# LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks

Feng Qin[§†]*, Cheng Wang[‡], Zhenmin Li[†], Ho-seop Kim[‡], Yuanyuan Zhou[†], and Youfeng Wu[‡]

[†] University of Illinois at Urbana-Champaign
[‡] Intel Corporation
[§] The Ohio State University

## Abstract

*Computer security is severely threatened by software vulnerabilities. Prior work shows that information flow tracking (also referred to as taint analysis) is a promising technique to detect a wide range of security attacks. However, current information flow tracking systems are not very practical, because they either require program annotations, source code, non-trivial hardware extensions, or incur prohibitive runtime overheads.*

*This paper proposes a* low overhead, software-only *information flow tracking system, called LIFT, which minimizes run-time overhead by exploiting* dynamic binary instrumentation and optimizations *for detecting various types of security attacks* without requiring any hardware changes. *More specifically, LIFT aggressively eliminates unnecessary* dynamic *information flow tracking, coalesces information checks, and efficiently switches between target programs and instrumented information flow tracking code.*

*We have implemented LIFT on a dynamic binary instrumentation framework on Windows. Our* real-system *experiments with two real-world* server *applications, one client application and* eighteen *attack benchmarks show that LIFT can effectively detect various types of security attacks. LIFT also incurs very low overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC INT2000 applications. Our dynamic optimizations are very effective in reducing the overhead by a factor of 5-12 times.*

## 1 Introduction

### 1.1 Motivation

Computer security is severely threatened by software vulnerabilities, such as buffer overrun, format string vulnerabilities, etc. These vulnerabilities allow malicious users to launch attacks by executing arbitrary code, causing denial of services, or stealing sensitive data on a vulnerable system. For example, the fastest-ever worm, Slammer worm in 2003, exploited the buffer overrun vulnerability in Microsoft SQL server. It brought down tens of thousands of machines within several minutes and cost hundreds of millions of dollars loss [30].

Although many tools or techniques such as StackGuard [16] and LibSafe [7] were proposed to detect some security attacks, they are far from effectively detecting attacks that exploit unknown software vulnerabilities because they are target for specific types of software vulnerabilities. As demonstrated by prior work [40], none of the five publicly available attack detection tools such as LibSafe [7] can detect all the designed eighteen types of security attacks, and even all of them combined together still leave 30% of attacks uncovered. Additionally, they provide little useful information regarding the attacks, for example, what are the attack input signatures, what are the attack steps, etc. This information is very useful for network-based applications to filter out future messages that match with attack signatures.

Several recent work [14, 34, 36] demonstrated that information flow tracking is a promising and effective technique for detecting many system-compromising security attacks that corrupts control data (e.g. return address, function pointer, etc.), even for exploitation of *unknown* types of software vulnerabilities. Generally this technique tags (labels) the input data from unsafe channels such as network connections as "unsafe" data, propagates the data tags through the computation (any data derived from unsafe data are also tagged as unsafe), and detects unexpected usages of the unsafe data that switch the program control to the unsafe data as exemplified in the stack smashing attack. In addition to the generality of attack detection, information flow tracking can also trace back to the input data that exploits the vulnerability to generate attack input signatures. This feature has been demonstrated to be very useful for effectively building a preventive network defense [14, 34].

* The work was done when Feng Qin worked at the University of Illinois at Urbana-Champaign

Table 1 shows an example to demonstrate the information flow tracking process. Initially, $a$ is received from the network, so it is unsafe. The second statement makes the information of $a$ flowing to $b$. When the program jumps to the location pointed by $c$, the system will raise an alarm if $c$ is unsafe.

| Target Program | Information Flow Tracking |
|---|---|
| receive (&a); | $Tag(a) = 1$ // unsafe as it is received from network |
| b=a; | $Tag(b) = Tag(a)$ |
| ... | ... |
| jmp c; | if ($Tag(c) == 1$), raise alert! |

**Table 1.** An example of information flow tracking

Additionally, information flow tracking can also be used in detecting information leaking (e.g. leaking passwords, etc) [31, 32, 39]. In this paper, we focus on detecting system-comprising security attacks, even though our technique can also be used for detecting information leaking with only a small modification.

So far information flow tracking has been implemented in three different ways. The first approach is track information flow at compile time for programs written in special type-safe programming languages [18, 19, 24, 31, 32]. While this approach can enforce the information flow security policies for programs without runtime overhead, it only works for programs that are written in the specific program languages and is therefore inapplicable to a large number of legacy programs written in type-unsafe languages such as C/C++. More importantly, due to lack of accurate runtime information, most tools in this category are designed for detecting sensitive information leaking instead of security attacks. For example, it is hard for them to detect attacks that alter the target of indirect branches, which can only be resolved at runtime.

The second approach is to track information flow and detect malicious exploits at runtime via either source code or binary code instrumentation. Source-code instrumentation-based information flow tracking, as done in Xu et al's work [42], has lower overhead than the alternative, binary-code instrumentation-based implementation, but it cannot track information flow in third-party library code and thereby will miss security exploits involving these libraries as reported in US-CERT [5]. Additionally, it requires programmers to provide a summary for each library function to allow the information flow through library calls, which can be an error-prone and tedious task as many library calls are fairly complex, causing many side-effects in addition to simple return values. In contrast, implementing information flow tracking via binary instrumentation, such as in TaintCheck [34], can track information accurately even in libraries, but suffers from a major overhead problem: it can slow down the program execution by around 37 times – too large to be used during production runs against security attacks.

The third approach of information flow tracking is to support it in hardware [17, 36, 39]. For example, the recent work RIFLE [39] and Suh et al's work [36] proposed new hardware extensions to track information flow for each instruction. While this approach is effective in detecting security attacks with low overhead, it requires non-trivial hardware extensions. Therefore, it is quite expensive and is not applicable to existing systems.

## 1.2 Our Contribution

This paper proposes a *low overhead*, *software-only*, *comprehensive* and practical information flow tracking system, called LIFT, that minimizes run-time overhead by exploiting *aggressive dynamic binary instrumentation and optimizations* for detecting various types of security attacks that corrupt control-data (e.g return address, function pointer, etc.) *without requiring any hardware changes*. Dynamic binary instrumentation and optimizations leverage accurate runtime information and enable more aggressive optimizations than static approaches at compile time. For example, at runtime we can eliminate many unnecessary *dynamic* information flow tracking when it can be sure that no unsafe data are involved in the computation.

More specifically, LIFT employs three runtime binary optimizations to minimize the overhead associated with information flow tracking for detecting general security attacks. The first optimization, referred as Fast Path (FP), eliminates unnecessary dynamic information flow tracking. This is based on the observation that, for most applications, the majority of computation involves safe data, for which it is unnecessary to track information flow. Therefore, by dynamically and efficiently performing a simple check before an execution region (e.g. basic block), LIFT can see whether involved data is safe or not; if it is, a fast binary version without any information flow tracking is executed; otherwise, the execution follows a slow version with detailed information flow tracking. By dynamically switching between fast and slow versions on demand, LIFT can effectively avoid unnecessary information tracking.

The second optimization, Merged Check(MC), further reduces the information flow checking overhead by coalescing data safety checks from multiple consecutive basic blocks into one. This optimization exploits both spatial locality and temporal locality of memory references because multiple safety checks for both nearby data and the same data are combined into one. It not only reduces the number of checks but also avoids bit operations because the safety of one byte is indicated by only one bit in the corresponding data tag. This optimization is applied to both consecutive basic blocks and dynamic instruction traces (i.e. dynamically-formed frequently executed code regions).

The last optimization, Fast Switch (FS), reduces the

overhead and number of context switches [1] between the target program and the information flow tracking code by using alternative cheaper instructions and status register liveness analysis, respectively. To avoid interference with the target program, most binary instrumentation frameworks such as PIN [29] and StarDBT [9] usually require saving/restoring some program execution context, including the status register and the runtime stack pointers, of the target program before and after executing the instrumented code (the reason will be discussed in details in Section 3.3). This context switch, even though much smaller than OS-level context switches, can still introduce large runtime overhead. LIFT minimizes this overhead by cleverly selecting cheaper instructions and performing condition register liveness analysis.

We have implemented LIFT based on a dynamic binary translator called StarDBT [9] on Windows. Our *real-system* experiments with two real-world *server* applications, one client application, and *eighteen* attack benchmarks [40] show that LIFT can effectively detect various types of security attacks that corrupt control data. LIFT also incurs very low overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC INT2000 applications. Our dynamic optimizations are very effective in reducing the overhead by a factor of 5-12 times.

Compared to previous approaches, LIFT provides several unique advantages:

- *Low-overhead.* Compared to other software-only binary-based information flow tracking system that slows down program execution by around 37 times, LIFT incurs significantly less overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC INT2000 applications, which indicate that LIFT is practical to use during production runs for detecting security attacks.

- *Not requiring any hardware extensions.* LIFT is a software-only approach based on dynamic binary instrumentation and optimization. Unlike previous hardware-based approaches [17, 36, 39] that require non-trivial hardware extensions, LIFT requires no hardware extension. Therefore, it can be used immediately in existing systems.

- *Not requiring source code.* Unlike source-level information flow tracking [42], LIFT works with binary code and thereby can work with commercial software whose source code is unavailable. Most importantly, it can performs accurate information flow tracking inside third-party library code and, consequently, can detect security vulnerabilities and exploits what occur inside these libraries.

[1]Note that the context switch here is *not* the OS-level context switch between different threads or kernel-user mode.

The rest of the paper is organized as follows. Section 2 describes the design and implementation of our basic information flow tracking system (LIFT-basic), followed by the three optimization techniques described in Section 3. Then Section 4 and Section 5 evaluate our work, followed by related work in Section 6. Finally we conclude in Section 7.

# 2 LIFT Basic Design and Implementation

LIFT tracks information flow at runtime via dynamic binary translation and optimization to detect general security attacks. Similar to other information flow tracking systems [14, 34, 36], LIFT dynamically instruments the binary of the target program to perform two tasks: (1) tracking information flow, and (2) detecting security exploits that switch the program control flow to unsafe data.

This section describes the basic design and implementation including the basic dynamic binary instrumentation framework, tag management, information flow tracking, exploit detection, an example of information flow tracking, and protection of tag space and LIFT code. The three dynamic optimizations for minimizing overhead will be described in the next section.

## 2.1 Dynamic Binary Instrumentation Framework

We build LIFT on top of a dynamic binary translator called StarDBT [9] developed by Intel. StarDBT automatically loads the original program code into memory and initializes the program execution context at program startup. Like other dynamic binary instrumentation and translation frameworks [6, 29, 37], StarDBT manages a code cache to store the translated code so that the original code is translated once and executed multiple times in order to amortize the translation cost. In addition, StarDBT collects profiling information to form hot traces of frequently executed code. More details about the basic dynamic binary translation and instrumentation framework can be found in [9].

At run time, LIFT uses StarDBT to instrument the translated code with instructions to perform information flow tracking and attack detection. Besides StarDBT, LIFT can also be built on top of other dynamic binary instrumentation tools or translators such as Dynamo [6], PIN [29], etc.

## 2.2 Tag Management

Similar to prior work [36], LIFT associates a one-bit tag (0 for "safe" data and 1 for "unsafe" data) for each byte of data in memory or general data registers. It can be easily extended to a multiple-bit tag for each byte as needed. For example, users may want to use different tags to express their trustiness for data from different sources such as network data, disk data, and other data. Using multiple-bit

tags can also reduce some overhead by avoiding bit operations in information flow tracking as demonstrated in prior work [42], but it significantly increases the space overhead for keeping tags and also increases processor cache pollution. Therefore, the prototype of LIFT uses one-bit tags.

LIFT stores the tags for memory data in a special memory region, called the *tag space*, via a one-to-one direct mapping between a tag bit and a memory byte in the target program's virtual address space. Such direct mapping makes it straightforward and fast (with only one memory access and a few arithmetic instructions) to get the tag value for a given memory location.

The current tag space incurs 12.5% space overhead. If the virtual memory space is limited, we can minimize the tag space using compression as memory data nearby each other usually have similar tag values: either all zeros or all ones. So we may keep only one value for the entire memory region (e.g. a page). Although this scheme saves memory space, it has extra runtime overhead for tag look-ups. Since the current prototype of LIFT is based on 64-bit architectures, where virtual memory space is seldom limited, we use the flat tag space management without compression.

LIFT stores the tags for general registers in a dedicated extra register to minimize overhead. Since register accesses are very frequent in program execution, the register tags are also accessed frequently. Therefore, for efficient register tag accesses, LIFT uses an extra 64-bit register to store the tags for all registers used in the target program. For architectures with no spare registers, we can use a special memory area to store tags for general registers. This will not significantly affect performance since most accesses to these tags will hit in the L1 cache.

At the beginning, all tags are cleared to zero. Based on the application-specific tagging policy, certain data (e.g. data read from the network or standard input) are tagged with 1 as "unsafe". As the program executes, other data may also be tagged with 1 via information flow. An unsafe data can become safe if its value is reassigned from some safe data.

## 2.3 Information Flow Tracking

As program executes, LIFT propagates the tag information from one data to another. It does this by dynamically instrumenting instructions with information flow track according to its type. For data movement-based instructions such as MOV, PUSH, POP, etc, the tag value of the source operand is propagated to the tag of the destination (e.g, if the source operand is unsafe, the destination also becomes unsafe). For arithmetic instructions, such as ADD, OR, etc, the corresponding tag values of the two source operands are OR-ed and the result is propagated to the tag of the destination operand since the information of the destination operand comes from both source operands. For instructions that involve only one operand, such as INC, etc, the tag of the operand does not change since the information of the operand flows to itself. Similar to previous work [14, 36, 39, 42], LIFT tracks information flows based on data dependencies but not control dependencies.

There are a few special instructions whose information flow tracking in LIFT does not follow the above general rules. For example, in x86 architecture, the instruction "XOR eax, eax" initializes the "eax" register to 0, therefore the tag value of "eax" should be reset to 0 ("safe" data). However, the general rule for this instruction keeps the tag of "eax" unchanged. To handle such cases, LIFT identifies these special instructions such as "XOR reg, reg" and "SUB reg, reg", and clear the tags of the corresponding registers or memory data.

In the baseline case (without any optimization described in the next section), the information flow tracking code is instrumented once at runtime and executed multiple times. The reason for instrumenting before instead of after an instruction in the original program is that execution of the instruction may change the operand address and thus make tag propagation more difficult.

## 2.4 Exploit Detection

In addition to information flow tracking, certain instructions are also instrumented to detect malicious exploits, i.e. improper usages of unsafe data that violate user-specified security policies. For example, "unsafe" data cannot be used as a return address or the destination of an indirect jump instruction, etc.

By default, similar to previous work [14, 36, 39], LIFT detects security attacks, regardless of the underlying security vulnerabilities, which use "unsafe" data for jump targets, return addresses, function pointers, or function pointer offsets. This allows LIFT to detect a wide variety of security attacks since the last step of many system-compromising security attacks requires directly or indirectly changing the program control flow to some unsafe data by altering the return address, function pointers, or general jump targets.

## 2.5 An Example of Information Flow Tracking for LIFT-basic

Figure 1 shows an example of information flow tracking instructions for three instructions (with bold font) from a target program. For different instruction type, the number of instrumented instructions for information flow tracking varies. For example, the first instruction moves a constant to a register, whose information flow tracking takes eight instructions, while the second and third instructions from the target program each requires twenty instructions for information flow tracking or exploit detection respectively.

We use the second instruction "ADD ebx, [ecx]" from the target program as an example to see how the informa-

| | MOV r10, gs:[30h] | MOV r10, gs:[30h] |
|---|---|---|
| | MOV r10, [r10+1488h] | MOV r10, [r10+1488h] |
| | MOV [r10−8], rsp | MOV [r10−8], rsp |
| | LEA rsp, [r10−8] | LEA rsp, [r10−8] |
| | PUSHFQ | PUSHFQ |
| | XOR r11, r11 | XOR r11, r11 |
| | LEA r11d, [ecx] | MOV r11d, ebx |
| | MOV r10d, r11d | MOV r10d, r11d |
| | SHR r11d, 3 | SHR r11d, 3 |
| | ADD r11, Tag_Space_Base | ADD r11, Tag_Space_Base |
| | MOV r13, [r11] | MOV r13, [r11] |
| | AND r10d, 0x07h | AND r10d, 0x07h |
| MOV r10, gs:[30h] | XCHG r10d, ecx | XCHG r10d, ecx |
| MOV r10, [r10+1488h] | SHR r13, cl | SHR r13, cl |
| MOV [r10−8], rsp | XCHG r10d, ecx | XCHG r10d, ecx |
| LEA rsp, [r10−8] | AND r13, 0x0fh | AND r13, 0x0fh |
| PUSHFQ | SHL r13, 0x04h | TEST r13, 0x0Fh |
| AND RegTag, 0xffffff0fh | OR RegTag, r13 | JNZ report_intrusion |
| POPFQ | POPFQ | POPFQ |
| POP rsp | POP rsp | POP rsp |
| **MOV ebx, 0x0400h** | **ADD ebx, [ecx]** | **JMP ebx** |

**Figure 1.** An example of information flow tracking for LIFT-basic. The instruction with bold font is an original instruction from the target program. The unbolded instructions instrumented *before* the bolded instruction perform information flow tracking.

tion flows. Instructions 1-5 do context switch, including switching to a different stack and saving the conditional flag register. Instructions 6-16 get the tag of the memory data "[ecx]". Instructions 17-18 propagate the tag of source operand in memory to the tag of destination operand in the register. The last two instructions restore the context.

### 2.6 Protection of Tag Space and LIFT Code.

In addition to overhead, another important concern is that LIFT code or the tag space can be corrupted by some program errors or carefully-crafted malicious inputs. Therefore, it is necessary to protect them. To protect the LIFT code against corruption, we use page protection to make the memory pages that store the LIFT code read-only. Thus, any attempt to modify the LIFT code causes a page fault.

To protect the tag space, we use a mechanism similar to prior work by Xu et al [42]. That is, we turn off the access permission of the pages that store the tag values of the tag space itself (note that the tag space is also a part of the virtual memory space, so there is a tag bit for each byte of the tag space). Thus any instruction in the original program or some hijacked code accessing the tag space results in information flow tracking that needs to access the corresponding tags and thereby triggers a protection fault.

## 3 LIFT Binary Optimizations

Section 2 described the baseline system of LIFT that does not have any optimizations. Similar to previous software-only information flow tracking systems [34, 14], it incurs large runtime overhead (up to 47 times as shown in our experiments). To minimize the overhead associated with information flow tracking so that it is practical to use during production runs against security attacks, LIFT employs three binary optimizations on top of the baseline system: (1) Fast Path (FP) that eliminates unnecessary information flow tracking, (2) Merged Check (MC) that merges multiple tag checks into one, and (3) Fast Switch (FS) that reduces the overhead incurred for switching between instrumented code and the original program.

All the above optimizations do not sacrifice the capability of detecting security attacks because they are all conservative: never eliminate any necessary tag propagations. In addition, they are all performed at the binary level so they work for software and libraries whose source code is unavailable. Even though it is possible to implement the third optimization, FS, via static instrumentation, the FP and MC optimizations benefit from the trace linking (also referred as hot traces) mechanism (each trace combines multiple basic blocks dynamically) available only in dynamic instrumentation frameworks. The following three subsections describe the three optimizations, respectively.

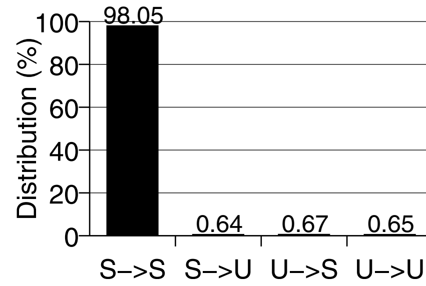



**Figure 2.** Distribution of four groups of tag propagation in Apache

### 3.1 Fast Path (FP) Optimization

The Fast-Path (FP) optimization is based on an observation that, for most server applications, majority of tag propagations are zero-to-zero, i.e., from safe data sources to a safe destination. To validate the above hypothesis, we collect some statistics of a running Apache web server. In the experiments, all data received from the network are tagged as one (unsafe). At run time, LIFT collect statistics on the distribution of different types of tag propagations: (1) $S \rightarrow S$: both the sources and the destination are safe; (2) $S \rightarrow U$: a safe data overwrites an unsafe data in the destination; (3) $U \rightarrow S$: the instruction propagates an unsafe data to a memory/register location that stores safe data. and (4) $U \rightarrow U$: the instruction propagates an unsafe data to a memory/register location that stores unsafe data.

As shown on Figure 2, majority of tag propagation belongs to the first type: $S \rightarrow S$. This is because, for most server applications, only data received from network are tagged as unsafe initially, and most other data that do not have data dependency on these data will remain safe, for at least *many execution periods (even though it may not be always safe for the entire execution)*. Therefore, any com-

putation among these safe data corresponds to zero-to-zero tag propagation.

The above observation provides a good dynamic optimization opportunity to eliminate unnecessary tag propagation. Specifically, before a code segment (either a basic block or a hot trace [9, 29])), we can insert some checks to see if all its live-in and live-out registers or memory data are safe or not. If so, then there is no need to do any information flow tracking inside this code segment. Checking the safety of all live-ins at the very beginning of a code segment is very intuitive as they are the source operands. We also need to check the safety of all live-out locations at the very beginning of a code segment because they may currently store unsafe data, and may be overwritten by some safe data inside this code segment, in which case it is necessary to do information flow tracking inside this code segment. There is no need to check other data because they are either not used in this code segment, or dead at the beginning or end of this code segment.

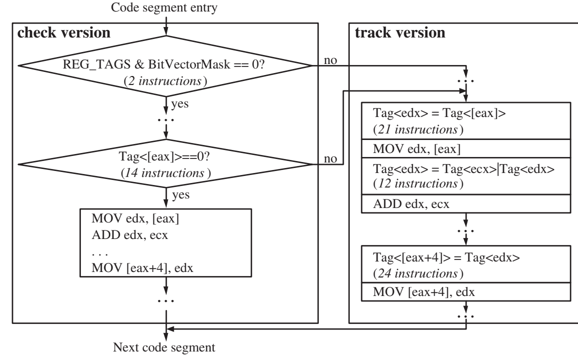



**Figure 3.** An example of the FP and MC optimizations. A code segment here can be a basic block or a hot trace, which is formed dynamically at runtime and can consist of multiple basic blocks.

The Fast-Path (FP) optimization is based on the above idea. It inserts information checks before entering a code segment. If all live-ins and live-outs are safe, it runs the fast binary version, referred as the *check version*, without any information flow tracking. Otherwise, it runs the slow version, referred as the *track version*, which performs information flow tracking. By *dynamically* switching between fast and slow versions, LIFT can effectively avoid unnecessary information flow tracking for *dynamic* code segments (dynamic instances of code segments) that do not involve any unsafe data. Since it always performs tag checks first to decide whether to run the track version, it does not affect the capability of detecting security attacks.

LIFT can easily check the tags for all the registers used in a program region. Essentially, it associates with each code segment a bit vector, called $BitVectorMask$, which records the live-in and live-out registers whose tags need to be checked. As demonstrated in Figure 3, at the beginning of a code segment, a check is inserted by performing an AND operation on the $BitVectorMask$ and $REG_TAGS$, which records the tags for all general data registers. If the result is zero, it follows the check version, otherwise it jumps to the track version.

Unfortunately, to know the memory live-ins and live-outs at the beginning of a code segment is much harder because some addresses may not be known at the beginning of the code segment. Therefore, as demonstrated on Figure 3, to handle memory data tags, LIFT postpones the information check of a memory location until its address is known, usually right before this memory instruction. If its tag is zero, it continues the check version; otherwise, it jumps to the corresponding instruction in the track version.

The granularity of a code segment can be either a basic block, or a hot trace which is formed dynamically at runtime and can consist of multiple basic blocks. At run time, if multiple basic blocks are frequently accessed one after another, the dynamic binary instrumentation engine will link them together by replacing indirect or conditional jump into a move and a direct jump. Then all these multiple basic blocks form a hot trace which is then stored in the trace cache. Obviously, it is better to perform the optimization at the trace granularity than at the basic block granularity because the former performs only one check for registers at the beginning of a trace and also provide opportunity for the next optimization, MC, to merge more memory tag checks into one.

The FP can be dynamically adapted to different behaviors of different code segments. For some code segments, the program may always execute the track version since there are always some "unsafe" data involved. For example, the functions that receive and directly process network data alway access "unsafe" data. After dynamically observing such behavior for some code segments, LIFT will retranslate these code segments to directly switch the program control to the track version and thus avoid useless checks in the check version.

### 3.2 Merged Check (MC) Optimization

Even after the FP optimization, many information checks are redundant or semi-redundant, which provides an opportunity for our second optimization: Merged Check(MC) optimization. MC reduces the number of information checks by combining multiple tag checks into one. Similar to the FP optimization, it is more beneficial to perform the MC optimization at the trace granularity.

To combine multiple checks into one, MC exploits both temporal locality and spatial locality of memory references commonly exhibited in many applications. Temporal locality says that a recently accessed data is likely to be accessed again in a near future, whereas spatial locality means

that after an access to a location, memory locations that are nearby are also likely to be accessed again in near future. To exploit the temporal locality characteristic, if a trace has multiple memory references to the same location, MC combines the tag checks and performs it only once right before the first memory reference. Secondly, MC exploits the spatial locality of memory references and merge multiple tag checks of nearby memory locations into one check.

To perform the optimization, MC needs to find ahead of time what memory accesses are to the same or nearby locations. It does this by performing memory reference analysis and then clustering the memory references into different groups. More specifically, MC first scans all the instructions in a trace and constructs a data dependency graph for each memory reference. The dependency graph for a memory reference consists of the version numbers of the registers and offsets for computing the address of this memory reference. It increments the version number of a register every time it is defined by an instruction explicitly or implicitly. For example, the stack operations may implicitly modify the stack pointer register. From these dependency graphs, MC can easily cluster the nearby/same memory references into a group. For example, if multiple references depend on the same version of the same register and the same offset, they are to the same memory location. And if their offsets differ by a small number, they are to nearby memory instructions. At the end, MC inserts one tag check before the first instruction of each group.

### 3.3 Fast Switch (FS) Optimization

In most general instrumentation frameworks such as PIN [29] and StarDBT [9], when the program execution switches between the original binary code and the instrumented code, i.e. the information flow tracking/checking code, it requires saving and restoring the context, including the condition register and the runtime stack registers (switching to a separate stack). The reason for saving the condition register before switching to the instrumented code is straightforward: the instrumented code may change the value of the condition register. The reason for using a separate stack for the instrumented code is for avoiding modifying the original data in the stack when executing the instrumented code. More explanation about this context switch requirement and process can be found in previous work [9, 29].

This context switch, even though much smaller than OS-level context switches, can still introduce large runtime overhead, especially the instrumentation is inserted at many locations as in our case for information flow tracking. LIFT minimizes the above switch overhead by cleverly selecting cheaper instructions and performing liveness analysis.

First, the FS optimization of LIFT reduces the overhead associated with each context switch. Similar to previous work [34, 14], the baseline system of LIFT saves/restores the context of the original code to/from the stack using simple but expensive $pushfq/popfq$ instructions in the x86 architecture. To make each context switch cheaper, LIFT uses two cheaper instructions $lahf\,/\,sahf$ to save/restore the condition register to other free registers. By eliminating $pushfq/popfq$, it also avoids the need of a separate stack for executing the information flow tracking/checking.

Second, with the FS optimization, LIFT performs condition register liveness analysis to eliminate those unnecessary condition register save and restore operations. In the x86 architecture, an *eflags* register saves the program conditional flags. Our liveness analysis tracks the define and use of eflags bits for each instruction within a program region of the original code. In many cases, the eflags register value is dead at the beginning of many program regions (e.g. instruction traces). Therefore, it is unnecessary to save it before switching to the instrumented code, i.e. the information tracking or checking code.

## 4 Evaluation Methodology

**Test Platform** Our experiments are conducted on real machines. The evaluated applications run on an EM64T machine with two 64-bit Xeon processors of 3.0GHz, 512KB L2 cache, and 1GB memory, running the Windows XP 64-bit version. For the network applications, we also use a second machine to act as the other party of the evaluated application. This machine has two Xeon processors of 2.2 GHz, 512KB L2 cache, and 512MB memory, runs Linux 2.6.9 and is connected to the EM64T machine via 100Mbps Ethernet network. We implement LIFT on StarDBT [9], a dynamic binary translator developed by Intel.

| Apps | Vers | Exploits | App Description |
|---|---|---|---|
| Apache | 1.3.24 | overwrite a function pointer | a web server |
| Savant | 3.1 | overwrite a return address | a web server |
| Putty | 0.53 | overwrite a return address | a telnet program |
| ATK | 2003 | 18 different types of exploits | a buffer overflow testbed |

**Table 2.** Applications and security exploits (Apps means applications, Vers means versions, ATK means attack benchmarks.

**Applications** We evaluate the functionality and performance of LIFT with a variety of applications, including three real-world network applications (two servers and one client) and two benchmark suites. The network applications include Apache Web server [1], Savant Web server [3] and Putty [38]. The first benchmark suite consists of eighteen different attack benchmarks developed by John Wilander [40] and covers a variety of different security exploits. We port the attack benchmarks from Linux version to Windows version. The second benchmark suite consists of seven SPEC INT2000 applications.

| Exploits Targets (Exploits #) | Detected #/Exploits # | | | | |
|---|---|---|---|---|---|
| | StackGuard | Stack Shield | ProPolice | LibSafe and Libverify | LIFT |
| Return Address (3) | 3/3 | 3/3 | 2/3 | 1/3 | 3/3 |
| Base Pointer (3) | 2/3 | 3/3 | 2/3 | 1/3 | 3/3 |
| Function Pointer (6) | 0/6 | 0/6 | 3/6 | 1/6 | 6/6 |
| Longjmp buffer (6) | 0/6 | 0/6 | 3/6 | 1/6 | 6/6 |
| Total (18) | 5/18 | 6/18 | 10/18 | 4/18 | 18/18 |

**Table 3.** Results of LIFT for attack benchmarks

To evaluate LIFT's capability in detecting general types of security attacks, we use three network applications as well as the eighteen attack benchmarks, as listed in Table 2. To play the real-world attacks for the three network applications, we leverage the Metasploit [2] framework to send the malicious inputs. The experiments cover a variety of different exploits, including overwrite function pointer, return address, etc. For example, the exploit in Savant Web server overwrites the return address in the stack. The attack benchmarks cover eighteen types of exploiting methods, including different overwrite techniques (direct or indirect), different buffer locations (stack or heap/BSS/data), and different attack targets (return address, base pointer, function pointer, or longjmp buffers).

For real-world network applications, we use Windows Layered Service Provider [25] technique to intercept network data and tag received data as "unsafe". This tagger works in the network layer and requires no source code of target programs. Since the attack benchmarks simulate network input, we have to modify the testbed to tag the simulated network input data as "unsafe".

To evaluate LIFT's overhead and the effects of our optimizations on latency and throughput, we use seven SPEC INT2000 benchmark and the Apache Web server. For Apache, we label all data received from network as "unsafe" and measure the throughput and response time. For SPEC benchmarks, we measure their performance with two input data tagging schemes: one tags all the input data from disk files as "unsafe" for simulating network data; the other tags all the input data from disk files as "safe" for measuring performance upper bounds.

# 5 Experimental Results

## 5.1 Security Attack Detection

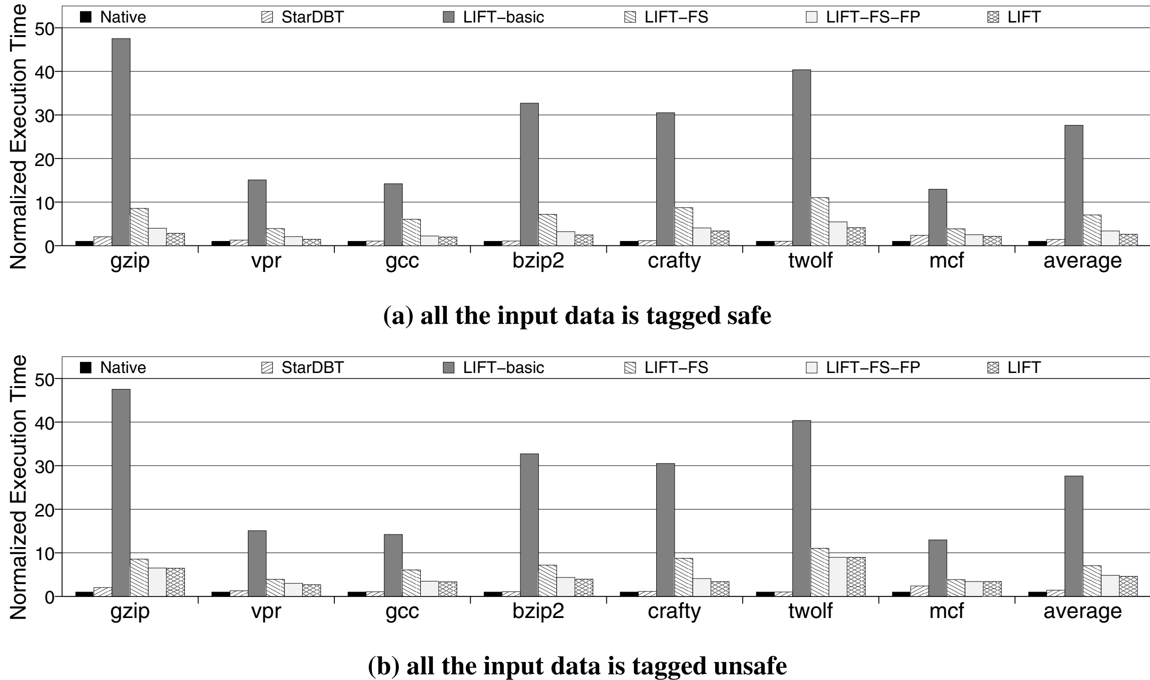
Table 3 shows the effectiveness of LIFT in detecting a wide range of security attacks that corrupt control data. We compare LIFT's results with those reported by prior work [40] that evaluated five existing tools, including StackGuard [16], Stack Shield [4], ProPolice [20], and LibSafe+LibVerify [7, 8], using the same eighteen attack benchmarks. We classify the eighteen types of attacks into four groups based on their exploiting targets, including return address, base pointer, function pointer, and longjmp buffer. Those targets can be either in the stack or heap/BBS/data regions.

| Configs | Throughput | | Response Time | |
|---|---|---|---|---|
| | RT (MBps) | OD (%) | RT (millisec) | OD (%) |
| Native | 8.06 | 0 | 1.1 | 0 |
| StarDBT | 7.79 | 3.4 | 1.5 | 36.4 |
| LIFT-basic | 6.40 | 20.6 | 5.1 | 363.6 |
| LIFT-FS | 6.97 | 13.6 | 3.5 | 220.0 |
| LIFT-FS-FP | 7.49 | 7.1 | 2.3 | 109.1 |
| LIFT | 7.56 | 6.2 | 2.1 | 90.9 |

**Table 4.** The throughput and response time of Apache running on native machine and with different optimization techniques applied (RT means results, OD means overhead). "Native" means that the Apache runs directly on the machine, "StarDBT" refers to our base line binary translation framework without any LIFT-related instrumentation. "LIFT-basic" is the basic LIFT system without any optimizations. "LIFT-FS" is LIFT-basic with the Fast-Switch optimization. "LIFT-FS-FP" is LIFT-basic with Fast-Switch and Fast-Path optimizations, and "LIFT" is LIFT-basic with all three optimizations. The requested file sizes are uniformly distributed among 4KB, 8KB, 16KB, to 512KB.

Overall, LIFT detects all the eighteen exploits of various types because it is oblivious to the specific exploit method such as smashing a return address, overwriting a function pointer, etc. All these exploit methods need to switch the program control to some "unsafe" data in order to hijack the program, so they are all detected by LIFT. In contrast, the other five tools shown in Table 3 can only detect some of the exploits since they are designed for certain types of exploits and cannot deal with other unknown exploits. For example, StackGuard and Stack Shield can only detect those attacks that try to smash a return address and a base pointer.

Our evaluation with three real-world network applications, including two popular Web servers (Apache and Savant) and one network client (Putty), shows that LIFT can also detect various types of attacks in real-world scenario. For example, the vulnerabilities in Savant Web server and Putty are exploited to overwrite the return address and switch the program control to some "unsafe" code. LIFT successfully reports these two attacks. For Apache, a long request overwrites the whole stack and triggers an exception when attempting to write beyond the stack bottom. The default signal handler in a system library fetches a function




**(a) all the input data is tagged safe**

**(b) all the input data is tagged unsafe**

**Figure 4.** Comparison of normalized execution time. “Native”, “StarDBT”, “LIFT-basic”, “LIFT-FS”, “LIFT-FS-FP”, and “LIFT” have the same meaning as in Table 4. All the input data from disk files are tagged safe in (a) and unsafe in (b).

pointer from the corrupted stack and switch the program control via that “unsafe” function pointer. In our experiments, we observe LIFT marking all the data in the corrupted stack as “unsafe” and theoretically it can catch this attack once the program control is switched via the “unsafe” function pointer. However, LIFT does not report this attack since the current version of StarDBT does not provide accurate exception handling. We are improving StarDBT on this issue.

LIFT raises no false alarms in all our experiments. We run LIFT normally with network applications such as Apache Web server and Putty and other small utilities such as Notepad without any false alarms reported. In addition, we tag all the input data from disk files for the tested SPEC-INT programs as “unsafe”, LIFT still runs through all the tests without raising any false alarms.

## 5.2 Performance Results

### 5.2.1 Overhead with Apache

Table 4 shows that LIFT incurs low runtime overhead for Apache. With all three optimizations, LIFT incurs only 6.2% for the throughput of Apache, close to 3.4% incurred by StarDBT. This is because LIFT aggressively applies dynamic optimization to eliminate unnecessary information flow tracking and provide a fast switch between the instrumented code and the original code. For example, the Fast Switch (FS) optimization reduces the overhead for the throughput from 20.6% to 13.6% and the overhead for the response time from 363.6% to 220%. The Fast Path (FP) optimization further improves the performance, bringing down the overhead for the throughput to 7.1% and the overhead for the response time to 109.1%.

The overhead of LIFT comes from several sources. The first is the StarDBT binary translation framework which incurs 3.4% overhead. The second source comes from the dynamic translating, instrumenting, optimizing and maintaining the binary code. The third source, the most significant one, is the overhead for executing the instrumented code to perform tag checks, tag propagation, and attack detection.

### 5.2.2 Overhead with SPEC INT Benchmarks

Figure 4 shows that LIFT incurs low runtime overhead for the seven SPEC programs. Benefited from the three optimizations, LIFT incurs 1.7-7.9 times overhead and an average of 3.6 times overhead when all the input data from disk files are tagged “unsafe”, much smaller than the large overhead (37 times slowdown) reported for a previous binary instrumentation-based information flow tracking tool [34]. This is because LIFT aggressively applies dynamic optimization to eliminate unnecessary information tracking and provide a fast switch between instrumented code and the original code.

Figure 4 also shows that the three optimizations effectively reduce the overhead incurred in the basic LIFT sys-

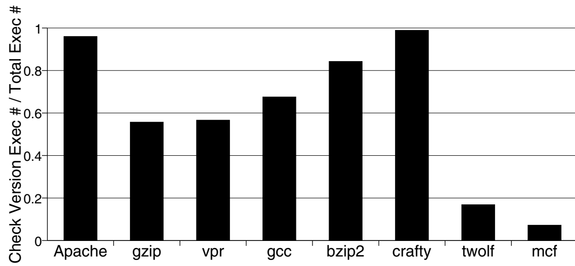



**Figure 5.** The check version execution percentage for SPEC. The execution number of the check version and the track version for basic blocks in Apache and SPEC. For SPEC, all the input data are tagged "unsafe". The total execution number is the sum of the execution number of both the check version and the track version.

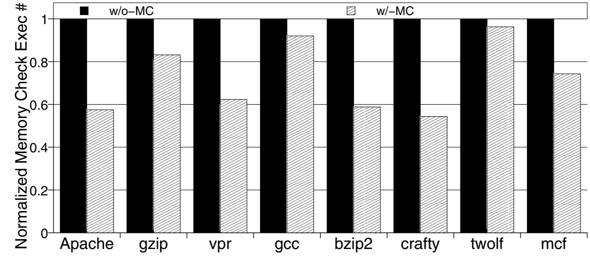



**Figure 6.** The execution number of memory checks for SPEC. All the input data are tagged as "unsafe". "w/o-MC" means run LIFT-FS-FP without the MC optimization applied. "w/-MC" refers to LIFT with all three optimizations including MC.

tem. For example, without any optimization, LIFT-basic slows down the program execution by 12.0-46.5 times and on average 26.6 times, which are effectively reduced by the three dynamic optimizations to an average of 3.6 times overhead, a factor of 7.4 times reduction in overhead!

With input data tagged "safe" or "unsafe", LIFT-basic shows no difference in terms of runtime overhead since the tag is propagated regardless it is "safe" or not. In contrast, LIFT, with all optimizations, does incur different overheads. For example, with all the input data tagged "safe" for vpr, LIFT incurs only 0.6 times overhead, and with all the input data tagged "unsafe" for vpr, it incurs 1.7 times overhead. This is because, if all the input data is tagged safe, LIFT always run the check version, which is much faster than the track version. Note here, even with all input data tagged "unsafe", LIFT does not necessarily always runs the track version since there still exists much computation that does not involve any "unsafe" data (because usually information flow tracking systems do not track control dependencies [14, 34, 36, 39]).

#### 5.2.3 Effects of Optimizations

Figure 4 shows that the three optimizations can effectively reduce the runtime overhead caused by LIFT-basic. Now let us examine the effect of each individual optimization. First, we apply the FS optimization to LIFT-basic since LIFT-basic has very frequent and heavy context switches. With all the input data tagged "unsafe", the FS optimization can reduce the overhead significantly by a factor of 4.4 times. This is because it reduces both the cost of each context switching by using cheaper instructions and the number of context switches by using eflags liveness analysis.

The FP optimization reduces the overhead incurred by LIFT-FS for all applications to different extent. For example, with all the input data tagged "unsafe", OPT-FP further reduces the overhead of LIFT-FS for crafty from 7.7 times to 3.1 times, while reducing the overhead of LIFT-FS for mcf from 2.9 times to 2.4 times. This is because the amount of overhead reduction depends on the percentage of check versions executed for each applications. As shown in Figure 5, with all input data tagged "unsafe", significant amount (98.9%) of the execution for crafty is in the check version, resulting in the large reduction of the overhead incurred by LIFT-FS, while mcf only has 7.2% of the execution in the check version.

MC further reduces the overhead of LIFT after the first two optimizations, FS and FP, for all cases. For example, the overhead of Apache's throughput decreased from 7.1% to 6.2% after applying MC. For SPEC applications the overhead reduction varies. For example, MC reduce the overhead for crafty from 3.1 times to 2.4 times, while it has no visible effects on the overhead for twolf (reducing from 7.97 times to 7.96 times). This is because the overhead reduction depends on how many executed memory checks are reduced and how many percentage the program executes the check version since only the check version contains memory checks. Figure 6 shows the normalized executed memory checks number with and without applying MC after for all the input data tagged as "unsafe". We can see that MC reduced 46% of executed memory checks for crafty, while it only reduced 4% of executed memory checks for twolf.

## 6 Related Work

Our work is related to many previous studies. Due to space limitation, this section only briefly describes closely related work that is not discussed in previous sections.

**Information Flow Tracking.** Information flow tracking systems can be implemented in three different ways, including language-based approaches [18, 19, 24, 31, 32], software-only approaches either in binary level [11, 14, 34] or in source level [42], or hardware extensions [17, 36, 39].

As discussed in Section 1, each approach suffers from its own limitations. In contrast, our LIFT is a low-overhead, comprehensive information flow tracking system that does not require any hardware extension and works for software and libraries whose source code are unavailable.

**Dynamic Binary Instrumentation and Optimization** There are many dynamic binary instrumentation tools, such as Dynamo [6], DynamoRIO [37], PIN [29], Valgrind [33], etc. They use similar techniques as StarDBT. They can be used for profiling, improving reliability [33], and software fault detection [35].

As for optimization, LIFT borrows some ideas from prior work. Computation reuse [28] and memorization [13, 12] save the computation results and reuse them later when the same computation is performed again. Unlike them, LIFT dynamically detects and eliminates unnecessary tag propagations. Some work such as DynamoRIO [37] and PIN [29] also found that save/restore eflags operations are very expensive. They, respectively, propose to leverage alternative cheaper instructions [10] and eflags liveness analysis [29] to reduce overheads. LIFT combines these two techniques in OPT-FS. Additionally, the other two optimizations, FP and MC, make FS more effective because the faster check version provides more opportunities to remove unnecessary save/restore eflags operations than the original track version.

**Security Attack Detection Methods** Some software-based dynamic methods detect certain types of vulnerabilities at runtime by statically or dynamically instrumenting source code [22, 26] or binary code [23, 33]. They incur large runtime overhead due to checking of each memory access. Therefore they are often used for in-house testing instead of production runs. Some software-based dynamic methods detect certain types of exploits at runtime via program transformation [15, 16] or library changes [7]. Unlike LIFT, these dynamic methods are limited to specific types of software vulnerabilities or exploits.

Several security attack detection or prevention methods, such as program randomization [41], program shepherding [27], and statistics-based intrusion detection [21], can also be used for detecting security attacks. For example, randomization shuffles program regions in memory during program load time and makes control flow jump to some bizarre point instead of the expected location and thereby causes program crash instead of being compromised. Statistical intrusion detection methods capture various invariants such as system call invariants and mark violations of those invariants as security attacks. Although these techniques are not limited by some specific types of software vulnerabilities, they provide little useful information regarding the attacks, for example, what are the attack input signatures, what are the attack steps, etc. Such information is very useful for effectively building a preventive network defense [14, 34].

## 7 Conclusion and Future Work

In summary, LIFT is a low-overhead, cheap (no hardware extension), comprehensive (works with libraries), and practical information flow tracking system for detecting a wide range of security attacks that corrupt control data (e.g. return address, function pointer, etc.). It minimizes runtime overhead by exploiting dynamic binary instrumentation and optimization including the Fast-Path, Merged-Check and Fast-Switch optimizations.

Our real-system experiments with three real-world network applications, including two Web servers and one client application, and *eighteen* attack benchmarks show that LIFT can effectively detect all tested 21 security attacks of various types, much more than five of the previous tools that can only detect at most ten attacks as shown in prior study [40]. More importantly, compared to other software-only binary-based information flow tracking system that slows down program execution by more than 40 times [34], LIFT incurs significantly less overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC2000 applications. The three optimizations also effectively reduce the overhead by a factor of 5-12 times.

We plan to extend and improve LIFT in several ways in our future work. First, we are in the process of further reducing LIFT's overhead by performing more aggressive optimizations such as learning from dynamic execution history to build a hybrid version that combines both information checks for some instructions and information tracking for others. Second, even though LIFT targets for detecting system-compromising security attacks, it can easily extend to detect information leaking in a way similar to previous information tracking systems such as RIFLE [39]. Third, similar to most previous information tracking systems, our current prototype does not support multi-threaded programs. Extending LIFT for multi-threaded programs is a challenging task and remains as our immediate future work. Finally, similar to previous work, LIFT does not track implicit information flow via control dependency since it is rarely exploited by security attacks.

## 8 Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. We appreciate useful discussion with the OPERA group members. This research is supported by Intel gift grant, IBM Faculty Award, NSF CNS-0347854 (career award), NSF CCR-0305854 grant and NSF CCR-0325603 grant.

## References


[1] Apache http server project. http://httpd.apache.org, 2006.

[2] Metasploit project. http://www.metasploit.com/, May 2006.

[3] Savant web server. http://savant.sourceforge.net/, May 2006.

[4] Stack shield – a "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/, 2006.

[5] US-CERT. http://www.us-cert.gov/, 2006.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In *PLDI*, Jun 2000.

[7] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[8] A. Baratloo, T. Tsai, and N. Singh. Libsafe: Protecting critical elements of stacks. White Paper, http://pubs.research.avayalabs.com/pdfs/ALR-2001-019-whpaper.pdf, 1999.

[9] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO*, 2006.

[10] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, EECS, MIT, 2004.

[11] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.

[12] D. Citron and D. Feitelson. Hardware memoization of mathematical and trigonometric functions. Technical report, Hebrew University of Jerusalem, Mar 2000.

[13] D. A. Connors and W. mei W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. In *Micro-32*, Nov 1999.

[14] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP*, 2005.

[15] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security*, Aug 2003.

[16] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, Jan 1998.

[17] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*, Dec 2004.

[18] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[19] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[20] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, 2006.

[21] J. Farshchi. Statistical-based intrusion detection. http://www.securityfocus.com/infocus/1686, Apr 2003.

[22] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, Jun 2002.

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

[24] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *POPL*, 1998.

[25] W. Hua, J. Ohlund, and B. Butterklee. Unraveling the mysteries of writing a winsock 2 layered service provider. In *Microsoft Systems Journal*, May 1999.

[26] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, May 1997.

[27] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security*, pages 191–206, 2002.

[28] B. Li, G. Venkatesh, B. Calder, and R. Gupta. Exploiting a computation reuse cache to reduce energy in network processors. In *2005 International Conference on High Performance Embedded Architectures&Compilers*, Nov 2005.

[29] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, Jun 2005.

[30] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[31] A. C. Myers. JFLow: Practical mostly-static information flow control. In *POPL*, 1999.

[32] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transaction on Software Engineering and Methodology*, (4):410–442, 2000.

[33] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electroic Notes in Theoretical Computer Science*, 89(2), 2003.

[34] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, Feb 2005.

[35] G. A. Reis, D. August, R. Cohn, and S. S. Mukherjee. Software fault detection using dynamic instrumentation. In *Proceedings of the Fourth Annual Boston Area Architecture Workshop*, Feb 2006.

[36] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI*, Oct 2004.

[37] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, 2003.

[38] S. Tatham. PuTTY: A free Telnet/SSH client. http://www.chiark.greenend.org.uk/˜sgtatham/putty/, May 2006.

[39] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. August. RIFLE:an architectural framework for user-centric information-flow security. In *MICRO-37*, Dec 2004.

[40] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, Feb 2003.

[41] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, 00:260, 2003.

[42] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, Aug 2006.