implementation must recognize system calls and and account for their effects on memory and pointers. These effects are relatively simple to model and account for, so system call handling in our prototype is more of a tedium than a conceptual difficulty. Signals are only slightly more difficult to handle than ordinary function calls – the OS performs a context switch and allocates a new stack frame before calling the handler function in the application, so our implementation must suitably initialize and clear taint information in the handler's entry and exit code, respectively.

4.3 Hardware-based Implementation

As explained in Section 3, one of the key advantages of our IMA detection technique is that the bulk of the performance overhead in the software-only implementation is due to propagation of pointer taints and comparisons between pointer and memory taints when accessing memory, both of which are amenable to hardware acceleration. Hardware acceleration of taint propagation and checking have already been discussed in the literature, both for fixed-functionality security schemes (*e.g.,* Minos [6]) and programmable ones (*e.g.,* Raksha [19] and FlexiTaint [30]).

Because our technique's taint propagation and checking needs differ from those in prior tainting work, in this section we briefly re-examine the design space to identify good candidates for an IMA-detection accelerator. The primary parameters of this design space are:

- How are taints processed and stored – tightly coupled with data in a "data-widening" implementation such as the one used in Minos [6] or Raksha [19], or separately from data in a "decoupled taint" implementation such as the one used in FlexiTaint [30]
- How are taints propagated and accesses checked – by hard-wiring the rules or programming one of the programmable accelerators proposed in the literature?

4.3.1 Taint processing and storage: Data-widening or Decoupled Taint?

The most straightforward way to implement a hardware DIFT scheme is to simply extend (widen) each word by a few bits to accommodate the taint information. This widening applies to all parts of the system where values can be stored or transmitted, including memory, registers, data buses, forwarding logic, etc. The main appeal of this approach is that taint information is simply an extension of a data value and naturally flows together with it. Whenever data values are operated on in an ALU, the taint bits of the operands can be processed in parallel with that ALU operation to produce the taint bits for the result. The main disadvantage is that it requires extensive changes (*e.g.,* widening of buses, forwarding logic, memory locations, registers, etc.) across the entire processor pipeline and the memory subsystem, and prevents use of standard memory modules and buses (must be widened to add taint bits).

Another way to implement our DIFT-based hardware IMA detector is to decouple taint storage and processing from data. A similar decoupling approach has been adopted in prior schemes for program monitoring, such as programmable monitoring of memory accesses in MemTracker [31], programmable DIFT support in FlexiTaint [30], and even IMA detection using "safe pointers" in HardBound [7]. In this approach, taint information is stored as a packed array in a reserved part of the application's virtual address space. This reserved virtual space is managed by the Operating System

(OS), allowing taint pages to be paged in and out similar to normal data pages. Existing page-level access controls can be simply extended to protect taints from accidental or malicious overwrites using normal data access instructions while allowing them to be initialized, propagated, and checked by the DIFT mechanism itself. Finally, a separate and small L1 cache can be dedicated to storing taints, preventing any pollution in the existing data L1 cache and avoiding any extra contention on its ports. Because taints are typically much smaller than the corresponding data, the L2 cache and memory capacity and bandwidth can be shared between data and taints, so they store taint regions of memory just like any other memory. More details on this implementation can be found in MemTracker [31] and FlexiTaint [30], including details on how a decoupled-taint approach can be implemented with minimal impact on performance-critical parts of the out-of-order instruction execution engine found in modern processors, as well as how it interacts with the OS.

Most prior proposals for hardware-assisted DIFT use this data-widening approach [6], [19]. These mechanisms use DIFT primarily to identify unsafe uses of "untrusted" values, typically using a one-bit taint to mark "untrusted" values that come from external sources, such as the network, propagating the "untrusted" taint to derived values, and detecting a possible attack when an "untrusted" value is used in an unsafe way.

The conceptually straightforward data-widening approach is very attractive for such schemes: one additional bit in each memory word in memory or in a bus can be accommodated relatively easily, *e.g.,* by repurposing ECC bits (if some degradation in reliability is acceptable). Also the additional bit in the data paths, caches, registers, etc. of the processor chip results in only a minor increase in area and circuit latency.

For our IMA detection technique, however, more than one taint bit is needed for each memory location, and ability to support multi-bit taint marks is also highly desirable: one of the key features of our IMA detection technique is that it permits a cost-accuracy tradeoff, i.e. using more taint bits to improve detection accurately at the cost of memory (to store taints) and performance (to process and manage larger taints), or using fewer taint bits to minimize cost with some degradation in accuracy.

Because the data-widening approach adds a taint mark to every data location and register, it requires the maximum number of supported taint marks to be decided at hardware design time. A data-widening implementation eliminates this important advantage. The maximum number of taint bits is decided at runtime, and the cost (additional bits in memory, registers, etc. and performance degradation due to larger circuitry and caches) is paid for that maximum number of bits. As a result, it makes little sense to not use anything but the maximum number of taint bits.

In contrast, a decoupled-taint implementation allows the desired point in the cost-accuracy tradeoff to be decided for each system or even application separately: how much memory and L2 space is occupied by taints depends on taint size; how much performance overhead is incurred also primarily depends on taint size (larger taints consume more bandwidth and suffer more misses in the taint L1 cache and in the shared L2 cache).

At one extreme of the cost-accuracy tradeoff, the technique can be turned off – no IMAs will be detected, but no memory is used for taint storage and the taint propagation and checking support in the processor pipeline can be turned off and kept outside the processor's critical path [30]. At the other extreme, a very large number of taints (*e.g.,* 32-bit or even 64-bit taints) can provide each area of memory with its own unique taint mark, resulting in detection of all IMAs but with a large cost in terms of memory space and performance; half or more of memory and L2 space is used for taints, and the small taint L1 cache is suffering large numbers of misses.

Storing taints as packed arrays in memory using the decoupled-taint approach has a secondary advantage in terms of performance – rapid initialization and clearing of memory taints in a given (allocated or deallocated) block of memory, because each memory word in the taint storage area stores taints for several consecutive data locations.

Overall, for our IMA detection technique, the decoupled-taint approach has multiple important advantages over the data-widening approach. As a result, the prototype hardware-accelerated implementation of our IMA detection scheme uses the decoupled-taint approach.

4.3.2 Taint propagation and access checking: Hard-wiring or programming an accelerator?

Our IMA detection technique can be implemented by hard-wiring our taint propagation rules (Section 3.1.2) and pointer-memory taint checks (Section 3.1.3) into the taint propagation and checking engine of a decoupled-taint DIFT accelerator. However, it would be highly desirable if these propagation rules and checks could be implemented using one of the previously proposed programmable DIFT accelerators: the cost of tainting support could then be amortized between existing DIFT-based schemes (detection of unsafe uses of untrusted data) and our new IMA detection scheme. An additional advantage of using programmable accelerators lies in future-proofing: hard-wired approaches would require a hardware upgrade each time DIFT rules are upgraded, whereas a scheme using a programmable accelerator can be upgraded by reprogramming the accelerator.

It should be noted that future-proofing, while still a possible concern, is not as pressing for our IMA detection scheme as it is for prior uses of DIFT that interpret the taint mark as an indication of whether the data is trusted or not by propagating the trusted/untrusted property to derived values and identifying unsafe operations that use untrusted values (*e.g.,* a jump instruction that uses an input-derived target address). In such schemes, different interpretations of trust and safety have resulted in many different sets of DIFT rules, and updates to DIFT rules may be needed as the interpretations of trust and safety are revised in the face of new attacks. In contrast, DIFT rules in our scheme mirror legal ways of deriving pointers from one another. Whether a value is a pointer and how it is derived are well-defined properties, so we do not expect frequent changes in our DIFT rules. Still, we cannot rule out the need for such upgrades, so an implementation that uses a programmable accelerator would still be desirable.

Unfortunately, our technique has four characteristics that are at odds with the assumptions that were made in the design of prior programmable accelerators. First, the resulting pointer taint mark for an operation (*e.g.,* ADD or SUB) is not a simple copy or a result of a logical operation (AND, OR, etc.) of the sources' taint marks – *e.g.,* for ADD, the taint mark of the result is the sum of sources' taint marks. This kind of propagation can only be implemented using an exception handler for each ADD/SUB/NOT, etc. instruction[1] in the Raksha [19] DIFT accelerator as it was originally described. However, it would be relatively simple to add hardware support for these taint operations to Raksha's repertoire. In FlexiTaint [30], the hardware accelerator can be programmed to perform these operations, but the performance degradation would be somewhat higher due to the large number of possible input-output taint combinations that could cause misses in FlexiTaint's TPC (a small cache used to memoize output taints for recently encountered input taint combinations). The second obstacle is that our technique divides taint information into pointer and memory taints which are propagated differently, then compares the pointer taint of the address with the memory taint of the location being accessed. In Raksha, different rules would require pointer and memory taints to be treated as different taint propagation schemes, which precludes comparisons of pointer and memory taints without resorting to exceptions[2] in Raksha [19]. Although extensions to Raksha that would enable such cross-scheme checks are feasible, they would not be straightforward. In FlexiTaint, the two separate taints can be implemented together, but would increase the number of frequently seen input taint combinations (which again reduces the effectiveness of its TPC). The third difficulty is that both schemes do not treat the taint of the destination (*e.g.,* memory location in a store instruction) as one of the input taints for the operation, which prevents them from comparing the memory taint of the target memory location with the pointer taint of the address in a store-to-memory operation[3]. Again, either scheme can be extended to support this, but the extension would be non-trivial, would increase the schemes' cost, and result in additional performance overhead. The final (fourth) and most pressing problem is that our taint propagation rules determine the resulting taint using not only the taints of the input operands, but also their data values – *e.g.,* to handle the `AND` operation[4] as described in Section 3.1.2. Extensions to allow efficient and programmable consideration of data values in taint propagation in either scheme (Raksha or FlexiTaint) are an open research problem.

As a result of these considerations, we chose a hard-wired approach for our implementation. However, we do expect that further work on programmable taint propagation accelerators to eventually allow an efficient implementation of our technique using a programmable DIFT accelerator. In fact, we hope that the above discussion of the existing accelerators' shortcomings will serve as motivation for improving their flexibility in this direction.

1. These instructions represent about 20% of all dynamic instructions in the benchmarks we used in our experimental evaluation. Even with a 10-instruction exception handler (which is extremely optimistic), the total instruction count would increase three-fold

2. Load and store instructions, which require these checks, represent about 39% of all dynamic instructions in the benchmarks we used in our experimental evaluation; even a 10-instruction exception handler would result in a five-fold increase in the total instruction count

3. 12% of dynamic instructions in our experiments, so an exception-based workaround would result in a two-fold instruction count increase

4. This operation represents only 0.5% of dynamic instructions in our experiments, so an exception-based workaround with 10-,20-,and 30-instruction handlers would increase the instruction count by 5%, 10%, and 15%, respectively, and likely cause somewhat higher performance degradation (exceptions cause expensive pipeline flushes in modern processors)

5 Empirical Evaluation

The goal of our empirical evaluation is to assess the 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 when using only a small number of taint marks?

RQ2: Does our technique erroneously report an IMA for any legitimate memory access?

RQ3: How much runtime overhead is a hardware-assisted implementation of the technique likely to impose?

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

5.1 Experimental Subjects and Setup

In our empirical studies, we used two sets of subjects. The first set consists of applications with known illegal memory accesses, shown in Table 1. Most of these applications are from the BugBench [17] suite. Four additional subjects were obtained by browsing on-line bug databases (CVE[5]): `pine` v4.44, an email and news client, `mutt` v1.4.2.li, an email and news client, `gnupg` v1.2.2, an implementation of the OpenPGP standard, and version 5.2.0 of the `php` language. `Pine`, `mutt`, `gnupg`, and `php` each have one known heap based IMA. Finally we also used a testbed [32] that performs various attacks exploiting stack, heap and global buffers. We use all these subjects to investigate RQ1.

Our second set of subjects consists of the twelve applications from the integer component of SPEC CPU 2000 [26]. 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`. 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. For RQ3, we used the SPEC CPU2000, SPEC CPU 2006 and Splash-2 [27] benchmark suites, a detailed list of the benchmarks we ran can be found in Section 5.4.

5. http://cve.mitre.org/

In our experiments (both x86_64 software-only and MIPS hardware-assisted), we use taints at the granularity of 32-bit memory words. This creates some risk of false positives and negatives when sub-word accesses are used, but no such problems exist in any of the applications we used: pointers stored in memory are word-aligned so word-granularity pointer taints are sufficient, dynamic memory allocation in the standard library is in terms of chunks that are at least double-word-aligned, and existing performance optimizations of stack and global memory result in pointer-accessed variables (i.e. arrays) being word-aligned as well. Note that this word-granularity tainting is again a cost-performance choice, not a fundamental requirement: when byte-granularity tainting is used instead, taints would use four times as many bits per word of data, i.e. two taint marks at byte-granularity would result in performance similar to the performance results we show for 16 taint marks, four taints marks would result in performance similar to what we show for 256 taint marks, etc.

5.2 RQ1

To address RQ1, we ran all applications from our first set of subjects while protecting them with our software-based tool configured to use only two taint marks. For each known IMA in these applications, we ran the application, reproduced the IMA, and checked if our tool detected it. The results of the study are shown in Table 1. For each IMA, the table shows the *application* containing the IMA, the *IMA location*, the *type* of the illegal access, and if our prototype *detected* the IMA. The type of the overflow can be either sequential (seq), where the contents of an adjacent memory location are overwritten, or random (rnd) where the base pointer of the buffer is used to access potentially any memory location.

As the table shows, all IMAs were detected by the prototype. Recall that we only expected about 50% of IMA occurrences to be detected when using only two taint marks, whereas our experiments indicate that the detection rate is 100%. This apparent discrepancy is a result of the fact that many IMAs in these test subjects (and likely in software in general) tend to involve abutting regions of memory – we try to ensure, whenever it is feasible, that abutting regions of memory have different taint marks, thus detection of IMAs that involve such region is highly likely. This result is very encouraging because it indicates that even with a very limited number of taint marks our technique can detect nearly all real heap, global and stack-based IMAs of the most common variety (those involving abutting regions) and a significant percentage (50% for two taint marks, 75% for four, etc.) of all other (non-abutting-region) 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 CPU 2000 integer benchmarks using our software-based tool, ran each of them 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.

TABLE 1
Results for RQ1 (effectiveness).

	Application	**≈kLoC**	**IMA location**	**Overflow & IMA Type**	**Detected**
BugBench	bc-1.06	14.4	storage.c:177	seq heap	✓
	bc-1.06	14.4	util.c:577	seq heap	✓
	bc-1.06	14.4	bc.c:1425	seq global	✓
	compress	1.9	harness.c:234	rnd global	✓
	gzip-1.2.4	8.1	gzip.c:457	seq global	✓
	go	29.2	g27a.c:137	seq global	✓
	man-1.5h1	4.1	man.c:978	seq global	✓
	ncompress	1.9	compress42.c:896	seq stack	✓
	polymorph-0.4	0.7	polymorph.c:120	seq global	✓
	polymorph-0.4	0.7	polymorph.c:193	seq stack	✓
	squid-2.3	93.5	ftp.c :1024	seq heap	✓
	Wilander et al [32]	0.6		seq {stack, heap, global}	✓
CVE	gnupg-1.4.4	117.3	parse_packet.c: 2095	rnd heap	✓
	mutt-1.4.2.li	453.6	utf7.c:199	seq heap	✓
	php-5.2.0	558.2	string.c:3152	seq heap	✓
	pine-4.44	221.9	rfc822.c:260	seq heap	✓

Although we observed no IMAs for eleven of the twelve applications, the prototype reported an IMA for `255.vortex`. After examining `255.vortex`'s source code, we discovered that the IMA reported was indeed caused by a heap-based temporal fault (dangling pointer) in the code. We then checked the documentation for the SPEC benchmarks and found that this is a known fault in 255.vortex that was corrected in the subsequent release of the benchmarks, and that this was the only known memory-related fault in that release of the benchmark suite.

Overall, the results for RQ2 are fairly positive: our technique generated no false positives, and was also able to detect the only actual memory-related fault in this 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 [4]. 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 result, the overall overhead of the software-based implementation varies between 100x and 500x, depending on the application. Second, no software implementation based on binary re-writing can approach the efficiency of a hardware-assisted implementation due to the intrinsic cost of instrumenting almost every instruction in the code.

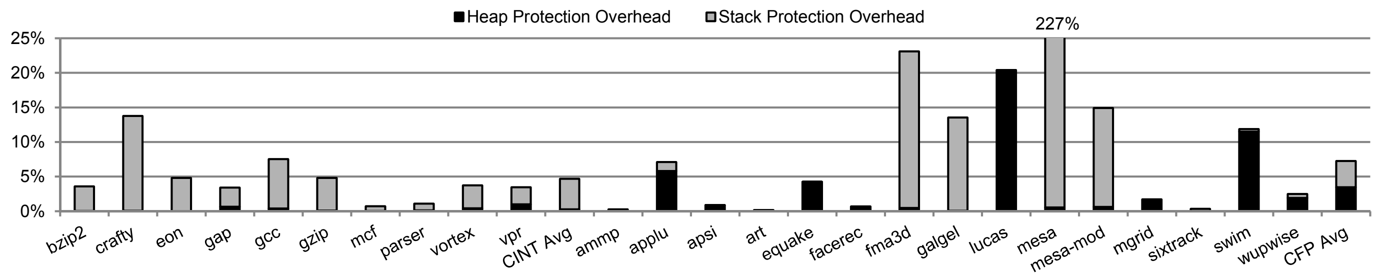

Fig. 2. SPEC 2000 performance overhead

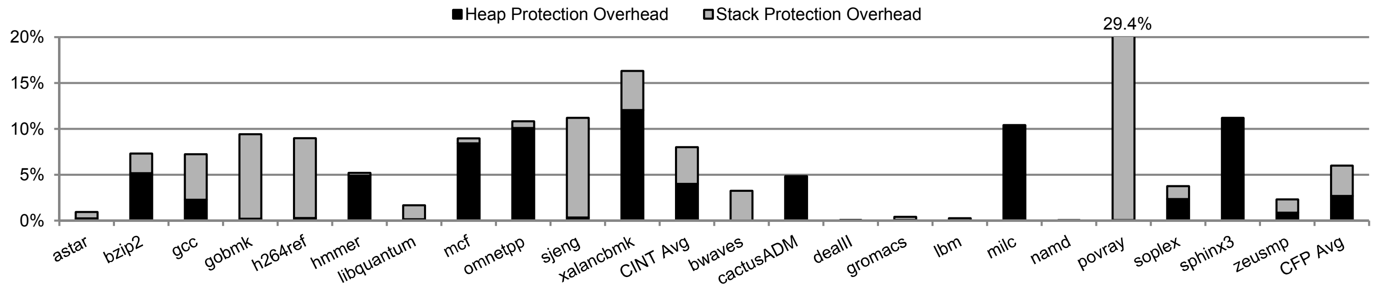

Fig. 3. SPEC 2006 performance overhead

The hardware implementation of our technique has three potential sources of overhead: 1) execution of code instrumented by LLVM [16], 2) initialization of memory taints when allocating and freeing memory, and 3) taint propagation in hardware can cause stalls in the taint processing unit and/or make taints and data compete for cache space. In the evaluation of our technique we believe that the most important points in this performance cost/detection accuracy tradeoff are the ones that allow detection of most IMAs, while suffering relatively small performance overheads – such settings would allow always-on use of our technique (continuous protection, even in production runs). Therefore, our evaluation will focus on configurations that use 2, 4, 16, and 256 distinct taint marks per word – this corresponds to using 2, 4, 8, and 16 taint bits for each word in memory; half of the taint bits are used for the memory taint of the memory location, and the other half for the pointer that stored in it (or to mark the value as a non-pointer).

To evaluate the taint propagation overhead, we employ cycle-accurate simulation using the open-source SESC [23] architectural simulator. We model a four-core multiprocessor, with Core2-like 4-wide out-of-order superscalar cores running at 2.93GHz. Each core has a private 32KByte, 8-way set-associative, dual-ported Data L1 cache with a 64-byte line size. All cores share an L2 cache that is 8MBytes in size, 32-way set-associative, single ported, and also with a 64-byte line size. The Taint L1 cache is 8KByte, 4-way associative, dual ported, and with a 64-byte line size. To determine taint initialization overheads, we added a taint initialization instruction that adds two cycles to the execution time to process each 32-bit word in the packed taint array. Finally, we had to estimate the overhead of running LLVM-instrumented code. LLVM 2.6 [16] does not provide, as of now, a stable MIPS back-end (SESC [23] uses the MIPS ISA). We estimate the instrumentation overhead by running an uninstrumented and then a LLVM-instrumented x86_64 version of the application code on a Xeon X5450 running at 3.0GHz. We then apply this overhead to the MIPS version of the code. This assumes that the instrumentation would cause similar slowdown in MIPS code and in x86_64 code. This is likely a conservative assumption (overestimates the overhead) because the instrumented code could benefit from the larger number of registers available in the MIPS ISA.

In our evaluation, we use SPEC 2000 and SPEC 2006 benchmarks [26], shown in Figures 2 and 3 respectively, all executed with reference input sets. The only omitted applications are those for which the baseline LLVM (without our instrumentation pass) did not produce a correct executable. To achieve reasonable simulation time, we fast-forward through the first 5% of the application's execution (to skip initialization) and then simulate 2 billion instructions in detail. For evaluating multi-threaded applications, we use all benchmarks from the Splash2 [27] suite with reference input sets, and simulate their execution from start to end when running with four threads.

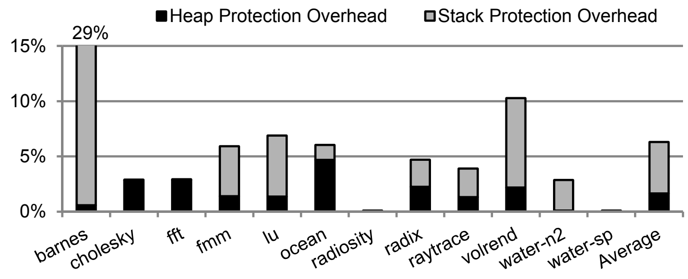

Fig. 4. Splash-2 performance overhead

Fig. 2, 3, and 4 present the performance overhead, broken

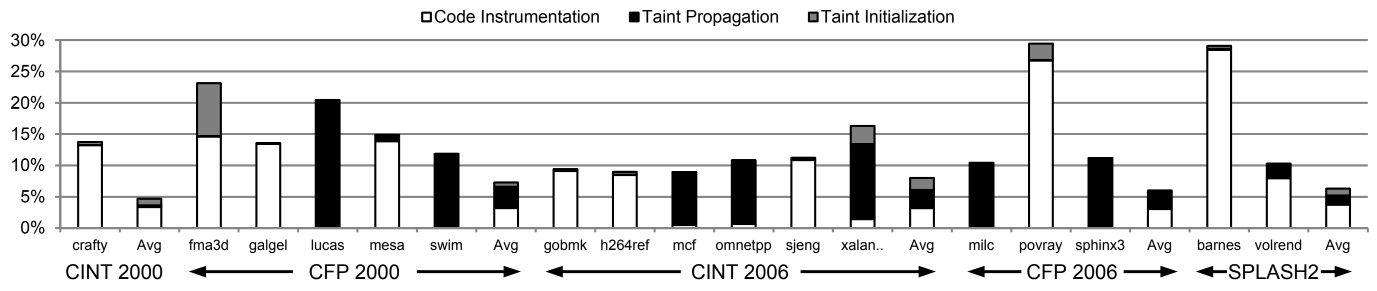

Fig. 5. Breakdown of the performance overhead

down into the overhead of IMA detection in the heap and in the stack (the overhead for global memory protection is negligible) when using four distinct taint marks[6].

The heap protection overhead accounts for the initialization of heap memory taints during allocation and deallocation, as well as *all* of the taint propagation cost. The stack and global protection overhead includes initialization of stack and global memory taints, as well as the instrumentation overhead (mostly from changing stack-pointer-based accesses to use our shadow pointers as described in Section 4.1). To gain more insight into these sources of overhead, Figure 5 breaks down the total overhead according to its cause (instrumentation, taint initialization, taint propagation) for applications whose total overhead is more than 9%, and also for the average of all simulated applications (not just the ones shown in Figure 5) in each benchmark suite.

In the benchmarks that exhibit higher overheads, the two factors that dominate are the taint propagation and the instrumentation necessary for taint initialization of the stack memory. For applications where the taint propagation overhead is dominant (*e.g.,* `lucas`, `mcf`, `omnetpp`, `xalancbmk`, `milc`, `sphinx3`), it is caused mainly by pipeline stalls due to misses in the Taint L1 cache and by competition between data and taints for L2 capacity. In some situations, the increase in the L2 miss rate also leads to increased contention for L2 bandwidth. This effect is especially pronounced in SPEC2006 benchmarks, which tend to be more memory-intensive than those in SPEC2000.

The instrumentation and initialization overhead is dominant in applications which allocate global or stack buffers often (*e.g.,* `crafty`, `fma3d`, `gobmk`, `h264ref`, `mesa`). A pathological case was mesa, which allocates stack buffers in frequently called functions for the maximum possible texture size (several KBytes), even when the application operates on much smaller textures (whose size is passed as a parameter to the function). This results in initializing taints for many stack locations at each function entry and exit point, then actually accessing only the first few of these locations. By changing the code to allocate only the necessary buffer size (*e.g.,* in texture.c:2265, change the array size from `PB_SIZE` to `n`), the taint initialization overhead is reduced by an order of magnitude (and the total overhead of IMA detection is reduced from 227% to 15%).

6. This corresponds to a total of 4 taint bits per memory location, two for the memory taint and two for the pointer taint

On average, SPEC2000 benchmarks exhibit a performance overhead of 7%. For SPEC 2006, the average is 8%, and in Splash2 the average is 6%. In the Splash2 benchmark suite, taint propagation overheads are increased because of coherence misses needed to keep taints in memory coherent. Instrumentation overheads in these benchmarks are also relatively high, primarily because their execution times are generally much shorter than in SPEC 2000 and especially SPEC 2006, resulting in reduced amortization of instrumentation overheads in initialization, setup, and cleanup code.

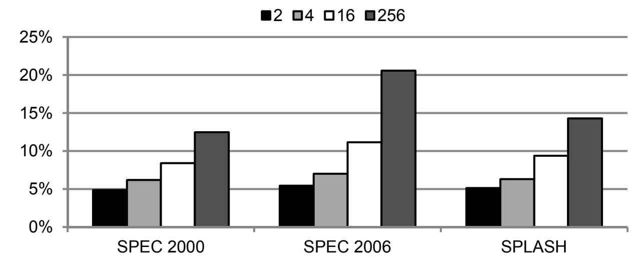

Fig. 6. Performance overhead vs. number of taint marks

Fig. 6 shows the average performance overheads across our benchmarks for various numbers of taint marks. We observe that, as we increase the number of taint bits, the overheads increase accordingly. The increase is non-linear because some of the overheads (*e.g.,* the overhead of running instrumented code) are fixed, and also because the taint L1 miss rates and the contention for L2 space are non-linear phenomena that depend on working set sizes. For applications with smaller working set sizes, data and taints can fit in the L2 cache even when large taints are used. When working sets are large (as in several SPEC 2006 applications), however, contention for L2 space exists even with the smallest taint size and gets worse as taint size is increased. As a result, SPEC2000 benchmarks still have a nominal 13% overhead even for 256 taint marks (16 taint bits for each memory location), whereas SPEC2006 exhibits overheads of 11% for 16 taint marks (8 taint bits for each memory location) and 21% for 256 taint marks.

In Splash2 benchmarks, the overhead steadily increases in spite of their small working sets. This overhead is caused by increased coherence activity, which almost doubles as we double the number of taint bits. The increased coherence activity can be attributed in part to false-sharing of lines containing taint bits – the taint bits of adjacent memory regions, which are mapped in different cache lines and cause no false-sharing when accessed, will reside in the same line. Although taint coherence traffic is still much smaller than data coherence traffic, it still contributes to interconnect contention and causes additional performance overhead.

Finally, we note that even the "large" overheads in SPEC2006 are still much lower than any software IMA detection scheme, such as a software implementation of our DIFT-based IMA detection (100X slowdown). Moreover, our technique loses very little efficiency when applied to multi-threaded code. In contrast, software-only approaches typically suffer very large additional overheads when applied to multi-threaded code, mainly because extensive lock/unlock activity is needed to ensure atomicity of data and meta-data (taints in our technique) updates and checks. We also note that some of our overheads (*e.g.,* the instrumentation overhead for detection IMAs in stack and global memory) may be reduced based on static analysis and optimizations similar to those applied in software-only IMA detection work (*e.g.,* SoftBound [20]) – in this proof-of-concept implementation, shadow pointers are used for all stack and global allocations, although many can be eliminated because the corresponding checks are redundant.

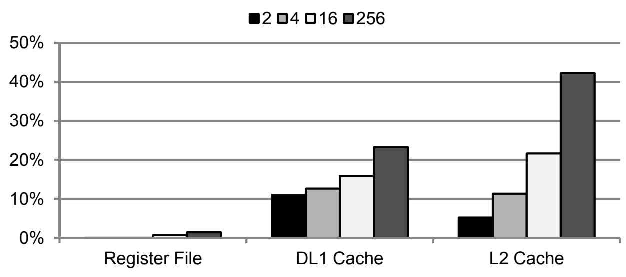

Fig. 7. Cache latency vs. number of taint marks.

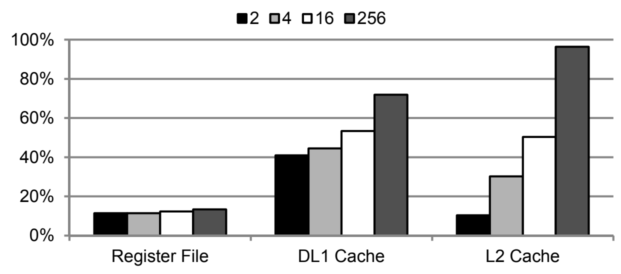

Fig. 8. Power overhead vs. number of taint marks.

Analysis of a Data-Widening Implementation: Our qualitative analysis in Section 4.3 has resulted in choosing a decoupled-taint implementation over a data-widening one. The primary reason for this choice is that data-widening incurs overheads that depend on the *maximum* number of supported taint marks, whereas overheads in decoupled-taint implementations mainly depend on the number of *actually used* taint marks. This choice allows a runtime selection of the cost-accuracy tradeoff.

However, our rejection of data-widening may still be flawed if overheads of data-widening with a large number of supported taint marks are still lower than with a decoupled-taint approach. Therefore, we performed additional quantitative analysis to estimate the overheads of a possible data-widening implementation. Figure 7 shows the latency increase and Figure 8 shows the energy consumption increase, both obtained using the Cacti v5.0 [28] tool, from data widening in the register file, L1 cache, and L2 cache. Both of these increases are mostly caused by the increase in the structure's size (i.e. longer bit-lines and other wires). From these figures, we see that data-widening causes significant energy and latency increases in L1 and L2 caches when using more than 4 taint marks. In modern energy-constrained processors, this is likely to reduce the processor's clock frequency and/or its IPC, resulting in performance overheads similar to those incurred by a decoupled-taint implementation when using a similar number of taint marks.

6 Related Work

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

Program-analysis-based tools (*e.g.,* [9], [10], [12], [14], [33]) 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 [15] and CCured [21], are also static in nature; they attempt 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 or program transformations can be considered complementary to dynamic approaches in terms of strengths and weaknesses.

Dynamic approaches for IMA detection instrument a target application to perform runtime monitoring. Instrumentation can either be done at the source-code level (*e.g.,* [8], [24], [34]) or a the binary level (*e.g.,* [13], [25]). 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 [25] 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.

Another popular approach for detecting IMAs is that of using safe pointers (sometimes referred to as fat pointers) [1], [7], [20], [21]. A smart pointer augments the pointer value with the upper and lower bounds of the memory object it points to. During the program execution, any derived pointers from the original fat pointer inherit the bounds information. For example, if a new pointer is obtained as a sum of an existing base pointer and an offset, the new pointer inherits the bounds metadata from that of the base pointer. During a memory accesses, the metadata of the fat pointer needs to be checked if the access is within bounds in order to verify that this a legal access.

Software-only implementations of these safe pointers need to instrument the program code not only to intercept allocation events that create pointers, but also to implement the meta-data propagation and checking mechanism. This result in high overheads – even recent schemes that analyze the code to eliminate redundant checks, such as SoftBound [20], still incur average overheads in excess of 50%. Another drawback of software-only IMA detection schemes is that they are typically not thread-safe – significant additional overheads would be needed to keep pointers and their bounds consistent (i.e. pointers and their bounds must be updated atomically, *e.g.,* using a critical section or a transaction).

Hardware-assisted solutions for safe pointers, such as HardBound [7], alleviate the cost of storing and propagating the pointer bounds meta-data, using bounds checking in hardware and with a number of optimizations aimed at efficiently storing and propagating this meta-data to and from memory. Additionally, hardware techniques such as HardBound can maintain data/meta-data consistency at a relatively low cost – both HardBound and our new DIFT-based technique store meta-data (taints for our technique, bounds for HardBound) in a packed array, decoupled from its data, and (as described in MemTracker [31] and FlexiTaint [30]) can leverage the existing instruction replay mechanisms in modern out-of-order processors to keep their data and meta-data consistent.

Even though safe pointers offer strong detection guarantees for spatial IMAs (access to a different region of memory region), they cannot efficiently detect temporal memory errors (access to memory the pointer is no longer allowed to access). For example, an access through a dangling pointer is undetectable with bounds information alone, and would require additional expensive checks (*e.g.,* a lookup in a hash table of still-allocated memory regions). In contrast, our DIFT-based IMA detector can detect both spatial and temporal IMAs with equal probability, because the association between a pointer and the memory it points to is broken when the taint of the memory location changes – it is reset on deallocation and then set to a different (with a probability that increases in proportion to the number of available taint marks) when it is reused for another allocation.

In addition to software-based approaches and hardware-assisted safe pointer approaches, there have also been numerous other proposals for hardware-assisted detection of particular classes of IMAs. In SafeMem [22], existing memory-error correcting codes were used to detect accesses to unallocated memory. MemTracker [31] 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). Our DIFT-based technique is more general than these approaches, in that it targets all spatial and temporal IMAs, *e.g.,* those that involve accesses to allocated and initialized memory. For example, SafeMem and MemTracker can only allow (or disallow) all reads from a location, but cannot prevent reads using a given pointer while allowing reads using another one.

Other related work includes DIFT support in hardware [2], [6], [19], [29]. These schemes 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, and are complementary to our technique in that they could, with some additional design effort, help amortize the cost of taint propagation and checking hardware.

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 pointer and the memory it accesses 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 previous dynamic techniques for IMA detection. First, it is highly effective: it was able to identify all known IMAs in the real programs we used in our evaluation. Second, it is amenable to a hardware-assisted implementation: detailed simulations of a hardware-based implementation show average performance overheads below 10%, even in multi-threaded applications. Finally, unlike previous IMA detection techniques, our technique can easily be tuned to achieve different tradeoffs between performance overhead and probability of detecting IMAs.

Acknowledgments

This work was supported in part by NSF awards CCF-0541080, CCF-0916605, and CCF-0964647 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.

References

[1] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient Detection of all Pointer and Array Access Errors," *SIGPLAN Not.*, vol. 29, pp. 290–301, 1994.

[2] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring," in *Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[3] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective Memory Protection Using Dynamic Tainting," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007.

[4] 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*, 2007.

[5] cplusplus.com, "Malloc Example," June 2007, http://www.cplusplus.com/reference/clibrary/cstdlib/malloc.html.

[6] J. R. Crandall and F. T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[7] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hard-Bound: Architectural Support for Spatial Safety of the C Programming Language," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[8] D. Dhurjati and V. Adve, "Backwards-Compatible Array Bounds Checking for C With Very Low Overhead," in *Proceeding of the 28th International Conference on Software Engineering*, 2006.

[9] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing Alias Analysis For Weakly Typed Languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[10] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*.

[11] J. S. Fenton, "Memoryless Subsystems," *The Computer Journal*, vol. 17, no. 2, pp. 143–147, 1974.

[12] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A System and Language for Building System-Specific, Static Analyses," *SIGPLAN Not.*, vol. 37, no. 5, pp. 69–82, 2002.

[13] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," in *Proceedings of the USENIX Winter 1992 Technical Conference*.

[14] D. L. Heine and M. S. Lam, "A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector," *SIGPLAN Not.*, vol. 38, no. 5, pp. 168–181, 2003.

[15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," in *USENIX Annual Technical Conference*, 2002.

[16] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*.

[17] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou., "BugBench: Benchmarks for Evaluating Bug Detection Tools." in *Proc. of the Work. on the Evaluation of Software Defect Detection Tools*, 2005.

[18] 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*.

[19] M.Dalton, H.Kannan, C.Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," in *International Symposium on Computer Architecture*, 2007.

[20] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[21] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, 2005.

[22] F. Qin, S. Lu, and Y. Zhou, "SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs," in *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, 2005.

[23] 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.

[24] O. Ruwase and M. S. Lam, "A Practical Dynamic Buffer Overflow Detector," in *NDSS*, 2004.

[25] J. Seward and N. Nethercote, "Using Valgrind to Detect Undefined Value Errors with Bit-Precision," in *Proceedings of the USENIX Annual Technical Conference*, 2005.

[26] Standard Performance Evaluation Corporation, "http://www.spec.org," 2004.

[27] S.Woo, M.Ohara, E.Torrie, J.Singh, A.Gupta, "The Splash2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture*, 1995.

[28] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi, "Cacti 5.0," *http://www.hpl.hp.com/techreports/2007/HPL-2007-167.html*, 2007.

[29] 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 *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[30] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation," in *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, 2008.

[31] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[32] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," in *NDSS*, 2003.

[33] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors," *SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 327–336, 2003.

[34] 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*, vol. 29, no. 6, pp. 117–126, 2004.

Ioannis Doudalis received his Diploma degree in Electrical and Computer Engineering from the National Technical University of Athens in 2006 and his M.S. degree in Computer Science from the Georgia Institute of Technology in 2009. He is currently pursuing a Ph.D. degree in the College of Computing at the Georgia Institute of Technology. His research interests are in computer architecture with an emphasis on security and debugging. He is a student member of IEEE.

James Clause is an instructor at the University of Delaware and a doctoral candidate in Computer Science at the Georgia Institute of Technology. In 2005 he received the M.S. degree in Computer Science from the University of Pittsburgh. His area of research is software engineering with an emphasis on program analysis and debugging. He is a member of the IEEE.

Guru Venkataramani received his Ph.D. degree from Georgia Institute of Technology in 2009. He is currently an assistant professor in the Department of Electrical and Computer Engineering, George Washington University, Washington DC. His research interests are in computer architecture with focus on programmability, security and debugging. He is a recipient of ORAU's 2010 Ralph E. Powe Junior Faculty Enhancement Award. He is a member of IEEE.

Milos Prvulovic is an Associate Professor in the College of Computing at the Georgia Institute of Technology, which he joined in 2003. He received his B.Eng. degree in Electrical Engineering from the University of Belgrade in 1998, and his M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 2001 and 2003, respectively. His research interests are in computer architecture, especially hardware support for reliability, programmability, and software debugging. He is a past recipient of the NSF CAREER award, a senior member of both ACM and IEEE, and is the Secretary/Treasurer of the ACM SIGMICRO.

Alessandro Orso is an Associate Professor in the College of Computing at the Georgia Institute of Technology. He received his M.S. degree in Electrical Engineering (1995) and his Ph.D. in Computer Science (1999) from Politecnico di Milano, Italy. From March 2000, he has been at Georgia Tech, first as a research faculty and now as an Associate Professor. His area of research is software engineering, with emphasis on software testing and static and dynamic program analysis. His interests include the development of techniques and tools for improving software reliability, security, and trustworthiness, and the validation of such techniques on real systems. Dr. Orso is a member of the ACM and the IEEE Computer Society.