Dynamic Information Flow Tracking on Multicores

Vijay Nagarajan [1] Ho-Seop Kim [2] Youfeng Wu [2] Rajiv Gupta [1]

[1] University of California, Riverside {vijay,gupta}@cs.ucr.edu

[2] Intel Corporation {ho-seop.kim,youfeng.wu}@intel.com

ABSTRACT

Dynamic Information Flow Tracking (DIFT) is a promising technique for detecting software attacks. Due to the computationally intensive nature of the technique, prior efficient implementations [21, 6] rely on specialized hardware support whose only purpose is to enable DIFT. Alternatively, prior software implementations are either too slow [17, 15] resulting in execution time increases as much as four fold for SPEC integer programs or they are not transparent [31] requiring source code modifications. In this paper, we propose the use of chip multiprocessors (CMP) to perform DIFT transparently and efficiently. We spawn a helper thread that is scheduled on a separate core and is only responsible for performing information flow tracking operations. This entails the communication of registers and flags between the main and helper threads. We explore software (shared memory) and hardware (dedicated interconnect) approaches to enable this communication. Finally, we propose a novel application of the DIFT infrastructure where, in addition to the detection of the software attack, DIFT assists in the process of identifying the cause of the bug in the code that enabled the exploit in the first place. We conducted detailed simulations to evaluate the overhead for performing DIFT and found that to be 48% for SPEC integer programs.

1. INTRODUCTION

Software attacks have become increasingly prevalent. US CERT Statistics [33] show that the number of attacks have increased rapidly over the years. At the same time it has become increasingly costly for businesses to deal with worms and viruses. FBI computer crime survey estimates that US businesses spent a total of $67.2 billion in the year 2005 to deal with computer crime, a significant proportion of which was spent in dealing with software attacks (worms and viruses).

Dynamic Information flow tracking (DIFT) [21, 15, 6, 17, 31] is a promising technique for providing security against malicious software attacks. The basic idea hinges on the fact that an important avenue through which an attacker compromises the system is through input channels. This is a direct consequence of most of the vulnerabilities being *input validation* errors. In fact, 72% of the total vulnerabilities discovered in the year 2006 are attributed to a lack of (proper) input validation [34]. Note that most of the memory errors including buffer overflow, boundary condition and format string errors fall into this category.

The main principle of DIFT is as follows. A set of input channels, for example network inputs, are considered insecure. The flow of information from these inputs is tracked and those values that are data dependent on such inputs are in turn marked tainted. Potential attacks are detected upon the suspicious use of such tainted values.

There have been prior implementations of DIFT both in hardware and software. Hardware-based approaches [21, 6] employ specialized hardware to perform DIFT operations concurrently along with the original operations. While this approach incurs little performance overhead, it requires substantial hardware changes. Furthermore these hardware changes are specialized, in that, these are required only for enabling DIFT.

On the contrary, software-based information flow tracking [31, 17, 15] while not requiring specialized hardware for performing DIFT, suffers from several deficiencies themselves. Recent research [31] showed how software based taint analysis can be used to deal with a wide variety of software attacks. But the above approach requires source code modifications, making it unsuitable for proprietary software and library functions for which source code may not be available. Additionally, static instrumentation schemes (source code level or binary level) suffer from the possibility of a sophisticated attacker bypassing the tracking code at runtime and annulling the benefits of performing tracking [7]. Finally, tracking for dynamically generated code and self modifying code cannot be handled via static instrumentation. Information flow tracking via dynamic binary instrumentation [31, 15] eliminates the above problems but suffers from a huge performance overhead. Even the most efficient implementation slows down the SPEC integer programs by a factor of four [17]. Thus, the motivation for this paper is the *lack of a transparent and efficient information flow tracking mechanism that does not use specialized hardware support.*

Our main idea is to use a separate core of a chip multiprocessor (CMP), which is becoming increasingly ubiquitous, to perform DIFT operations transparently and efficiently. In our DIFT framework, a dynamic binary translator scans the program image and transparently generates a helper thread that performs DIFT for the main thread. Here, the main thread refers to the original program. This helper thread is scheduled on a separate core and tracks the information flow of its corresponding main thread concurrently. Each register of the helper thread holds the taint-value for the corresponding register in the main thread while the taint-value for each byte in the original program is stored in a linear fashion in an additionally allocated memory space. Whenever the use of a value violates the specified security policy, the helper

thread raises an exception and interrupts the main thread.

Having described our design in a nutshell, we now make some observations to illustrate some of the design issues. First, the helper thread should, as far as possible, execute only the instructions required for DIFT. Otherwise, the helper thread may run much slower than main thread [17]. Second, the helper thread should exactly follow the control flow of the main thread. Only then can it perform tracking for all the executed instructions. Third, several instructions, for example register indirect jumps, use register values as its operands. As the helper thread executes only DIFT instructions it does not have access to these register values; neither does it have the necessary flag values to exactly follow the control flow of the main thread. Thus an important consequence of this design is that the main thread needs to continuously *communicate registers and flags* to the helper thread at a fine grained level. We initially explored communication of these values through shared memory. But we found that this caused high overhead since most of the time was spent in executing the instructions for enabling this communication. To alleviate this problem, we utilize a dedicated interconnect in the form of a hardware FIFO buffer for the communication between the cores. Moreover this FIFO buffer is software controlled through two new instructions *enqueue*, that pushes values into the FIFO, and *dequeue* which pops values from the FIFO. Although this interconnect feature is not available in current CMPs, a number of recent studies [18, 16, 19, 26] have been advocating use of such queues for intra- chip core-to-core communications for such wide ranging applications as automatic thread extraction and debugging. Using this *generalized hardware support*, we found that DIFT can be successfully performed with only 48% slowdown, which is a significant improvement over current software based schemes.

1.1 Bug Location

There has also been significant recent research on the subject of recovery from software attacks [20, 25]. The primary goal is to ensure the *availability* of the server, despite it being a target of a software attack. The key idea is to have techniques to save the machine state (including memory contents) at periodic intervals. In the face of an attack, instead of terminating the application under attack, a prior uncorrupted machine state is restored. It is worth noting that this is only a temporary solution, since this does not prevent the same attack from being carried out again [20]. Clearly, the best possible recovery scheme is to apply a software patch so that the bug in the code that causes the vulnerability is removed. To do this, we first need to locate the bug in the code that causes the vulnerability. In prior work [11] architectural support has been provided to store the necessary trace information, so as to enable deterministic replay debugging. In this work, we go a step further in this direction and leverage the DIFT infrastructure to assist us in the process of bug location. The basic idea is quite simple. Instead of propagating the boolean taint values, we propagate *PC values*, where PC refers to program counter. Later, we show that the CMP architecture is able to tolerate the extra taint memory overhead incurred gracefully. As usual, a zero indicates untainted data and a non-zero (PC) value represents tainted information. At any instant, the PC value corresponding to a tainted location is the PC of the *most recent* instruction that wrote to the location. When an attack is detected, the PC taint value of the tainted memory location (or register) gives us additional information, namely the most recent instruction (statement) that modified it. This information can be vital in identifying the source of the bug and our preliminary experiments confirm that in most cases this directly points to the statement that is the root cause of the bug.

1.2 An Example

To illustrate the key aspects of performing DIFT in a multicore, let us consider the example in Fig. 1. This expository example will illustrate some of the design issues involved in this work which will be considered in detail in the following sections. The first column shows the source code of the original program. The second shows the machine instructions that are executed by the main thread. The third column shows those that are executed by the helper thread for DIFT while the last column shows the instructions executed (shows only changes) by the helper thread with support for bug location.

Original Code. The function *fun* calls three different functions based on the variable *choice*, passing the character string *buf* as a parameter to the function call. The string is read from the file into a global using the (safe) library function *fgets*. But the unsafe *strcpy* is used to copy the global string into the stack. This overflows the buffer allocated to *buf*, when the size of the read string is greater than 10 bytes, thereby overwriting the stack variable *choice*, that has been *spilled* on to the stack. Since the variable *choice* is later used as an index into the jump table (due to the switch statement), overwriting this can potentially lead to a *return-to-libc* attack.

Helper Thread for DIFT. Now let us consider the instructions in the main and the helper threads. As mentioned previously, the registers in the helper thread contain the taint-values of corresponding registers of the main thread and taint-values of corresponding memory locations are assumed to be offset by the value *off*. Initially, let us assume that the value *choice* resides in the *eax* register. Furthermore, let us assume that the variable *choice* is not tainted, which means the initial value of *eax* of helper thread is 0.

- **Statement 1.** This corresponds to a bounds checking operation; the main thread terminates if the value of *eax* exceeds 3. To ensure that helper thread follows the control flow of the main thread, the branch outcome needs to be communicated from the main to the helper thread. Recall that the helper thread does not have access to the register (and flag) values of the main thread.

- **Statement 2.** This corresponds to the spill code, where the value of *eax* is pushed on to the stack. Accordingly, the helper thread moves the value of its *eax*, which has the taint-value corresponding to the main thread's *eax*, into the appropriate taint-memory location corresponding to the main thread's stack. To enable the computation of the address of this taint memory location, the main thread communicates its *esp* to the helper thread. The semantics for Statement 5 are similar.

- **Statement 3.** This corresponds to the call to *fgets*, where the input is obtained and copied into the string

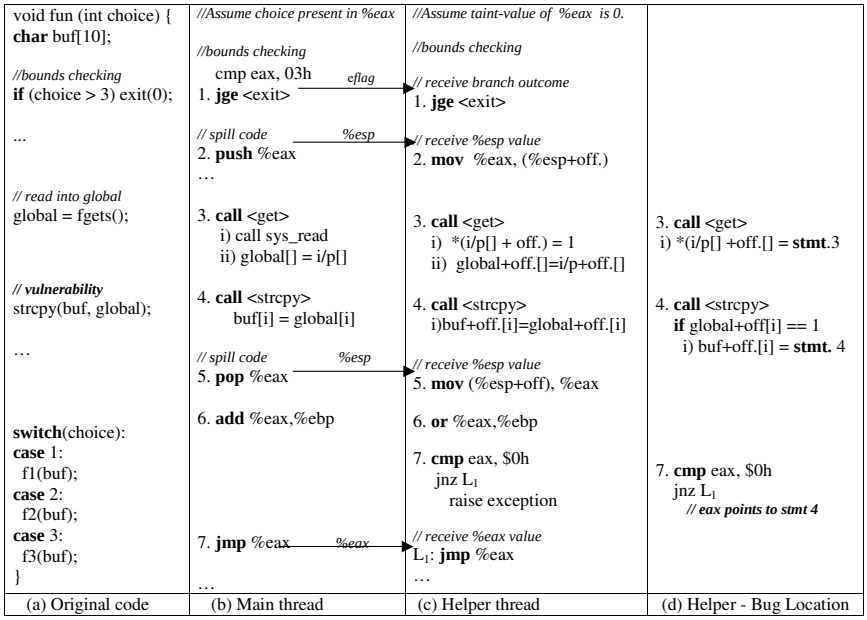

Figure 1: DIFT/Bug Location using helper threads

global. The helper thread marks the taint-value corresponding to input buffer as *high* and the taint-values are faithfully copied when the input buffer is copied into *global* in the main thread.

- **Statements 4.** This involves a call to *strcpy*. As we can see from the third column, the helper thread performs memory movement operations of the taint-values corresponding to original memory movement.
- **Statement 6.** This involves a computation operation for which helper thread performs a logical *or* to reflect the fact that the target should be tainted if any of the source operands are tainted.
- **Statement 7.** This corresponds to a register indirect jump based on *eax*. To check for a control flow attack, the helper thread checks the corresponding taint-value of *eax*; if it is tainted, it raises an exception. If it is not tainted, the helper thread should follow the control flow of the main thread. To enable this, the main thread communicates the value of *eax*.

Had there been a buffer overflow in statement 4, this would have tainted the spilled value. When the spilled value is subsequently popped into *eax* in statement 5, the corresponding *eax* of the helper thread would also be tainted. Thus, an exception would have been raised in statement 7, preventing the attack.

Helper Thread for Bug Location The last column provides the code for the helper thread with bug location support. The only changes to the code are for statements 3 and 4. For the *gets* statement, instead of initializing the taint-values with a value of 1, we instead initialize it with the PC of source code statement that caused the input (statement 3). Similarly for *strcpy*, we perform the following. If the source is not tainted, we propagate the taint-value of 0 to the target. But if the source is tainted, we update the taint-value of the target of the copy with the PC that caused the copy (statement 4). This would enable us to detect the cause of the error (statement 4) when the attack is detected in statement 7.

2. DIFT USING ADDITIONAL CORE

In this section, we discuss the detailed design and implementation of DIFT using multicore. Initially we give a brief overview of the creation of the helper thread and explain how appropriate code is generated for the two threads. Then we move on to the software and hardware techniques for communication between the threads. We discuss the hardware FIFO buffer in detail and introduce optimizations to reduce the communication cost. Then we move on to taint-value (register and memory) management in the helper thread.

2.1 Thread Management

We used a dynamic binary translator (DBT), to generate DIFT code and manage the main and helper threads. In a DBT based implementation, the DBT automatically loads the original program code into memory and initializes the processor execution context in the beginning. It also uses a code cache to store the translated code so that the original code is translated once and executed multiple times in order to improve the overall program execution performance. We used Intel's *StarDBT* [1] infrastructure as the dynamic translation engine for our work. We modified StarDBT in the following way. Initially, we create two StarDBT threads, both of which translate the original code. We modified the translation engine such that it outputs different translations based on the the current thread-id. In the main thread, the translator is made to faithfully output the original code verbatim additionally generating code that communicates the required registers and flags. In the helper thread, the translator is made to generate the required code for performing DIFT operations in addition to instructions for communicating with the main thread.

2.2 Inter-thread Communication

Since the helper thread performs only the DIFT operations, it does not have access to the values computed by the main thread. As we saw in the example, the main thread communicates the appropriate register/flag values for branching instructions and indirect memory reference instructions.

2.2.1 Communication through Shared Memory

Communication can be accomplished through shared memory employing a software queue abstraction. There are two sources of slowdown when this type of communication is used. First, the extra instructions introduced to accomplish communication. Extra instructions need to be executed both by main and helper threads to send and receive data from the queue. This is also known as *intra-thread latency* in prior work [18]. Secondly, the actual communication latency, known as *inter-thread latency* [18], can be substantial due to a large number of cache misses inherent in producer-consumer communication patterns. This is accentuated by the fact that most multicore processors do not share the L1 cache and all *receive* operations need to necessarily go to the L2 cache.

2.2.2 Hardware based Communication

As we discussed in the previous section software based communication has two disadvantages: increased inter-thread and intra-thread latencies. Each of these can be greatly mitigated by the use of hardware support for communication between cores and by the addition of ISA support for using single instructions for send and receive operations. The *enqueue* and the *dequeue* instructions, each take a register as an operand; while the former puts the register contents into the queue, the latter pops it from the queue into the register. We model the queue as a FIFO buffer of fixed size. Furthermore, the dequeue instruction blocks the processor if the queue is empty and the enqueue instruction blocks if the queue is full. Thus, there is no requirement to explicitly perform any other synchronization operation. The realization of the such an interconnect structure is discussed in detail in prior work [26]. More recently, a light-weight memory enhancement [18] has been proposed which can also be utilized for this work. This light weight memory enhancement obviates the need for a direct hardware structure that connects the cores and uses the memory system to for achieving the communication.

We believe that the cost of utilizing hardware support in the form of changes to ISA or assumption of a communication queue is minimal and more importantly general compared to the changes that need to be made for incorporating DIFT in hardware. Also hardware support for communication between cores is currently an active research area and has several other applications including software pipelining [16] and debugging [19].

2.2.3 Optimizations to Reduce Communication

The communication bandwidth required will play an important role in the actual complexity of the design of the communication system. We conducted some simple experiments with SPEC integer benchmarks to measure the bandwidth required. While communication for register-indirect jumps required a bandwidth of 1 byte every 100 instructions, communication of eflag registers and communication for register-indirect memory addressing required a bandwidth of 50 bytes and 150 bytes every 100 instructions. So we looked at optimizing the costs for the last two.

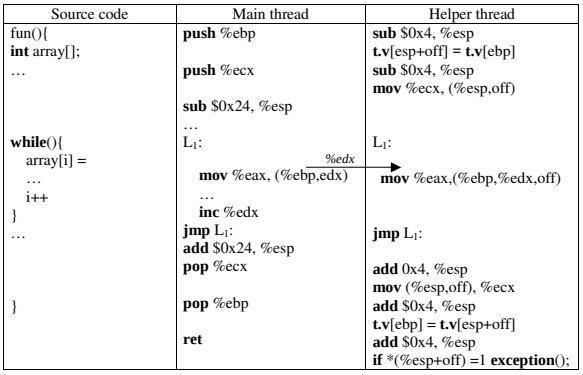

Figure 2: Register indirect Optimization

OPT1: Register Indirect Memory addresses. The *x86* instruction set allows memory addressing with the base and index registers. We observe that several memory operations have the *ebp* register as the base register, due to the use of local variables in functions. This corresponds to the *mov* instruction in Fig. 2 moving values into the stack allocated variable, *array*. Similarly, all the stack operations use the *esp* register for addressing, albeit implicitly. At the same time, for compiler generated code, DIFT operations involving these registers are not frequent. For instance, consider the *add* and *sub* instructions that change the stack pointer in Fig. 2. Obviously no DIFT operations are needed for these. Thus we make a design decision of maintaining the *actual* values of stack pointer and the base pointer in the helper thread. By doing these, we avoid the communication of these registers. This is seen in the *mov* instruction where only the index register *edx* is communicated. Similarly this saves us from communicating the stackpointer, *esp*, for all push/pop instructions. Instead the stack pointer of the helper is updated to reflect the push/pop. It is important to note that we store the taint-values of *esp* and *ebp* in memory, so that we can perform DIFT operations for these registers if the need arises. This is illustrated in the *pop ebp* instruction in the example. The value of *ebp* may have been corrupted while it was in memory. So it is necessary to perform DIFT and accordingly the taint-value (t.v) of *ebp* in memory is updated.

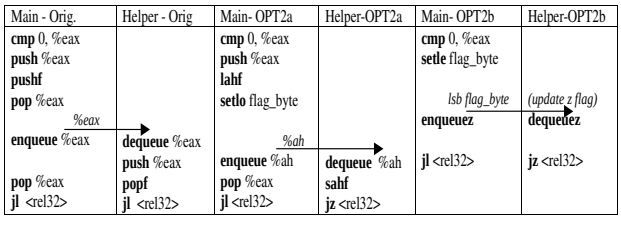

Figure 3: EFLAGS Optimization

OPT2a: Software EFLAGS Optimization. In the naive scheme, to communicate the control flow the EFLAGS regis-

ter is communicated fully, as shown in the first two columns of Fig.3. But to perform control flow transfers only the *status* flags namely the sign, carry, auxiliary carry, parity, zero and overflow flags need to be communicated. All of the status flags, except the overflow flag is contained in the first byte of the eflags registers. Hence, using the *lahf* and the *sahf* instructions, we can communicate all flags except the overflow flag. The overflow flag is communicated via shared memory separately through the *seto* instruction as in [3]. This optimization has two advantages. First, communication bandwidth is reduced, since only a byte of flag information is communicated as opposed to a word. Second, it also improves the performance as it eliminates the need of executing a *popf*, which takes tens of cycles to execute [3].

OPT2b: Hardware EFLAGS Optimization. Intuitively, we just need to communicate one bit information for branches; whether the branch is taken or not-taken. To handle this we could have expanded the enqueue instruction to specifically communicate the exact flag bit. Note that due to inorder nature of the queuing, sending multiple sized data (a word or a bit) is not a problem. But since, several branch instructions use multiple flag bits for deciding the branch outcome, we need to again use multiple enqueues/dequeues for branch instructions. For example, the *jl* instruction takes the branch if $SF = OF$, requiring us to communicate both of these flag bits. Here, we make use of the *setcc* instruction to write the taken/not taken information in a specific allocated memory location ($flag - byte$). Then we expand the enqueue/dequeue semantics to include the *enqueuez* and *dequeuez* instructions. The *enqueuez* instruction reads the lsb of this memory location ($flag - byte$) and pushes it into the queue. The *dequeuez* instruction pops a bit from the queue and sets the *zero* flag. We then use the *jz* instruction, irrespective of the original branch instruction, in the helper thread. This optimization is illustrated in the last two columns of Fig. 3.

2.3 Fail-Safety

When an attack is detected, it is important that the main thread is notified as soon as possible. Thus, if the helper thread runs slower compared to the main thread, then the exception may be reported much later, presenting a time window during which the program can be compromised. The size of this window is not expected to be large because of two reasons. First, the dynamic number of instructions executed in the helper thread are almost similar to the those executed by the main thread. Moreover, the types of instructions for performing DIFT are similar, sometimes even simpler, compared to the original instructions and several instructions in the main thread (eq. increment) do not require DIFT to be performed. For example, the original move instructions have corresponding moves. But the costly multiply instructions are replaced by the cheap *or* instructions, since we just need to propagate the taint-values. Furthermore, several instructions in the main thread do not require DIFT performed. (eg. increment instructions). Second, even if the helper thread trails the main thread, it is not allowed to do so indefinitely. It is restricted by the queue size, since the main thread stalls if the queue happens to become full. Thus we can control the window size indirectly by changing the size of the queue.

Nevertheless, it may be desirable to have the *fail safety* property - a property which guarantees that the main thread is notified before it deviates into a malicious control flow. We accomplish this using *two way communication*. The helper thread sends a bit of information before every vulnerable control transfer instruction. This is shown in the middle two columns of Fig. 4 where the helper-fail-stop thread enqueues the taken/not-taken information. At the same time, the main thread performs an *dequeuez* instruction before every register indirect jump, raising an exception if the value received is a high bit. Here, it is important to note that the fail-safety property needs a bidirectional queuing system, which results in additional hardware complexity. It also comes at the expense of lower performance of DIFT, because the main thread should repeatedly synchronize with the helper thread before every vulnerable control transfer instruction, including register indirect jumps,calls and returns.

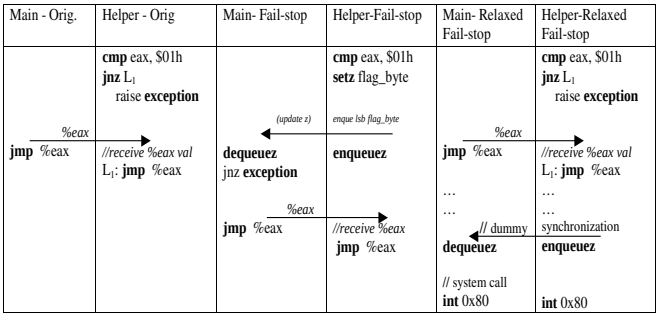

Figure 4: Ensuring fail safety

2.3.1 Relaxed Fail-Safety

To reduce the performance penalty of ensuring fail safety, we describe a relaxed fail safety implementation inspired from *system call monitoring*. System call monitoring [27, 29] is a commonly used technique for intrusion detection. The basic idea is that a compromised application can cause real damage only through system calls and thus monitoring system calls is a good technique to detect intrusions. Thus, instead of the main thread synchronizing with the helper thread before every vulnerable control transfer instruction, the main thread synchronizes with the helper thread only before system calls. Consequently, as shown in the final two columns of Fig. 4 the helper thread performs an *enqueuez* instruction before every system call and the main thread performs a dummy *dequeuez* instruction before every system call purely for the purpose of synchronization, in that the dequeued value is not really used by the main thread. However, this ensures that the main thread will proceed executing the system call, if and only if the helper thread reaches the corresponding execution point. Had there been any malicious control flow in the original thread, the helper thread would have detected this and raised an exception before the main thread gets a chance to execute the subsequent malicious system call. Since the number of system calls executed in a program is typically much lesser than the number of register indirect control transfers and returns, the performance penalty of ensuring relaxed failsafety is expected to be lesser. In fact, in our experiments we found that we were able to implement relaxed fail safety with negligible overhead even if we performed the synchronization before

system calls using software shared memory. Thus we were able to enforce relaxed fail safety with negligible overhead with only unidirectional queue support.

2.4 Taint-values Management

In this section, we describe how the taint values, including those for registers and memory, are managed in the helper thread. As we discussed previously, we store the taint-values for the registers in the main thread (except *esp* and *ebp*) in the corresponding registers in the helper thread. Similarly we store taint-values of memory locations of the original program in a linear translated memory, which is specially allocated. Similar to previous schemes [21, 17] we considered 1 bit taint values for every byte of memory. But this means for every memory access we need to perform computations, in the order of ten to fifteen instructions for locating the exact bit from the memory and fetch/update that particular bit. The fact that memory operations are very frequent in *x86* programming model, compounds the problem. To eliminate this problem, we use 2 bits as taint values as proposed in earlier work [31]. The main advantage being *word* operations in the original program are converted to *byte* operations as opposed to the complex bit making and shifting operations.

We also experiment with allocation of a taint-value of one word for every word of original memory for the purpose of implementing the bug location via tracking. Note that this is required in order to track the PC values for bug location. Obviously we need to pay a price of 100% memory overhead. But with the advent of 64 bit architectures, this additional virtual memory requirement may not be a huge problem. Moreover, since multicores have a separate L1 cache, the cache effects of the additional memory requirement may be mitigated. As our experiments later show, we find this is the case for the SPEC integer benchmarks.

3. USING DIFT FOR BUG LOCATION

In this section, we discuss how the DIFT infrastructure is effectively used to locate the bug that caused the vulnerability in the first place. The general idea is to flow additional information during tracking, so that at the time of detection of the attack, we have useful information that can ultimately lead to the location of bug. The first step in any attack is the

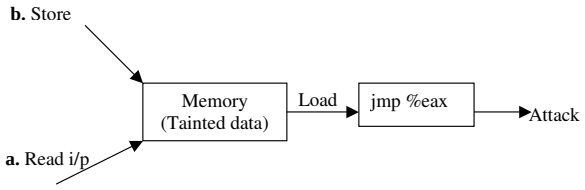

Figure 5: Control Data Attack

corruption of memory with attacker crafted values. This is accomplished using vulnerabilities like buffer overflow, format string or integer overflow etc. The goal of this work is to exactly *pin-point the statement in the user program that causes this memory corruption.* This can be very useful in debugging and in several cases, this directly corresponds to the root cause of the bug. To accomplish this, intuitively, the following needs to be performed. Whenever there is a write to a memory location, we first need to keep track of the source code statement that causes the write. Then we need to be able to track this information as it is used in computations. Clearly, the pattern is very similar to operations involved in DIFT itself and thus the DIFT infrastructure can be used effectively for this purpose.

The steps of a control data attack are shown in Fig. 5. The two common ways through which data is written to the memory are either by calling the read/receive system call or by executing a store instruction. Let us first consider the read/receive system call, which is executed due to the use of input statements in the original code. Insecure statements like *gets* can cause a buffer overflow thereby corrupting memory. Ideally, we want to identify this statement when the attack is finally detected. To accomplish this, we incorporate the following minor change to DIFT. Instead of initializing the taint-value with a boolean high value, when there is a read, we update it with the *PC of the original statement* in the program that caused the read. Without any other changes to DIFT, at the time of the attack, the taint-value of the jump target will correspond to this original PC. (and hence the faulty statement in the program).

Memory corruption can also take place due to a store instruction. For example, memory corruption due to use of insecure statements like *strcpy* and *% n format string vulnerability* are all caused finally by store instructions that overwrite memory locations with tainted data. Note that the store instruction is not the cause of tainted data – it is still the user input which is the cause. We just want to remember the identity of store instruction, just in case it is part of a buggy statement (like strcpy) that is used by an attacker to cause memory corruption. To handle this case, we alter DIFT in the following manner. For every store instructions of the form *store reg, addr*, we check the taint-value of *reg*; if it is indeed tainted, we update the taint-value of *addr* with PC_{store}, so that we have a means of remembering this store instruction. Here, we are making an inherent assumption that the store that causes the memory corruption is the final store to it, before the attack is detected. This is almost always true, as control data is used as soon as it is loaded into the CPU.

4. EXPERIMENTAL EVALUATION

As we already mentioned we used the StarDBT dynamic binary translation infrastructure [1] to generate the required code for performing tracking and communication. We implemented the hardware queue part in the *Simics* full system simulator [12]. All the experiments using the hardware queue were performed in this simulator. The executables targeted the IA32 architecture. The target machine is a 32 bit dual core machine with a shared 4 way L2 cache of 512KB size and a hit latency of 10 cycles and separate 2 way L1 caches of size 16KB each with a hit latency of a cycle. The memory latency was assumed to be 150 cycles. The default queue size was assumed to be 512bytes with a communication latency of 10 cycles. Our assumption of queue latency is comparable with with prior work [16, 18] which used latencies ranging from 1 cycles to 10 cycles. Also, in our sensitivity analysis we study the effect of increased queue latencies.

We conducted experiments with several goals in mind. First and foremost, we wanted to study the execution time overhead of performing DIFT in multicores. Our baseline for this experiment is prior work with the most efficient implementation that is comparable [17]. As our technique hinges

on efficient communication between the cores, we also measured the communication bandwidth required for performing tracking and whether the proposed optimizations are able to reduce it. Here, we also evaluate the overhead of a shared memory based communication. Then we evaluated the overhead of enforcing both fail safety and relaxed fail safety. At the same time we want to make sure our approach does not produce false negatives /positives when tested with attack programs. Finally, we study the efficacy of our proposed debugging technique with real attacks.

4.1 Execution Overhead with h/w Queue

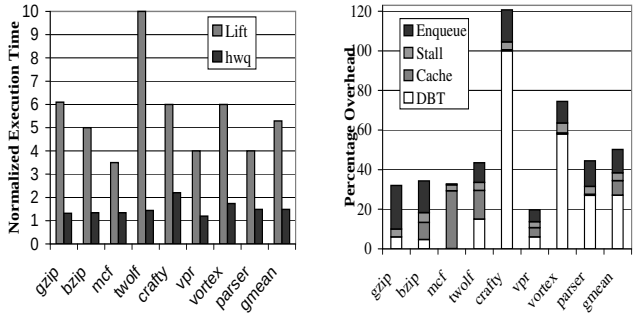

Figure 6: Execution Time Overhead and Break up of Overheads

We implemented a dedicated hardware queue in Simics, a full system simulator with ISA support for enqueuing and dequeuing from the queue, with configurable size and latency of the queue. We initially set the size of the queue to be 512 bytes and set the latency to be 10 cycles. The baseline for comparison is the LIFT technique. As we can see from Fig. 6, the dedicated hardware queue is able to greatly lower the overhead of DIFT from 4.6x to 48% overhead.

The second column shows the break up of the overhead. A significant part of the overhead is due to the use of a dynamic binary translator. But this is a cost we have to pay for transparent and secure implementation of DIFT. The other major cause of overhead is due the cache effects due to sharing of L2 cache by the main and the helper threads. This cost is greatest for a memory intensive application like mcf but generally pretty low. The final two overheads are due to the hardware queue themselves. The first overhead is because of the stalling of the main thread whenever the queue becomes full. As we see from the results, this overhead is very low confirming the fact that the helper thread is able to keep up with main thread. This is not surprising as the DIFT operations are computationally similar to the main threads operations. Finally, there is the overhead of executing the extra enqueue and dequeue instructions, which were modeled to have one cycle latency. (Note that this is different from the queue latency).

4.2 Sensitivity Analysis

The extra taint values used for tracking could potentially pollute the L2 cache since the L2 cache is a shared resource in a CMP. We especially want to determine the effect of this cache pollution in the debugging scenario where we allocate a word of taint value for each original word in memory. The results of cache sensitivity are presented in Fig. 7. The first bar (512) refers to the normalized execution time of the default configuration of 512 KB cache size. The second bar (512:100) refers to the same cache size but an allocation policy of a word of taint value for each word in memory. The third bar refers to the overhead of tracking with a 256KB cache size. Note that this overhead is normalized to the original application overhead with identical cache size. The last bar refers the 128KB cache size. As we can see, there is a very nominal degradation in performance as the cache size is decreases. Surprisingly even under the 100% taint allocation policy the degradation is only around 10% to 20%. Thus the *overhead of bug location using DIFT infrastructure is also nominal.*

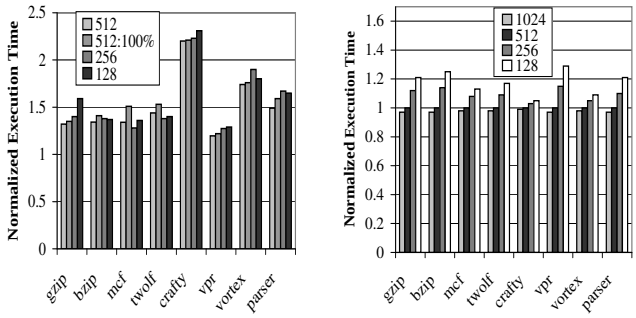

Figure 7: Sensitivity on L2 Cache Size and Queue Size

We conducted similar experiments by varying the queue size. We tried smaller queue sizes of 128 bytes and 256 bytes and also tried a larger queue size of 1024 bytes. Again we see almost a linear variation in the execution times and the variations are nominal.

We also conducted experiments by varying the queue latency to values to greater values namely 20 cycles and 30 cycles. The detailed results of this experiment are not presented because we could not observe a perceptible change in execution times as we varied the latency. This indicates that there is a constant lag between the leading and trailing threads, mainly because the trailing thread is able to keep up with leading thread. In some sense, this vindicates our approach of offloading tracking in another core. *This invariance to the queue latency makes it possible for us to use the light weight queues proposed in [18] instead of a direct hardware based structure.* It is worth noting that the above light weight scheme still uses ISA support in the form of *enqueue* and *dequeue* instructions. As we shall see in the next section, this ISA support is crucial for our performance, without which we might have to execute a lot of instructions to perform the enqueues and dequeues.

4.3 Communication Bandwidth and s/w Queue Overhead

The communication bandwidth required by our technique is an important parameter which potentially could affect the hardware complexity of the hardware queue. As we already covered in detail, the main thread should communicate memory addresses and branch outcomes to the helper thread, to enable it to performing DIFT. We measure the average bandwidth required for performing this communication. The unoptimized refers to the communication required for the baseline implementation. The average bandwidth, as we can see from Fig. 8 required is pretty high on the order of 3.1 bytes/cycle. OPT1 refers to our optimization in which we maintain values of the stack pointer and base pointer in

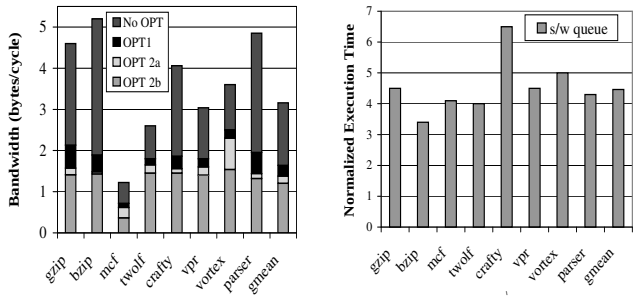

Figure 8: Communication Bandwidth and Software Queue Overhead

the helper thread to obviate their communication. As expected this optimization is very effective reducing the bandwidth requirement by 50% across all benchmarks. OPT2a refers to the optimization where we communicate only the first 8 bits of the EFLAG register and communicate the *overflow* flag separately. This is able to reduce the bandwidth by a further 20% and finally OPT 2b, the EFLAGS optimization which is a hardware based optimization reduces it further by 7%. Although the final bandwidth is reduced significantly through the above optimizations the bandwidth is still around 1.5 bytes per cycle. This means we are required to perform communication very frequently. Perform this communication through software translates into executing lots of additional instructions for maintaining the circular queue. To estimate the optimal performance of a software queue, we conducted a simple experiment so as to simulate the effects of the increases instruction count. In other words we ran just the single thread simulating the extra instructions required for the software queue. Even in this case, we observe from Fig. 8 that the overhead is 4.4x, though marginally lesser than overhead of LIFT [17], is still extremely high for practical deployment. This basically serves as additional motivation for the requirement of ISA support in the form of enqueue and dequeue instructions for handling the bandwidth needs of DIFT.

4.4 Fail-Safety

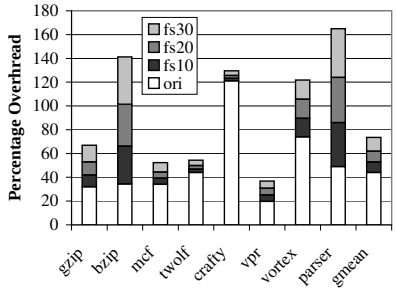

Figure 9: Overhead of enforcing Fail Safety

In this experiment, we evaluate the impact of the enforcing *fail safety* where the leading thread is forced to verify whether all indirect jumps are benign. Clearly this would cause additional; overhead since the leading thread has to necessarily stall. For this experiment we varied the communication latencies from the default value of 10 cycles through 20 cycles and 30 cycles. As we see from Fig. 9, the fail safety condition causes an extra overhead of about 10% for the default case, which is tolerable. We observe an almost linear increase in the performance with increase in the communication latency. This is in contrast to no observed variation when fail safety was not considered. This is because with the fail safety requirement the main thread is forced to wait for acknowledgment and this wait is proportional to the communication latency.

We also evaluated the impact of enforcing *relaxed fail safety* in which the leading thread is made to synchronize with the trailing threads before executing a system call. For the SPEC benchmarks, we found that the main thread needed to synchronize with the helper thread very infrequently (less than 1000 times per billions of instructions executed). This resulted in practically no additional overhead for enforcing relaxed fail safety. We repeated the experiment, now using shared memory for synchronization before system calls and found that this still resulted in negligible overhead (Performance overheads very small and hence not shown). It is worth noting that relaxed fail safety could thus be enforced with just unidirectional queuing support between processors.

4.5 Attack Detection

In this experiment, we wanted to test if our implementation is able to correctly identify attacks without false negatives or positives. A false negative is caused when the technique fails to detect a security attack and a false positive is caused when the technique falsely raises an exception when there is no attack. For these experiments we used a version of multicore DIFT which provides protection against control data attacks.

Table 1: Attack Detection False Negatives/Positives

Program	Vulnerabiltites	Exception Raised
Attack Benchmarks [30]	return address function pointer long jump buffer	Yes
ncompress4.2.4	return address	Yes
Polymorph0.4	return address	Yes
SPEC	None	No

We tested our multicore DIFT with the 18 attack benchmarks [30]. These synthetic benchmarks exercise various types of buffer overflows including return address, long jump buffer and function pointer, and have been commonly used in prior art to test the efficacy of attack detection systems. As shown in Table 1, our multicore DIFT implementation was able to correctly raise an exception for each of the attacks. We also tested our system with two real world vulnerabilities namely *ncompress4.2.4* and *polymorph0.4* [32], each of which have buffer overflow (return address) vulnerabilities. We first performed a remote code injection attack in the latter two programs using malicious inputs that take advantage of the vulnerabilities. We then performed the attack with the multicore DIFT running in the background. Here also, multicore DIFT was able to correctly identify the attack. To test for false positives, we ran the SPEC integer benchmarks, which do not have any known vulnerabilities. For this experiment we considered file inputs to be tainted and found that there were no exceptions reported. This is the expected behavior, since the SPEC integer programs do not have any known vulnerabilities.

4.6 DIFT Assisted Bug Location

In this experiment, we wanted to test the efficacy of our DIFT infrastructure for locating the bugs that caused the security vulnerability. We applied our technique to two real world programs namely stack buffer overflow vulnerabilities in *ncompress4.2.4* and *polymorph0.4.*. In *ncompress4.2.4*,

```
File: compress42.c
void  comprexx (char **fileptr)
{
        int    fdin;
        int    fdout;
        char  tempname[MAXPATHLEN];

        // buffer overflow happens here
        strcpy (tempname, *fileptr);
        …
        …
        …
       // attack detected here
       // taint PC value points to strcpy
       return;
}
```

Figure 10: Bug location for ncompress program

a stack based buffer overflow is caused when a file name of length greater than 1024 characters is used as input. The actual vulnerability is illustrated in Fig. 10 which shows the *comprexx* function that contains the vulnerability. The vulnerability arises due to the improper use of the *strcpy* function. Since the local variable, *tempname* is allocated only 1024 characters (the value of *MAXPATHLEN* is 1024), this local variable is overflowed for inputs greater than 1024. We carefully craft such an input so as to corrupt the return address of the *comprexx* function.

We then executed *ncompress* using the above malicious input, with our multicore DIFT with bug location support turned on. Recall that with bug location support turned on, we track PC values instead of 1 bit taint values. As expected, our technique is able to detect the attack when the control returns from *comprexx*. Furthermore, we found that the propagated taint PC value for the return address corresponds to the PC of the *strcpy* instruction inside the *comprexx* function. Thus, our technique is able to successfully identify the buffer overflow causing statement, which in this case, was the statement that is the root cause of the security vulnerability. We obtain similar results for *polymorph0.4*, which also has a similar vulnerability.

5. RELATED WORK

There has been a lot of prior work done in the area of information flow. They can be coarsely categorized into two parts: information flow tracking for *confidentiality* [22, 13, 8, 9] and integrity [21, 15, 17, 31]. The former tracks information flow, mostly statically, to make sure that a program does not inadvertently transfer information from secure variable into insecure variables. Fenton's Data Mark Machine[9] was one of the earliest systems that used information flow tracking to enforce security policies. RIFLE [22] is a more recent proposal that implemented information flow tracking policies in hardware. The latter techniques can be thought as a dual, and makes uses of information flow methods to ensure that untrusted external inputs do not exert a malicious influence on program's execution. While in the former information flow due to control flow is taken seriously, almost all the techniques in the latter neglect flows due to control flow. This is based on the fact that it is considered hard to construct attacks based on control flows. Our work falls into the latter category.

At the same time, there have been a lot of other approaches to detect software attacks. These techniques can be coarsely divided into static or language based techniques [14] and dynamic techniques. The problem with the former is that programs must be rewritten in a new language or recompiled. Dynamic techniques can be further subdivided into those that require compilation and those that do not. Compiler patches such as *Stackguard* [4] and *Stackshield* [23] have been developed to prevent stack based vulnerabilities. Among the software techniques, *Address randomization* [2] make it probabilistically unlikely to launch a successful attack. But they need source code modifications and their overhead is not tested for SPEC benchmarks. One important problem with compiler patches are that they are not applicable to library and legacy code.

The other dynamic techniques can be divided into hardware and software approaches. We already discussed some of the disadvantages of hardware based information flow tracking schemes [21, 6]. They mainly require specialized hardware support in the form of changes to processor pipeline and memory management. The dynamic software techniques typically employ a dynamic translator. Dynamic Taint-checking [15] has very high overhead in the order of 40 times for SPEC integer programs. LIFT [17] reduces it considerably and brings it to a more tolerable level of four times execution time overhead for Spec programs. This overhead may still prove to be a hindrance for widespread use of information flow tracking. With the advent of multicores, this work which utilizes multicores effectively to reduce this overhead, is an important step in that direction. One significant limitation of the current dynamic techniques, including the current work, is that they are not applicable for multithreaded programs. However the current work is the first software technique, as far as we know, that is capable of providing protection against non-control data attacks and is able to perform instruction taint checking.

The use of multicores for security has been proposed in INDRA [20], where a dedicated core in the CMP is channeled for the purpose of detection and recovery from software attacks. For detecting a software attack, their work uses a technique called introspection, which in turn is based on Program Shepherding [10]. Program Shepherding, while being effective against a variety of code injection attacks with a low overhead may not be effective against attacks that dont inject any new code but target existing code such as return into libc attacks [15]. As their technique mainly checks the code origin (ensures that code that has been modified is not executed), it may not be applicable for several programs where it is legal to have an executable stack [21].

There has been prior work [15] that uses DIFT for recovering against a software attack. While the main purpose of the above work, is to generate signatures for the attack, we are more concerned with the problem of identifying the bug in the code that caused the vulnerability in the first place. There has been a lot of recent work [11, 32, 19] that deals with hardware support for debugging. BugNet [11] mainly deals with hardware support for efficient tracing, so that the program can be replayed and debugging can be performed. AccMon [32] provides architectural support for the detection of memory errors. It uses spurious dependencies effectively

to detect memory errors. Our technique does not consider the problem of debugging as a whole, it only considers bugs which could lead to a security attack. It is just a starting point to show that information tracking infrastructure can be an efficient mechanism for debugging also. We feel there is a lot of scope of utilizing the hardware support utilized in BugNet and AccMon along with information tracking infrastructure for efficiently performing debugging.

6. CONCLUSIONS

In this paper, we describe the design and implementation of an information flow tracking infrastructure using multicores. Since we use a dynamic binary translator, it is a transparent mechanism that is applicable to legacy code and proprietary code. Since the multicore design necessitates fine grained communication, we use an inter-core communication queue with added ISA support for sending and receiving. We evaluated our system for correctness and performance and found that the overhead of implementing DIFT in a multicore is only 48%. Since our performance overhead is tolerant to inter-core communication latency, light-weight solutions for enabling inter-core communication can be used. Finally, we describe a novel application of the DIFT infrastructure where it is used to assist in the debugging process.

7. REFERENCES

[1] E.Borin, C.Wang, Y.Wu and G.Araujo, "Software-Based Transparent and Comprehensive Control-Flow Error Detection," *CGO 2006*

[2] S. Bhatkar, R. Sekar and D. C. DuVarney "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *14th USENIX Security Symposium*, Baltimore MD, August 2005.

[3] D. Bruening "Efficient, Transparent and Comprehensive Runtime Code Manipulation," *Ph.D. Thesis, MIT, September 2004*

[4] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. "Stack- Guard: Automatic adaptive detection and prevention of buffer-overflow attacks.", *In Proc. 7th USENIX Security Symposium, pages 6378*, San Antonio, Texas, Jan. 1998.

[5] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer, "Non-control-data attacks are realistic threats", *In Proceedings of the Usenix Security Symposium*, pages 177–192, 2005.

[6] M.Dalton, H.Kannan, C.Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," *Proceedings of the 34th Intl. Symposium on Computer Architecture* (ISCA), San Diego, CA, June 2007

[7] M.Dalton, H.Kannan Hari, and C.Kozyrakis, "Deconstructing Hardware Architectures for Security" *The 5th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)* at ISCA, June 2006.

[8] D. E. Denning. "A lattice model of secure information flow". *Communications of the ACM*, 19(5):236243, 1976.

[9] J.S.Fenton, "Memoryless Subsystems", *The Computer Journal* 17,2 (May 1974), 143-147.

[10] V.Kiriansky, D.Bruening, and S.P.Amarasinghe, "Secure Execution via Program Shepherding", *Proceedings of the 11th USENIX Security Symposium,* pp. 191-206, August 2002.

[11] S. Narayanasamy, G. Pokam, B. Calder, "BugNet: Recording Application-Level Execution for Deterministic Replay Debugging", *32nd International Symposium on Computer Architecture*, June 2005

[12] P.Magnusson et al. "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50-58, Feb., 2002.

[13] A. C. Myers. "JFLow: Practical mostly-static information flow control". *Principles of Programming languages*, 1999.

[14] G. C. Necula, S. McPeak, and W. Weimer. "CCured: Type-safe retrofitting of legacy code." In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[15] J.Newsome, and D.Song, "Dynamic Taint Analysis for Automatic Detection", Analysis, and Signature Generation of Exploits on Commodity Software" *Proceedings of the 12th ISOC Symposium on Network and Distributed System Security*, pp. 221-237, February 2005.

[16] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining" *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[17] F.Qin, C.Wang, Z.Li, H.Kim, Y.Zhou, and Y.Wu, "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks" *Proceedings of the 38th International Symposium on Microarchitecture*, November 2006.

[18] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. August, and G. Cai "Support for High-Frequency Streaming in CMPs", *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, (MICRO), December 2006.

[19] R.Shetty, M.Kharbutli, Y.Solihin, and M.Prvulovic, "HeapMon: A Helper-Thread Approach to Programmable, Automatic, and Low-Overhead Memory Bug Detection", *IBM Journal of Research and Development*, Vol. 50, No. 2/3, 2006.

[20] W.Shi, H.Lee, L.Falk and M.Ghosh, "An Integrated Framework for Dependable and Revivable Architecture Using Multicore Processors", *Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006.

[21] G.E.Suh, J.W.Lee, D.Zhang and S.Devadas, "Secure program execution via dynamic information flow tracking", *Proceedings of the 11th International Conference on Architectural Support For Programming Languages and Operating Systems*, pp. 85-96, October 2004.

[22] 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", *Proceedings of the 37th International Symposium on Microarchitecture*, November 2004.

[23] Vendicator. "Stackshield: A stack smashing technique protection tool for linux". *http://www.angelfire.com/sk/stackshield/.*

[24] J.Xu, and N.Nakka, "Defeating Memory Corruption Attacks via Pointer Taintedness Detection", *Proceedings of the 2005 international Conference on Dependable Systems and Networks*, (Dsn'05) - Volume 00, June 28 - July 01, 2005.

[25] A. Smirnov and T. Chiueh, "DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attack", *12th Annual Network and Distributed System Security Symposium,* 2005.

[26] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar operand networks", *IEEE Transactions on Parallel and Distributed Systems*, February 2005.

[27] D.Wagner and D. Dean, "Intrusion detection via static analysis", *IEEE Symp. on Sec. and Priv.*, pages 156169, 2001.

[28] C.Wang, H.S.Kim, Y.Wu, "Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection", Accepted for publication at CGO 2007.

[29] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models", *IEEE Symp. on Sec. and Priv.*, 1999.

[30] J. Wilander and M. Kamkar. "A comparison of publicly available tools for dynamic buffer overflow prevention.", In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.

[31] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," *15th USENIX Security Symposium*, Vancouver, BC, Canada, August 2006.

[32] P.Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants", *Proceedings of the 37th Annual IEEE/ACM international Symposium on Microarchitecture*, Portland, Oregon, December 04 - 08, 2004.

[33] CERT/CC Statistics 1988-2006, *http://www.cert.org/stats/*

[34] National Vulnerability Database Statistics, *http://nvd.nist.gov/statistics.cfm*