

From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware

Haibo Chen[†], Xi Wu[†], Liwei Yuan[†], Binyu Zang[†], Pen-chung Yew[‡], and Frederic T. Chong[§]

[†]*Parallel Processing Institute, Fudan University*

[‡]*Department of Computer Science and Engineering, University of Minnesota at Twin-Cities*

[§]*Department of Computer Science, University of California at Santa Barbara*

Abstract

Dynamic information flow tracking (also known as taint tracking) is an appealing approach to combat various security attacks. However, the performance of applications can severely degrade without hardware support for tracking taints.

This paper observes that information flow tracking can be efficiently emulated using deferred exception tracking in microprocessors supporting speculative execution. Based on this observation, we propose SHIFT, a low-overhead, software-based dynamic information flow tracking system to detect a wide range of attacks. The key idea is to treat tainted state (describing untrusted data) as speculative state (describing deferred exceptions). SHIFT leverages existing architectural support for speculative execution to track tainted state in registers and needs to instrument only load and store instructions to track tainted state in memory using a bitmap, which results in significant performance advantages. Moreover, by decoupling mechanisms for taint tracking from security policies, SHIFT can detect a wide range of exploits, including high-level semantic attacks.

We have implemented SHIFT using the Itanium processor, which has support for deferred exceptions, and by modifying GCC to instrument loads and stores. A security assessment shows that SHIFT can detect both low-level memory corruption exploits as well as high-level semantic attacks with no false positives. Performance measurements show that SHIFT incurs about 1% overhead for server applications. The performance slowdown for SPEC-INT2000 is 2.81X and 2.27X for tracking at byte-level and word-level respectively. Minor architectural improvements to the Itanium processor (adding three simple instructions) can reduce the performance slowdown down to 2.32X and 1.8X for byte-level and word-level tracking, respectively.

1 Introduction

Software security has become a severe economic and social problem [1, 2]. Apart from typical low-level attacks such as buffer overrun, high-level semantic attacks that subvert legitimate uses of resources have emerged recently as a major security threat. For example, high-level attacks such as cross-site scripting and SQL injection were ranked the top 2 in reported vulnerabilities through 2005 to 2006 [4]. One common feature of these viruses and attacks is that they often hijack the normal control flow of software and/or cause illegitimate uses of untrusted data.

One effective way to combat these attacks is to dynamically track the information (both control and data) flow to defend against malicious uses of tainted data [24, 18, 22, 8]. Generally, these approaches mark (taint) data from untrusted sources (e.g., network), track it during program execution, and detect unsafe usages of the tainted data (e.g., being executed or used as system call arguments). Compared to other techniques, dynamic information flow tracking (DIFT) can provide precise information (e.g., flow of tainted data) to detect and reason about various attacks, even unknown ones, with few or no false positives. Moreover, the results of such reasoning could be used as feedback to generate accurate intrusion prevention signatures [6, 18].

A number of systems have been built to employ DIFT to detect various attacks. Previous systems can be classified into two categories: software-based systems [22, 18, 27] that utilize a compiler or a dynamic binary translator to instrument application code and detect information flow anomalies; and hardware-based systems [25, 24, 7, 8, 26] that provide architectural enhancements to improve the efficiency of information flow tracking. Software-based approaches can assign various policies to detect a wide range of attacks including high-level semantic attacks. However,

they come with a heavy performance slowdown ranging from 4.6X to 37X [22, 18]. Hardware-based systems are more efficient but they require non-trivial changes to core processor architectures and are usually less flexible to handle high-level attacks.

In this paper, we propose a *low-overhead* scheme, called SHIFT (speculative hardware based information flow tracking), that leverages *existing* modern architectural features, namely speculative execution [14, 10] and deferred exceptions [16, 11, 9], to support dynamic information flow tracking. We observe that dynamic information flow tracking is similar to speculative state tracking. To support speculative state tracking, the processor extends each general-purpose register with a deferred exception token to track exceptions (speculative state) during speculative execution. This token is propagated along the program execution path. In taint tracking, the tainted information is also maintained through a tag and the tag must also be propagated along the program execution path.

SHIFT uses the hardware mechanisms for tracking speculative state to track information-flow taints within processors. SHIFT instruments each load and store from/to memory to track taints in memory using a bitmap. Security policies can be assigned by changing a configuration file for the instrumentation compiler. By tracking taints using hardware mechanism but assigning policies in software, SHIFT can detect a wide range of security exploits (including high-level semantic attacks) with a relatively small performance slowdown.

We have implemented a prototype system by modifying GCC to instrument load and stores. The prototype uses the Itanium processor’s deferred exceptions to track taints within the processor. The prototype is complete enough to run many applications without modifications, but currently doesn’t support multi-threaded applications.

We have tested several real-world security exploits against this SHIFT prototype. These tests show that SHIFT can detect all these attacks with no known false positives. Performance measurements indicate SHIFT incurs about 1% performance overhead for server applications. The average slowdown¹ for SPEC-INT2000 is 2.81X (ranging from 1.32X to 4.73X) and 2.27X (ranging from 1.34X to 3.80X) for tracking at byte-level and word²-level respectively, which are to date the best performance results for DIFT systems. We also show that some simple architectural improvements to the Itanium processor can reduce the performance slowdown notably.

In summary, this paper makes the following contributions:

- The observation that dynamic information flow track-

¹Performance slowdown is calculated by dividing the new execution time with the original execution time

²In this paper, a word refers to 8 bytes memory.

ing can be emulated using existing hardware mechanism.

- The SHIFT design for information flow tracking, which is based on the above observation. SHIFT decouples detecting mechanisms from security policy assignments. Hence, it enjoys the advantages of both hardware-based (efficiency) and software-based (flexibility) approaches.
- A working implementation of SHIFT. The implementation shows that SHIFT is effective in defeating real-world attacks and that SHIFT results in a negligible performance slowdown for the Apache web server and a modest slowdown for SPEC-INT2000.

The rest of this paper is organized as follows. In section 2, we provide some background information on speculative execution and the hardware support for tracking deferred exceptions. Section 3 presents the design of SHIFT and section 4 describes the implementation issues for the Itanium processor and GCC. Then, section 5 evaluates the detection ability of SHIFT using a set of real-world security attacks. Section 6 presents a performance evaluation of SHIFT using the Apache web server and the SPEC-INT2000 benchmarks. Finally, we discuss the related work and conclude the paper with a brief note on our future work.

2 Background

As SHIFT reuses hardware support for speculative execution for dynamic information flow tracking, this section provides the necessary background information for both topics.

2.1 Dynamic Information Flow Tracking

Figure 1 gives a real-world buffer-overflow vulnerability in *qwik-smtpd 0.3* and shows how DIFT defeats the exploit. As shown in the figure, the server checks the *clientIP* to prohibit relaying a mail not from the localhost. However, as the server doesn’t check the string length of *arg2* in line 5, an attacker may supply a long input to overwrite *localIP* to *clientIP*. Afterwards, the attacker can relay any e-mail through the server.

A program built with DIFT defeats the exploit by marking the user input (e.g., *arg2*) as tainted (by setting the corresponding tags) and propagating the tags through the program execution path following control and data dependencies. Hence, when the tainted data (*arg2*) bypasses the boundary of *clientHELO* and overwrites *localIP*, the program also marks *localIP* as tainted. Finally, the program detects the security exploit when tainted data is used in an unsafe way. In this example, by specifying a policy that disallows tainted data to be compared and alter the control flow, the exploit is detected.

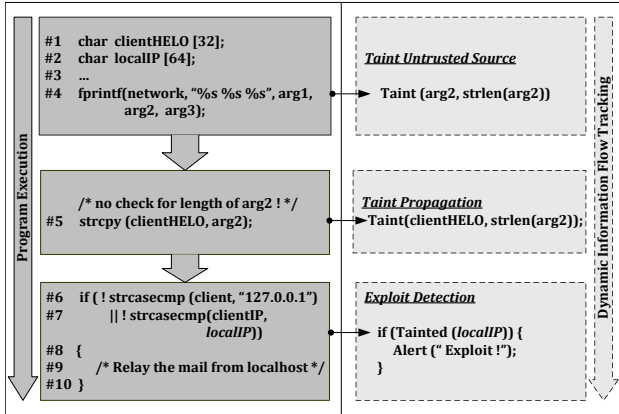


Figure 1: A buffer overflow vulnerability in `qwik-smtpd 0.3` and how DIFT defeats it.

2.2 Speculative Execution and Hardware Support

Speculative program execution is a combination of hardware and software techniques to aggressively execute code as early as possible to hide memory latency, avoid memory aliasing or improve code scheduling. There are typically two types of speculations: control speculation, which optimistically executes code before knowing whether the code should be executed or not in a control path; and data speculation, which executes code depending on likely correct operand values. Currently, SHIFT leverages only *control speculation* to assist information flow tracking.

Figure 2 gives an example of control speculation. If the compiler knows that the branch condition (i.e., “`cond`”) is likely to be true (e.g., from profiling), the load instruction can be moved up to be speculatively executed (using `ld.s`) as early as possible. Since the load operation is moved away from the subsequent store instruction, its execution can overlap with others to hide load latency. The overall critical path can thus be shortened.

However, speculatively executed instructions may cause exceptions, which may not occur during the normal program execution. For example, `r13` may contain an invalid address when `cond` is not true. In such a case, the exception should not occur since the instruction “`ld.s r14 = [r13]`” needs not be executed. To address this issue, researchers have proposed deferred exceptions [16, 11] for precise exception handling during speculative execution. There are generally three extensions to the processor architecture: (1) Each general purposed register is extended with an exception token to record possible exception. Special instructions are provided to test the token in registers. (2) Instructions are categorized as either speculative or non-speculative. Exceptions caused by speculative instructions are recorded instead of being thrown out immediately. (3) The processor pipeline is modified to propagate the exception token along

<pre> if(cond) { ld8 r14 = [r13] and r15 = r14, 8 st8 [sp] = r15 } ... </pre> <p>Original Code</p>	<pre> ld8.s r14 = [r13]; and r15 = r14, 8 ; ... if(cond) { chk.s r15, recovery; Next: st8 [sp] = r15; } ... recovery: ld8 r14 = [r13]; and r15 = r14, 8; br.cond next; </pre> <p>Optimized Code</p>
-----------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 2: An example code of control speculation: the `ld` instruction is speculatively moved up so that its execution can overlap with others. An instruction checking exceptions (`chk.s`) is inserted in its initial location to catch speculation failure and jump to the recovery code.

the program execution path.

If an exception occurs when an instruction is speculatively executed, the exception token in the target register will be set instead of signaling the exception. The token in that register will then be propagated along the program execution path in an OR-based fashion, that is, if one source register is with an exception token, the token in the target register will be set. In the example above, if an exception occurs when “`ld r14 = [r13]`” is executed, the exception token in `r14` is set. The token is then propagated to `r15` and tested by the `chk.s` instruction, which examines the exception token in `r15` and redirect the execution to the recovery code.

Registers with exception tokens cannot be used by non-speculative operations which may cause possible side effects such as altering control flows and memory operations, to prevent bringing irreversible state. Improper uses of the tokens will trigger an exception.

3 SHIFT Design: From Speculation to DIFT

From the previous section, it can be observed that dynamic information flow tracking (DIFT) and deferred exception propagation (DEP) are similar in nature: DIFT tracks tags describing untrusted data, while DEP tracks tokens describing deferred exceptions. Both need to propagate the tags during program execution. They also need some mechanisms to detect possible violations. Because of their similarity, DIFT can be implemented mostly in the same way as DEP if we simply *treat tags describing tainted data as deferred exceptions*.

Building on DEP, we consider a DIFT scheme using hardware-provided mechanisms to track information in internal processors and accelerate DIFT, while using software-assigned policies to specify security violations. Hence, security policies can be cleanly separated from the

tracking and detection mechanisms. This allows developers to implement both high-level and low-level security policies. Existing commodity hardware components can be reused instead of mandating design changes to the core processor components or memory systems.

3.1 Tracking Tags within Processors

To support DIFT, we need to extend DEP to include:

Setting and clearing taint tags. In DIFT, setting and clearing taint tags happen frequently, to contaminate and purify data according to program semantics. For example, when loading tainted data, the taint tag in the target register should be set accordingly. However, such instructions are absent in the typical ISA supporting DEP. It is rather straightforward to add such instructions without touching the internal processor pipelines.

Taint-aware compare instructions. To survive speculative failure, compare instructions in DEP usually reset the flags for both branch targets when the branch condition contains an exception token. This prevents mis-speculation from causing irreversible changes to program state. However, it breaks DIFT since sometimes a branch condition is allowed to be tainted. Hence, it is necessary to add taint-aware compare instructions that proceed normally even if the branch condition is tainted.

Other ways to extend DEP to support DIFT include adding taint tags to on-chip caches, widening memory buses and dedicating memory for taint tags in the DRAM, as done in previous work [7, 24, 8]. These extensions are promising to improve performance as taint tags can be automatically managed by hardware. However, as our design goal is to minimize the changes to standard, commodity processors, we instead use software approaches to handle tag exchanges between processors and memory systems.

3.2 Tracking Tags in Memory Systems

One remaining issue is that, in speculative execution, deferred exception tokens will never be propagated to the memory system (including caches and main memory). Thus, it is required to track the taint tags describing tainted data in memory. Similar to other prior work [22, 18, 27], SHIFT uses a bitmap to maintain in-memory taint information. The bitmap maintains a bit (i.e., tag) for each memory location indicating whether the location is tainted or not ("1" for tainted data and "0" for normal data).

To maintain the coherence of taint tag between memory and registers, SHIFT instruments memory operations (e.g., load and store) in a program. On loading data from memory/cache to registers, SHIFT inserts code to first consult the bitmap whether the data is tainted or not. The taint bit is then set to the exception token in the target register. On storing data to memory/cache, the inserted code updates the bitmap according to the exception-token in the source register.

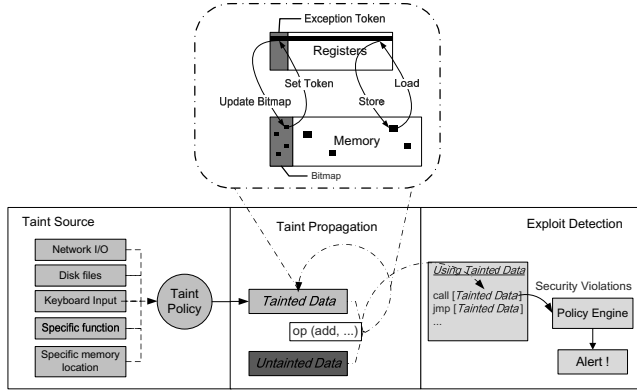


Figure 3: The general working flow of SHIFT.

3.3 SHIFT

Figure 3 depicts the working flow of information flow tracking in SHIFT. First, a program tags tainted data according to the predefined policies. Then during program execution, the processor automatically propagates the tags (i.e., exception tokens) in registers and SHIFT uses a bitmap to maintain the tags in memory. Finally, upon an illegitimate use of tagged data, the program raises a security alert and handles it according to a predefined policy engine. The rest of this section details the design of SHIFT.

3.3.1 Taint Sources

Taint sources determine which data should be marked as tainted in the bitmap and registers. Since the taint sources may vary for different applications and diverse attacks, SHIFT allows customization. Generally, the following channels can be potential sources of tainted data: (1) network I/O; (2) disk files; (3) keyboard input; (4) return values of specific functions; (5) specific memory locations. SHIFT allows users to configure the sources of tainted data by writing a configuration file. The compiler uses the policies in the configuration file to set the taint tags for data from untrusted sources.

3.3.2 Taint Tracking

SHIFT relies on exception token propagation to track taint tags in processors. It uses instrumentation code generated by compilers to maintain taint tags in memory. Similar to previous work [25, 6, 27, 22], SHIFT tracks only data dependency and does not track control dependency since many attacks do not rely on control dependency [22].

Propagation Rules: Generally SHIFT uses the default exception-token propagation rules in computations. For memory operations, SHIFT supports a more customizable policy and allows propagation of tags from/to address registers and the referenced memory contents. For example, in a load operation such as `ld r14=[r13]`, the exception token can be set according to both the address register `r13`

and the data referenced by $r13$. This allows flexible taint policies for pointers, e.g., whether a tainted pointer should be allowed to reference data and how the tag is propagated if it is allowed.

Another issue in tag propagation is handling bounds checking code and translation tables, which is important in increasing the accuracy of exploits detection and reducing both false positives and false negatives [8]. SHIFT handles them using two approaches. First, since SHIFT can access program code and thus has program semantics, it can identify such code using program analysis. Second, for specific translation or lookup tables, SHIFT allows users to write application-specific rules which assist the software (e.g., compiler) to recognize and instrument such code.

Implicit Information Flow: SHIFT handles corner cases such as $xor\ r15=r15,r15$ and $sub\ r15=r15,r15$ by clearing the taint tag in the register. However, SHIFT currently does not aim at handling general implicit information flow, since it is usually of little importance and may incur many false positives [22, 8]. Moreover, modern compilers can usually analyze simple implicit information flow and translate it into explicit information flow to improve performance. Simple compiler analysis could also be useful in handling some specific kinds of implicit information flow, if desired.

3.3.3 Violation Detection

Naturally, the default policies in DEP prevent the uses of tainted data tagged with taint tag from being moved into special-purpose registers or used as branch targets or return addresses. These policies are normally applicable for most applications. Moreover, SHIFT can insert instructions checking for exception token ($chk.s$) before the use of critical data. Using $chk.s$ allows handling of security violation exceptions at the user-level, which can significantly reduce the overhead of analyzing program behavior or perform further security checks to filter out false alarms.

3.3.4 Combining SHIFT with Control Speculation:

Although SHIFT has used the exception-token for taint tracking, control speculation can still use it when necessary, at the cost of some false positives. The approach is straightforward: reverting execution to non-speculative version of code upon speculation failure, no matter whether the exception token is caused by tainted data or deferred exceptions. The reason is that speculatively executed code fragments should not have any side-effects on memory, e.g., storing the pending results into memory system. Thus, the executed instruction trace does not need information flow tracking code since there is no committed memory operation operating on tainted data. The recovery code contains a non-speculative version of the code and follows the normal information flow tracking policies to propagate the exception-token.

Since a speculation failure may be caused by tainted data instead of deferred exceptions, it may introduce false positives for control speculation. Thus, to preserve program performance, control speculation is effective only when there is little tainted data involved. Profiling-guided optimizations could be helpful to decide whether control speculation is efficient in a specific code fragment.

4 Implementation

We have implemented SHIFT by modifying GCC and using the Itanium processor. Other than missing a few important instructions, the Itanium’s deferred exception design is good enough to allow us to demonstrate applicability and efficiency of applying DEP to DIFT. Implementing SHIFT on a commodity available processor enables us to run widely-used software with realistic security policies and measure real performance, instead of using simulation or emulation. We believe that SHIFT can be similarly implemented on other processors built with deferred exception mechanisms. This section describes the implementation and possible future work.

4.1 Implementation Issues on Itanium

Deferred Exception Support in Itanium: Itanium processors are built with good support for deferred exception tracking. Each general purposed register has an additional NaT bit (NaTVal for floating point registers) to record the deferred exception token. The NaT-bits are propagated in parallel with the data or addresses. Itanium also provides an instruction ($chk.s$) checking the existence of the NaT-bit and jumping to recovery code if the NaT-bit is set. There are also instructions ($ld8.spill$ and $st8.fill$) to save/load the NaT-bit to/from a NaT register (UNAT), which is automatically saved across function calls.

Setting and Clearing NaT-bit: Unfortunately, the Itanium ISA does not include instructions to set and clear the NaT-bit in a register. In SHIFT, the taint source (i.e., NaT-bit) for a register is obtained by *artificially* generating a deferred exception that sets the NaT-bit of the register. The content of the register is set to zero. Other registers requiring NaT-bits can perform *add* operations to taint themselves. To clear the NaT-bit in a tainted register, SHIFT first uses a spill instruction (e.g., $st8.spill$) to spill the register to memory and then loads it to the register without filling the NaT-bit. These operations hurt performance since SHIFT needs to perform such operations frequently.

Relaxing NaT-sensitive Instructions: In Itanium, the predicate registers for both branch targets are usually cleared to zero if the condition contains a NaT-bit. Further, load or store from an address containing a NaT-bit will cause a NaT consumption fault. These prohibit the use of tainted data in compare-related instructions and load or store addresses. To handle these, SHIFT relaxes the operands to these instructions when necessary. Specifically,

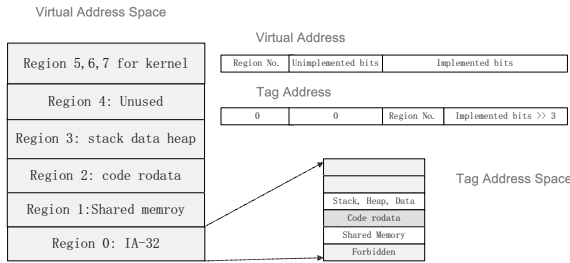


Figure 4: Mapping the virtual address space to the tag address space.

SHIFT analyzes the legitimate uses of tainted data and adds some relaxing code. The relaxing code clears the NaT-bit before the legitimate uses and restores the NaT-bit after the use completes.

Tag Space Management: The tag space is the virtual address space for taint tags. Updating the tag space requires address translation between the virtual address space and the tag space. Unfortunately, the translation is more costly in Itanium than in traditional x86 machines. The virtual address space in Itanium is usually partitioned into eight equally-sized regions, with region 0 being reserved for IA-32 applications. The top three bits in an address indicate which region the address refers to. The region 0 is not used by normal applications, and SHIFT reuses it for tag space. Itanium also uses *unimplemented bits* to limit the virtual address space available to software, by forbidding the use of a range of bits in a 64-bit address. The unimplemented bits create holes in a virtual address space. Thus, one cannot use a simple right-shift operation (e.g., `address >> 3`) to obtain the tag address from a virtual address. Instead, SHIFT moves down the region number and combines with it the implemented bits to obtain the final tag address, as shown in Figure 4.

4.2 Compiler Implementation Issues

We modified gcc-4.1.1 to implement SHIFT for Itanium. SHIFT is composed of two main parts: (1) *policy specification* that controls what data should be marked as tainted and sets the actions when a security alert is raised; and (2) *instruction instrumentation* that instruments loads, stores and compare-related instructions. Users specify policies by writing a simple configuration file, which is then read by SHIFT to control the process of instrumentation. At runtime, the instrumented code checks if the program execution violates the assigned policies.

Information flow tracking can be implemented in various software life-cycles, including source-level [27], high-level intermediate representation (IR), low-level IR, and even at runtime [22]. Currently, we chose to implement SHIFT mainly at the low-level IR (RTL in GCC) since with this choice SHIFT can still extract enough program semantic in-

formation while potentially supporting multiple languages. And, SHIFT can share the fine-grained control over information flow at instruction level. SHIFT is implemented by adding a phase between the phases “pass_leaf_regs” and “pass_sched2” in the GCC back-end. In this phase, all registers have been allocated and the instructions are not scheduled yet. Thus, the implementation of SHIFT can avoid interference with the register allocation algorithm and the code scheduling algorithm for Itanium, which are two of the most complicated phases in GCC.

As SHIFT requires accesses to the source code, it cannot instrument code written directly in assembly code. To overcome this problem, SHIFT requires users to provide wrap functions that summarize the taint tag propagation of the untransformed assembly functions, as done in [27]. In instrumenting glibc, we added about 17 such wrap functions.

4.3 An Example of Taint Tracking Code In SHIFT

<pre> 1. dep.z r4=-81,56,8;# fake an invalid addr 2. ld8.s r4=[r4]; # generate a NaT error 3. xor r4=r4,r4; # zero the r4 Get a zero-value register with NaT bit 1. extr.u r30=r14,61,3 # get region no 2. extr.u r31=r14,3,40;# get lower 40 bits 3. dep r31=r30,r31,40,3;# get bitmap addr 4. ld1 r30=[r31]; # get tags 5. cmp.ne p14,p15=0,r30;# tainted ? 6. ld8 r15=[r14];# perform the real load 7. (p14) add r15=r4,r15 # taint target reg ld8 r15 = [r14] </pre>	<pre> 1. tnat.nz p12,p13=r16 # src reg tainted? 2. extr.u r30=r33,61,3 # get region no 3. extr.u r31=r33,3,40; # get lower 40 bits 4. dep r31=r30,r31,40,3; # get bitmap addr 5. (p13) mov r30=0 # clear taint tag 6. (p12) mov r30=ff; # add taint tag 7. st1 [r31]=r30 # update bitmap 8. st8.spill [r33]=r16 # perform the real store st8 [r33]=r16 </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 5: An example of instrumenting load and store instructions for information flow tracking, as well as the generation of a source register with NaT-bit. The instructions with bold font are the original instructions. The numbered ones are the instrumented version accordingly.

Figure 5 gives an example of instrumenting load and store instructions for information flow tracking generated by SHIFT. It includes the code to obtain a NaT-bit. To obtain a source register with the NaT-bit set, SHIFT fakes an invalid address (Instruction 1) and issues a speculative load (i.e., `ld8.s`) from the address. Speculative loading from an illegal address will set the NaT-bit in the target register instead of raising an exception. The content of the target register is cleared to zero so that it can be used as a NaT-bit source to taint other registers.

For a load operation, instructions 1-4 get the corresponding tag from the bitmap. Instruction 5 tests if it is tainted. Instruction 6 performs the real load operation and instruction 7 taints the target register if the tag in the bitmap is set. Note that SHIFT normally does not allow a load from a tainted address and permits it only if the analysis (e.g., bounds checking) indicates that the load is safe.

For a store operation, instruction 1 tests if the source reg-

ister is tainted or not. Instructions 2-7 update the bitmap according to the NaT-bit in source registers and the original value in the bitmap. Instruction 8 performs the real store operation. Since *st8.spill* allows storing a register with a NaT-bit into memory, we choose *st8.spill* instead of *st8* to omit additional code to save and restore the NaT-bit of the source register.

4.4 Discussion

Compiler Optimizations: There are many compiler optimization opportunities in SHIFT. Sophisticated compiler optimization could further reduce the performance overhead of SHIFT. In our future work, we plan to optimize SHIFT to reduce unnecessary tracking code and enable adaptive tracking. For example, we intend to use program analysis and profiling-guided optimizations.

Self-Modifying Code: As a compiler-based instrumentation system, SHIFT has difficulties in handling self-modifying code that is not aware of SHIFT instrumentation. This may lose some taint information if tainted data is involved in self-modifying code. Fortunately, self-modifying code is usually not common, and it is generally rare for self-modifying code to operate on tainted data.

Multi-threaded Code: Like most previous software-based systems, our current implementation does not support multi-threaded applications since accessing the bitmap is not serialized. In our future work, we intend to extend SHIFT for multi-threaded applications and investigate the performance implications.

Possible Minor Architecture Enhancements: As pointed out before, setting and clearing the NaT-bit in a register is rather costly on Itanium. In our development process, we found that artificially generating a register with the NaT-bit at the granularity of functions degrades the performance by a factor of 3X, compared to generating a NaT-bit and keeping it for all subsequent uses. Thus, we believe that adding simple instructions to set and clear the NaT-bit will largely improve program performance. Further, adding a compare instruction that works for operands with NaT-bit will save the cost for spilling and filling a NaT-bit in relaxing a compare instruction. We present a quantitative measurement on the performance benefit in section 6.3.

5 Security Evaluation

This section evaluates the detection ability of SHIFT using a set of real-world attacks, including both high-level and low-level exploits. We first describe the security policies used in SHIFT to defend against typical attacks. Then, we use the described policies to defend against real-world attacks to measure their effectiveness as well as false alarms.

5.1 Attack Detection Policies

The main goal of SHIFT is to detect both low-level memory corruption exploits and high-level semantic attacks.

Thus, the policies include both high- and low-level policies. Policies in SHIFT are not fixed and can be easily adjusted for diverse applications. Table 1 shows an incomplete list of policies in SHIFT and the corresponding attacks that the policies are to defend against.

For example, policy H1 and H2 protect applications from directory traversal attacks, by not allowing tainted data used as a file path name to be absolute paths (e.g., starting with “/”) or traversing out of the document root (e.g., using multiple “..” strings to forge a file path that is out of the document root). For low-level policies, policy L1 prevents a program from de-referencing a tainted pointer and policy L2 prevents a potentially malicious *store* instruction from overwriting critical data (e.g., GOT entry in an ELF file). Policy L3 guarantees that the important state of CPU cannot be overwritten by tainted data. Specifically, policy L3 prohibits a program from transferring control to malicious code, by not allowing tainted data to be moved into branch registers. The low-level policies are relatively fixed and are usually turned on as the default policies in SHIFT. Multiple policies can be combined to detect specific attacks.

5.2 Attack Detection

Table 2 summarizes our security evaluation using several real-world vulnerabilities obtained from CVE ³: three directory traversal exploits of tar, gzip and Qwikiwiki; three cross-site scriptings with Scry, php-stats and phpsysinfo; one SQL command injection; and one format string attack (we made a minor adjustment of Bftpd to make it vulnerable of arbitrary code execution). We do not provide attacks that overflow function return addresses since Itanium has already prevented them by using dedicated registers for function calls and returns.

We compile the applications using SHIFT with the specified policies. The applications are tracked at both byte-level and word-level. We first run them normally without attacking them to see if there are any false positives. In our tests, all applications run normally and no security alert is raised. Then, we attack the applications using artificially forged input to evaluate the detection ability of SHIFT. All these attacks are successfully detected by SHIFT. Without SHIFT protection, all attacks succeed.

SHIFT detects these attacks by combining various high-level and low-level policies. For example, SHIFT marks data read from disks as untrusted and combines the low-level policies with policy H1 to detect directory traversal attacks to GNU Tar. To detect security exploits on Qwikiwiki, SHIFT marks the file path as tainted when reading the http request and tracks the propagation of the tainted string. When the tainted data is used as an argument of fopen, SHIFT examines the argument. If the file path traverses out of the document root, then a security alert is raised. Pol-

³Common Vulnerabilities and Exposures, <http://cve.mitre.org/>

Policy	Attacks to Detects	Description
H1	Directory Traversal	Tainted data cannot be used as an absolute file path
H2	Directory Traversal	Tainted data cannot be used as a file path which traverse out of the document root
H3	SQL Injection	Tainted data cannot contain SQL meta chars when used as a part of the SQL string
H4	Command Injection	Tainted data cannot contain Shell meta chars when used as arguments to system()
H5	Cross Site Scripting	No tainted script tag
L1	De-referencing tainted pointer	Tainted data cannot be used as a load address
L2	Format string vulnerability	Tainted data cannot be used as a store address
L3	Modify critical CPU state	Tainted data cannot be moved into special registers

Table 1: Security Policies in SHIFT

CVE#	Program (Version)	Language	Attack Type	Detection Policies	Detected?
2006-6097	GNU Tar (1.4)	C	Directory Traversal	H1 + Low level policies	Yes
2005-1228	GNU Gzip (1.2.4)	C	Directory Traversal	H1 + Low level policies	Yes
2006-0983	Qwikiwiki (1.4.1)	PHP	Directory Traversal	H2 + Low level policies	Yes
2006-2001	Scry (1.1)	PHP	Cross Site Scripting	H5 + Low level policies	Yes
2007-4334	php-stats (0.1.9.2)	PHP	Cross Site Scripting	H5 + Low level policies	Yes
2005-3347	phpsysinfo (2.3)	PHP	Cross Site Scripting	H5 + Low level policies	Yes
2006-6912	phpmyfaq (1.6.8)	PHP	SQL Command Injection	H3 + Low level policies	Yes
N/A	Bftpd(0.96 prior)	C	Format string attack	L2	Yes

Table 2: Security Evaluation Results of SHIFT.

icy L2 is strong enough to detect exploits on the example format string vulnerability in Bftpd. The malicious input causes Bftpd to overwrite the GOT entry for library function such as “*system*”. Since the address to *store* is tainted and there is no explicitly bounds checking, a security alert is raised.

False Positives and False Negatives:

As a flexible information flow tracking system, SHIFT can assign various policies to a protected program. However, one drawback of the flexibility is that SHIFT may suffer from possible false positives or false negatives due to overly restrictive or overly permissive policies. Fortunately, typical security exploits usually have some common and expressive characteristics. Hence, it is often not difficult to assign accurate policies to protect a program from typical security exploits in practice. One possible but rare issue is the false alarms due to implicit information flow or information propagation through control dependency, which SHIFT currently does not track. Fortunately, they do not seem to be a serious problem in practice, as pointed out by previous work [27, 22, 8]. As expected, we did not experience either false positive or false negative in our security evaluation. Furthermore, to handle sophisticated security exploits, one might be able to use machine learning techniques to reduce or eliminate possible false alarms.

6 Performance Results

In this section, we measure the performance of SHIFT to answer the following questions: (1) whether applica-

tions using SHIFT can provide an acceptable level of performance? (2) whether adding three instructions can further reduce the performance slowdown? (3) what contributes to the remaining overhead and in what proportions?

The tests were performed on an HP Integrity rx1620 server equipped with two 1.6GHz Itanium processors and 4GB of memory running Redhat Linux Enterprise 4. We test Apache web server and eight SPEC-INT2000 benchmarks executed with the reference inputs. We compare the performance of the benchmarks compiled using the original GCC-4.1.1 and our enhanced instrumentation compiler (SHIFT-GCC) at the -O3 optimization level (except -O1 for 176.gcc⁴).

6.1 Overhead with Apache

The measurements are performed by using the apache benchmark (ab) to issue 1,000 requests for a single file with 200 concurrent processes. The requested file size is 4 KB, 8 KB, 16 KB and 512 KB each time. Figure 6 shows that SHIFT incurs negligible overhead for Apache. The geometric mean of the overhead for throughput and latency at all file sizes is about 1%. The overhead of SHIFT mainly comes from the added instrumentation for load and store instructions. Since Apache is an I/O intensive application, the instrumentation added by SHIFT has only a little impact on its performance. The overhead for requesting a 4 KB file is a bit larger than that for other file sizes. This is

⁴GCC-4.1.1 for Itanium cannot successfully build 176.gcc at -O3 optimization level

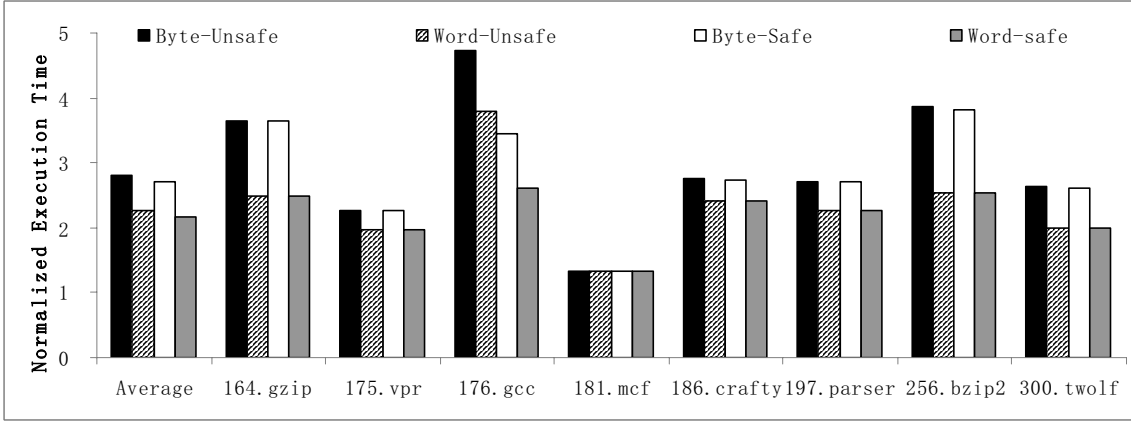


Figure 7: The relative performance of SHIFT against non-instrumented version for SPEC-2000: the four bars mean tracking at byte/word level with input data tagged as unsafe/safe.

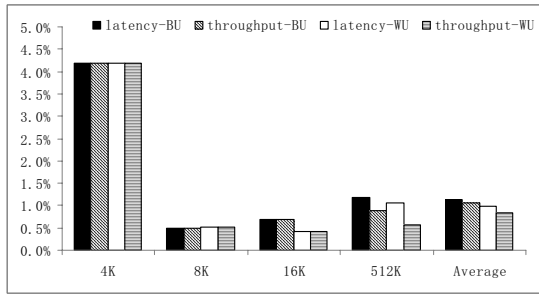


Figure 6: The relative performance of SHIFT against non-instrumented version for Apache: the four bars mean the overhead for latency and throughput at byte/word level.

because the I/O processing time in requesting a 4 KB file contributes a bit less in the total program execution time than requesting files in other sizes. Nevertheless, the incurred overhead is still low (about 4.2%). The overhead for tracking at byte-level is a bit more than tracking at word-level since the former one requires more code to instrument a single instruction.

6.2 Overhead with SPEC-INT2000

To measure the performance slowdown with the instrumented code for benchmarks in SPEC-INT2000 operating on tainted data, we mark all data read from disk as tainted. We compare the performance of applications compiled by unmodified GCC and SHIFT-GCC. Figure 7 depicts the performance slowdown for each individual benchmark as well as the geometric average of eight benchmarks. As shown in the figure, the performance slowdown ranges from 1.32X to 4.73X for byte-level tracking and 1.34X to 3.80X for word-level tracking. The average slowdown is 2.81X for byte-level tracking and 2.27X for word-level tracking when untrusted data is involved. The slowdown mainly comes

from the code to instrument load and store, and to relax compare-related instructions. The performance slowdown of SHIFT is significantly smaller than LIFT (4.6X)⁵ [22].

6.3 Architectural Enhancements

As there are no simple instructions to set and clear the NaT-bit in a register on Itanium, the cost is relatively high for such operations. To get a quantitative result on the impact of minor architectural enhancements, we adjust the instrumentation code and test the following configurations: (1) using two simple instructions to simulate the effect of setting and clearing the NaT-bit in a register; (2) removing the relaxing code for compare instructions to simulate the effect of providing a NaT-aware compare instruction. We compare the results with the basic data of byte/word level tracking with tainted data (byte/word-unsafe).

As shown in figure 8, the first architectural enhancement results in a reduction of 16% performance slowdown for both byte/word level tracking. Here, the reduction of performance slowdown is the difference between the original and new performance slowdowns. Combining the two architectural enhancements can result in a reduction of 49% and 47% in total for byte-level and word-level tracking. The reduction of slowdown ranges from 2% to 173% and 5% to 166% respectively, depending on the amount of involved tainted data in each benchmark. For example, the reduction of slowdown for gcc is 173% after applying the two enhancements for byte-level tracking (166% for word-level tracking). By contrast, the reduction is rather smaller for applications manipulating relatively little tainted data : only 2% and 5% for mcf when it is tracked at byte-level and word-level.

⁵LIFT uses the notion of “incurred overhead”, which equals (“slowdown” - 1). Thus, the “incurred overhead” of 3.6X should be 4.6X slowdown.

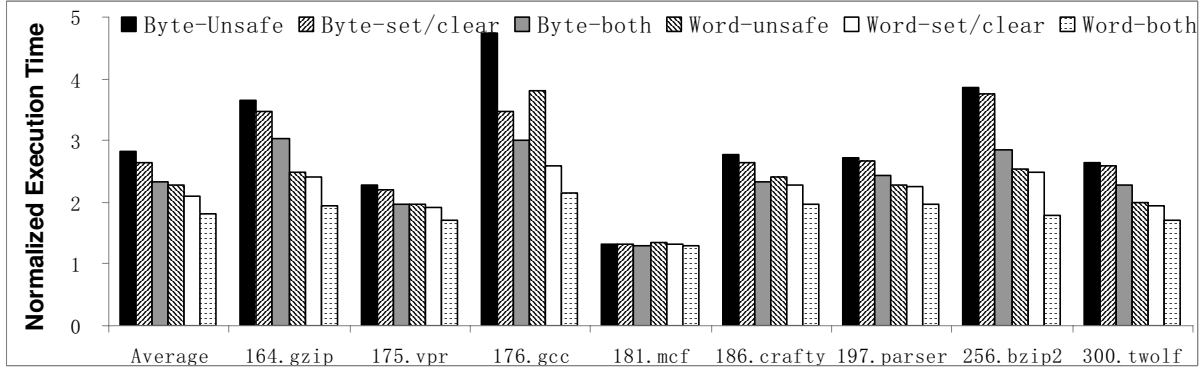


Figure 8: The impact of minor architectural enhancements. We compare the data for adding instructions for set/clear NaT bits (byte/word-set/clear) and the data for adding instructions for both the set/clear NaT-bits and NaT-aware comparison (byte/word-both) against the original SHIFT version (byte/word-unsafe).

6.4 Remaining Overhead

After the architectural enhancements, there is still some performance overhead for SHIFT, which mainly comes from the basic costs to instrument each load and store instruction. The costs are mainly composed of two parts for each load and store instruction: (1) computation that translates a virtual address to a tag address and computes the tag. (2) memory access that reads or updates the bitmap. To understand their contributions to the total cost, we measured a breakdown of the cost for each part. Figure 9 shows the performance overhead from computation and memory access in load and store instructions for each benchmark in SPEC-INT2000. As shown in the figure, computation incurs much more overhead than memory access to tag space. This is probably because the unimplemented bits in Itanium make computing a tag more costly than traditional x86 architectures. Since most memory accesses actually hit in L1 cache, the cost for memory access is not significant. The gap between computation and memory access is more significant for byte-level tracking since computing a tag for a byte is more complex than that for a word. The instrumentation for load instructions contributes much more overhead than that for store instructions, since the number of executed load instructions is much larger than the number of store instructions according to our analysis using PIN [15]. Based on the fact that computation contributes to the major slowdown in SHIFT, one possible compiler optimization might be reusing the computation code for some adjacent data.

6.5 Code Size Expansion

Table 3 shows the impact of compiler instrumentation on the code size of glibc and SPEC-INT2000. The expansion in code size is not significant for these applications. The degree of the code size expansion depends on the proportion of instructions required to be instrumented, including load, store and comparison instructions. The expansion in

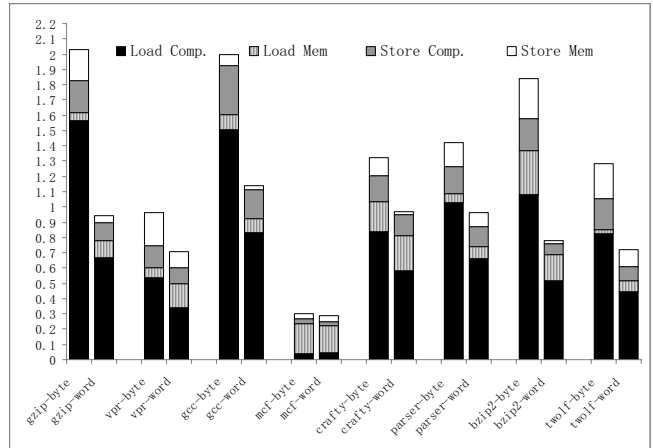


Figure 9: A breakdown of the performance slowdown among computation and memory access in load and store instructions for tracking at both byte- and word-level.

code size is relatively small (35% and 45%) for glibc. By contrast, the expansion for benchmarks in SPEC-INT2000 is more notable due to the relatively large proportion of instrumented code. Since information flow tracking at byte-level requires a bit more code than that at word-level, the code expansion is a bit larger for byte-level tracking than word-level tracking.

7 Related Work

While there are a number of dynamic information flow tracking (DIFT) systems, SHIFT differs from existing efforts in that it makes novel uses of existing hardware supports for speculative execution. The following discussion will focus on most related work to SHIFT.

Apps.	Orig. size	Word level	Overhead	Byte level	Overhead
glibc	11M	15M	36%	16M	45%
gzip	192K	528K	175%	627K	226%
gcc	3.6M	9.0M	150%	9.6M	160%
crafty	541K	1.3M	146%	1.5M	184%
bzip2	164K	531K	223%	637K	288%
vpr	485K	1.1M	132%	1.3M	174 %
mcf	59K	163K	176%	167K	183%
parser	583K	1.6M	181%	1.7M	198%
twolf	851K	2.5M	200%	2.8M	237%

Table 3: The impact of compiler instrumentation on code size.

7.1 Software-based DIFT

Source-level instrumentation [27, 12] is a viable solution to track dynamic information flow and enforce security policies. It instruments the source code of software with DIFT code that propagates security tags and checks security violations. It shares the high-level semantic information in the source code, but loses low-level control of the generated code. It thus has difficulty in taking advantage of the architectural support to lower the performance overhead. Consequently, it incurs a relatively high performance overhead, which prevents its wide use in production run.

Dynamic binary translation is an alternative approach that instruments binary code on-the-fly with security tag management and detection mechanisms (e.g., LIFT [22] TaintTrace [3]). In contrast to source-level instrumentation, it does not require accesses to the source code and can have fine-grained control of hardware resources. Thus, it is capable of utilizing hardware features to lower DIFT overhead. For example, LIFT [22] uses additional 64-bit registers in x86-64 for security tag propagation. Furthermore, the runtime information such as execution traces opens the opportunities to adaptive tracking of security tags to reduce the performance loss. LIFT heavily uses the information to lower the performance slowdown from 27.6X to 4.6X. However, both LIFT and TaintTrace cannot detect high-level semantic attacks.

Interpretation or emulation [18, 19, 21, 23] tracks dynamic information flow by translating the executing instructions (either high-level or low-level) into lower-level operations. They then can embed operations to track dynamic information flow and detect information anomalies. The advantage of these approaches is that they do not need to access the source code. The major disadvantage is that the incurred overhead can be quite significant. It is also hard to detect high-level attacks that depend on program semantics.

7.2 Hardware-based DIFT

Minos [7] and work done by Suh et al. [24] are two parallel and independent efforts aiming at providing efficient hardware supports to DIFT systems. These supports include

tagging registers and caches, adding tag propagation mechanisms in instruction set architectures. Minos targets only control data attack while the latter handles both control and data attacks. Both are designed to combat low-level attacks such as memory corruption attacks but cannot detect high-level semantic attacks.

Raksha and FlexiTaint[8, 26] are two recent DIFT systems that try to improve the flexibility and programmability of hardware-based DIFT system. It allows software to direct hardware analysis and to gain control of security violation handlers at the user-level. Thus, it is capable of detecting high-level attacks as well as multiple concurrent attacks. Moreover, instead of using simulation, it provides a FPGA-based prototype and supports information flow tracking through operating systems.

Speck [20] aims to provide a unified framework to accelerate security checks multi-core platform, by parallelizing computation code and taint tracking code. They use process-level log and replay to synchronize state between the computing thread and the security-tracking thread. Checkpoint and rollback are used to ensure that an application can be rolled back to a safe state once an intrusion is detected. Their idea is orthogonal to SHIFT, which reuses instruction-level speculation instead of OS-level speculation.

7.3 Other Uses of DIFT Systems

Apart from using DIFT to detect security exploits, there are efforts in utilizing DIFT for debugging, testing and programming understanding [17, 13, 5]. Specifically, Masri et al. [17] propose using dynamic information flow analysis to discover and debug unsafe flow in program, to enforce information flow policies. COMET [13] uses dynamic taint tracing to improve the coverage of software testing. Further, being aware of the importance of DIFT, there are also efforts trying to implement general DIFT systems that are customizable to detect security exploits, analyze program behavior and testing [5, 12]. However, the generality provided in these systems is usually at the high cost of performance overhead. For example, GIFT [12] requires a call to a function on each taint tracking operation.

8 Conclusion and Future Work

We have presented SHIFT, a low-overhead dynamic information flow tracking system for improving software security. SHIFT leverages existing hardware support for deferred exception tracking to lower the runtime overhead. We have implemented a prototype by modifying GCC and using the Itanium processor. Our security evaluation shows that SHIFT can defeat a set of real-world attacks with no false positives. Performance measurements on SPEC-INT2000 indicate that the performance slowdown due to SHIFT is modest. Quantitative measurements show that minor architectural enhancements can reduce the performance slow-

down further.

As future work, we plan to extend and improve SHIFT in several directions. First, we are currently exploring various compiler optimization techniques such as adaptive tracking and profiling-guided optimizations to further lower the incurred performance overhead. Second, we plan to extend and apply SHIFT to analyze modern security exploits, generate accurate intrusion-prevention signatures, and detect possible information leakages. Third, as SHIFT is currently implemented using a compiler and requires accesses to the source code, we plan to implement SHIFT using a binary translator and apply various dynamic optimization techniques to further lower its overhead. Finally, we intend to extend SHIFT for multi-threaded applications and investigate this extension's performance implications.

Acknowledgment

The authors thank Frans Kaashoek and the anonymous reviewers for their insightful comments. This research was funded by China National 973 Plan under grant numbered 2005CB321905.

References

- [1] Cybersecurity: A crisis of prioritization. Technical report, Presidents Information Technology Advisory Committee (PITAC), Feb. 2005.
- [2] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and J. Nazario. The Blaster Worm: Then and Now. *IEEE Security & Privacy*, 3(4):26–31, 2005.
- [3] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proc. ISCC*, 2006.
- [4] S. Christey and R. A. Martin. Vulnerability type distributions in cve. <http://cwe.mitre.org/documents/vulntrends/index.html>, May 2007.
- [5] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proc. ISSSTA*, pages 196–206, 2007.
- [6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proc. SOSP*, pages 133–147, 2005.
- [7] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *Proc. Micro*, pages 221–232, 2004.
- [8] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. ISCA*, pages 482–493, 2007.
- [9] C. Dulong. The IA-64 architecture at work. *Computer*, 31(7):24–32, 1998.
- [10] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *Micro, IEEE*, 20(5):12–23, 2000.
- [11] V. Kathail, R. Gupta, B. Rau, M. Schlansker, W. Worley Jr, and F. Amerson. Method and system for deferring exceptions generated during speculative execution, Nov. 25 1997. US Patent 5,692,169.
- [12] L. C. Lam and T. cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proc. ACSAC*, 2006.
- [13] T. Leek, G. Baker, R. Brown, M. Zhivich, and R. Lippmann. Coverage Maximization Using Dynamic Taint Tracing. Technical Report 112, MIT Lincoln Laboratory, 2007.
- [14] J. Lin, T. Chen, W. Hsu, P. Yew, R. Ju, T. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proc. PLDI*, pages 289–299, 2003.
- [15] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, pages 190–200, 2005.
- [16] S. Mahlke, W. Chen, R. Bringmann, R. Hank, W. Hwu, B. Rau, and M. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *TOCS*, 11(4):376–408, 1993.
- [17] W. Masri, A. Podgurski, and D. Leon. Detecting and Debugging Insecure Information Flows. In *Proc. ISSRE*, pages 198–209, 2004.
- [18] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. NDSS*, 2005.
- [19] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proc. ISC*, 2005.
- [20] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proc. ASPLOS*, pages 308–318, 2008.
- [21] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proc. RAID*, 2005.
- [22] 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. In *Proc. Micro*, pages 135–148, 2006.
- [23] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proc. POPL*, 2006.
- [24] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. ASPLOS*, pages 85–96, 2004.
- [25] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Otttoni, J. Blome, G. Reis, M. Vachharajani, and D. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proc. Micro*, pages 243–254, 2004.
- [26] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: Programmable Architectural Support for Efficient Dynamic Taint Propagation. In *Proc. HPCA*, 2008.
- [27] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. Usenix Security*, 2006.