Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2003-004
MIT-LCS-TR-912

July 21, 2003

Secure Program Execution Via Dynamic Information Flow Tracking

G. Edward Suh, Jaewook Lee, David Zhang, and Srinivas Devadas

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Secure Program Execution via Dynamic Information Flow Tracking

G. Edward Suh, Jae W. Lee, David Zhang, Srinivas Devadas
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Massachusetts Institute of Technology
Cambridge, MA 02139
{suh,leejw,dxzhang,devadas}@mit.edu

ABSTRACT

We present a simple architectural mechanism called dynamic information flow tracking that can significantly improve the security of computing systems with negligible performance overhead. Dynamic information flow tracking protects programs against malicious software attacks by identifying spurious information flows from untrusted I/O and restricting the usage of the spurious information.

Every security attack to take control of a program needs to transfer the program's control to malevolent code. In our approach, the operating system identifies a set of input channels as spurious, and the processor tracks all information flows from those inputs. A broad range of attacks are effectively defeated by checking the use of the spurious values as instructions and pointers.

Our protection is transparent to users or application programmers; the executables can be used without any modification. Also, our scheme only incurs, on average, a memory overhead of 1.4% and a performance overhead of 1.1%.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous;
D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Design, Performance

Keywords

Buffer overflow, format string, hardware tagging

1. INTRODUCTION

Malicious attacks often exploit program bugs to obtain unauthorized accesses to a system. We propose an architectural mechanism called *dynamic information flow tracking*, which provides a powerful tool to protect a computer system from malicious software attacks. With this mechanism, higher level software such as an operating system can make strong security guarantees even for vulnerable programs.

The most frequently-exploited program vulnerabilities are buffer overflows and format strings, which allow an attacker to overwrite memory locations in the vulnerable program's memory space with malicious code and program pointers. Exploiting the vulnerability, a malicious entity can gain control of a program and perform any operation that the compromised program has permissions for. While hijacking a single privileged program gives attackers full access to the system, attacks to hijack any program that has access to sensitive information represent a serious security risk.

Unfortunately, it is very difficult to protect programs by stopping the first step of an attack, namely, exploiting program vulnerabilities to overwrite memory locations. There can be as many, if not more, types of exploits as there are program bugs. Moreover, malicious overwrites cannot be easily identified since vulnerable programs themselves perform the writes. Conventional access controls do not work in this case. As a result, protection schemes which target detection of malicious overwrites have only had limited success – they block only the specific types of exploits they are designed for or they are too restrictive and cannot handle some legitimate programs such as dynamically generated code.

To thwart a broad range of security exploits, we can prevent the final step, namely, the unintended use of I/O inputs. For example, in order to obtain full control of the victim process, every attack has to change the program's control flow to execute malicious code. Unlike memory overwrites through vulnerabilities, there are only a few ways to change a program's control flow. Attacks may change a code pointer for indirect jumps, or inject malicious code at a place that will be executed without requiring malevolent control transfer. Thus, control transfers are much easier to protect for a broad range of exploits.

We propose architectural support, called *dynamic information flow tracking* to track I/O inputs and monitor their use. In our approach, a software module in the operating system marks inputs from potentially malicious channels, i.e., channels from which malicious attacks may originate, as spurious. During an execution, the processor tracks the spurious information flows. On every operation, a processor determines whether the result is spurious or not based on the inputs and the type of the operation. With the tracked information flows, the processor can easily detect dangerous

ASPLOS'04, October 7–13, 2004, Boston, Massachusetts, USA.

uses of spurious values and trap to a software handler to check the use. For example, checking if an instruction or a branch target is spurious prevents changes of control flow by potentially malicious inputs and dynamic data generated from them.

Experimental results demonstrate our protection scheme is very effective and efficient. A broad range of security attacks exploiting notorious buffer overflows and format strings are detected and stopped. Our restrictions do not cause any false alarms for Debian Linux or applications in the SPEC CPU2000 suite. Moreover, our scheme only requires, on average, a memory overhead of 1.44% and a performance degradation of 1.1%. At the same time, our approach is transparent to users and application programmers.

We first describe our security model and general approach for protection in Section 2. Section 3 presents our protection scheme including architectural mechanisms to track spurious information flow at run-time. Practical considerations in making our scheme efficient are discussed in Section 4. We evaluate our approach in Section 5 and Section 6. We discuss the advantages of our scheme compared to related work in Section 7. Finally, we discuss possible extensions to our scheme and conclude the paper in Section 8.

2. ATTACK AND PROTECTION MODELS

This section describes our security attack model and the approach to stop the attacks. We first explain the general attack and protection models, and discuss specific types of attacks that we are focusing on in this paper. We also provide two examples to illustrate our approach.

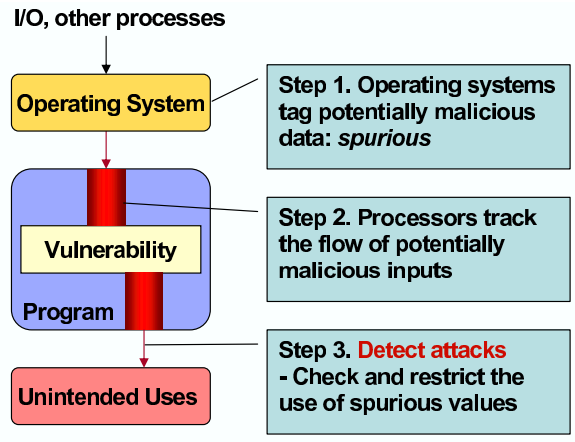

Figure 1: Attack and protection model.

Figure 1 illustrates security attacks and our protection approach in general. A program has legitimate I/O channels that are either managed by the operating system as in most I/O or set up by the operating system as in inter-process communication. An attacker can control an input to one of these channels, and inject a malicious input that exploits a vulnerability in the program. Note that we assume that the programs can be buggy and contain vulnerabilities, but they are not malicious. Thus, we do not consider the case when a back door is implemented as a part of the original program functionality.

The vulnerability in the program allows the malicious inputs to cause unexpected changes in memory locations which are not supposed to be affected by the inputs. Because the vulnerability is due to programming errors that are not detected by a weak type system, these unexpected values can be used not only as regular data but as any data type including code pointers such as function pointers and return addresses, data pointers, and even dynamic instructions that are not protected as being read-only. Once injected, the unexpected values can propagate into other locations even without a bug, generating more unexpected values.

Two frequently exploited vulnerabilities are buffer overflows [16] and format strings [15]. We give examples of how these vulnerabilities propagate malicious inputs in Sections 2.1 and 2.2.

Finally, the malicious instructions, pointers, and data are used by the victim process altering the process' behavior or results. The attacker can achieve various goals depending on how the unexpected malicious values are used:

- **Gaining total control**: The attackers obtain total control of the victim process if they can get to a shell with the victim's privilege. The victim process needs to execute the shell code, also called payload code, for this to happen. There are three possible ways to achieve this goal.

  First, attackers may inject instructions for the shell code, and make the victim process execute them. If existing instructions are overwritten, corrupting instructions may itself be enough to gain total control of the victim. Otherwise, this attack also requires corrupting a code pointer as described below.

  Second, attackers can corrupt code pointers such as function pointers and return addresses, which allows an arbitrary control transfer. Due to standard libraries, suitable payload code such as `execve()` often exists in the victim process' memory space even though the process does not use them. Thus, making a program jump to the existing payload code is often enough to obtain total control.

  Finally, for very special cases where the shell code is already used by the victim process, corrupting a data pointer and overwriting the appropriate data that determines whether the victim process executes the shell code or not may be used to gain total control.

- **Selecting a control path**: Corrupting data that is used to generate a branch condition can result in a malicious control transfer within the victim's original control flow graph. Note that unlike corrupting code pointers this attack can only change the path the victim process takes, and cannot make the victim execute arbitrary code.

  In order for this type of attack to cause damage, the attacker needs to be able to change *arbitrary* branch conditions. Therefore, this attack also needs to corrupt data pointers.

- **Corrupting sensitive data**: Attackers can corrupt sensitive data without changing the control flow. By changing data pointers, the attacker can corrupt arbitrary data in the victim's memory space and change the result computed from the data.

- **Crashing the process**: If denial of service is the goal, corrupting pointers to a random value can also be useful. The victim process will crash by accessing invalid memory addresses if either a code pointer or a data pointer is corrupted.

Note that all of these attacks require the victim process to unexpectedly use the value that depends on the malicious I/O inputs. Therefore, we can protect vulnerable programs by *identifying malicious I/O inputs*, *tracking the values* generated from the inputs, and *checking their uses*.

First, all values from potentially malicious I/O channels are tagged as *spurious* indicating the data is controlled by untrustworthy inputs. On the other hand, other instructions and data including the original program are marked as *authentic*.

During an execution, the processor tracks how the spurious values are used. If a new value is generated from the spurious ones, the processor marks the propagation by tagging the new value as spurious as well. We call this technique *dynamic information flow tracking*.

Finally, if the processor detects the suspicious use of spurious values, it generates a trap, which is handled by the software module in the operating system. For example, spurious instructions and pointers can cause a trap. Once the security trap occurs, the software module determines whether the use of the spurious value is legitimate or not. If the usage violates the system's or program's security policy, the victimized process is terminated by the operating system.

This general approach can be used to detect all types of attacks described above. In this paper, however, we focus on the attacks that try to take total control of the victim process as a primary example. Gaining total control of a vulnerable program is by far the most common and the most serious security threat.

As discussed above, there are only three possible ways for the attacker to obtain total control of a process. In most cases, preventing spurious instructions and spurious code pointers is enough to stop the attack. In special cases when the victim process itself uses the shell code, spurious data pointers for stores should be also prevented to stop attackers from overwriting critical data that determines whether to execute the shell code or not.

2.1 Example 1: Stack Smashing

A simple example of the stack smashing attack is presented to demonstrate how our protection scheme works. The example is constructed from vulnerable code reported for *Tripbit Secure Code Analizer* at SecurityFocus^{TM} in June 2003.

```
int single_source(char *fname)
{
    char buf[256];
    FILE *src;

    src = fopen(fname, "rt");

    while(fgets(buf, 1044, src)) {
        ...
    }

    return 0;
}
```

The above function reads source code line-by-line from a file to analyze it. The program stack at the beginning of the function is shown in Figure 2 (a). The return address pointer is saved by the calling convention and the local variable `buf` is allocated in the stack. If an attacker provides a source file with a line longer than 256 characters, `buf` overflows and the stack next to the buffer is overwritten as in Figure 2 (b). An attacker can modify the return address pointer arbitrarily, and change the control flow when the function returns.

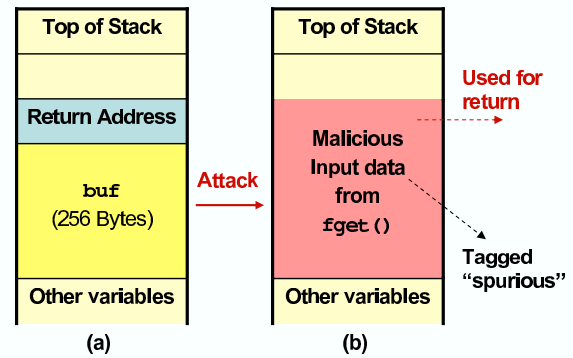

Figure 2: The states of the program stack before and after a stack smashing attack.

Now let us consider how this attack is detected in our scheme. When a function uses `fgets` to read a line from the source file, it invokes a system call to access the file. Since an operating system knows the data is from the file I/O, it tags the I/O inputs as *spurious*. In `fgets`, the input string is copied and put into the buffer. Dynamic information flow tracking tags these processed values as spurious (cf. *copy dependency* in Section 3.3). As a result, the values written to the stack by `fgets` are tagged spurious. Finally, when the function returns, it uses the `ret` instruction. Since the instruction is a register-based jump, the processor checks the security tag of the return address, and generates an exception since the pointer is spurious.

2.2 Example 2: Format String Attacks

We also show how our protection scheme detects a format string attack with `%n` to modify program pointers in memory. The following example is constructed based on Newsham's document on format string attacks [15].

```
int main(int argc, char **argv)
{
    char buf[100];

    if (argc != 2) exit(1);

    snprintf(buf, 100, argv[1]);
    buf[sizeof buf - 1] = 0;
    printf(``buffer: %s\n'', buf);

    return 0;
}
```

The general purpose of this example is quite simple: print out a value passed on the command line. Note that the code is written carefully to avoid buffer overflows. However, the `snprintf` statement causes the format string vulnerability because `argv[1]` is directly given to the function without a format string.

For example, an attacker may provide `''aaaa%n''` to overwrite the address `0x61616161` with 4. First, the `snprintf`

copies the first four bytes `aaaa` of the input into `buf` in the stack. Then, it encounters `%n`, which is interpreted as a format string to store the number of characters written so far to the memory location indicated by an argument. The number of characters written at this point is four. Without an argument specified, the next value in the stack is used as the argument, which happens to be the first four bytes of `buf`. This value is `0x61616161`, which corresponds to the copied `aaaa`. Therefore, the program writes 4 into `0x61616161`. Using the same trick, an attacker can simply modify a return address pointer to take control of the program.

The detection of the format string attack is similar to the buffer overflow case. First, knowing that `argv[1]` is from a spurious I/O channel, the operating system tags it as spurious. This value is passed to `snprintf` and copied into `buf`. Finally, for the `%n` conversion specification, `snprintf` uses a part of this value as an address to store the number of characters written at that point (4 in the example). All these spurious flows are tracked by our information flow tracking mechanism (cf. *copy dependency* and *store-address dependency* in Section 3.3). As a result, the value written by `snprintf` is tagged spurious. The processor detects an attack and generates an exception when this spurious value is used as a jump target address.

3. PROTECTION SCHEME

This section explains the details and the implementation of our protection approach presented in the previous section. We first describe how the protection functions are partitioned between the software layer (in the operating system) and the processor. Then, we discuss the *security policy* that controls the mechanism to efficiently detect malicious attack without generating false alarms.

3.1 Overview

Our protection scheme consists of three major parts: the *execution monitor*, the *tagging units* (flow tracker and tag checker) in the processor, and the *security policy*. Figure 3 illustrates the overview of our protection scheme.

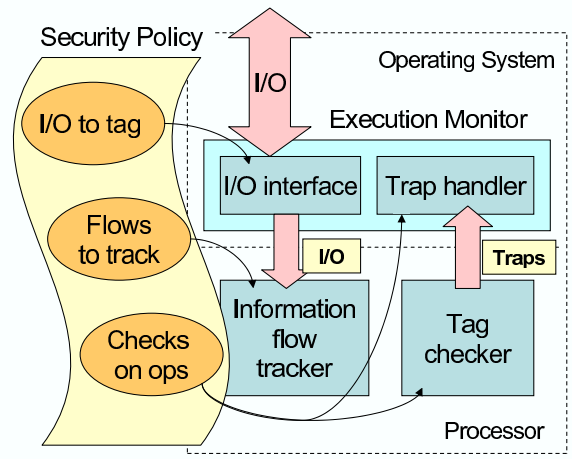

Figure 3: The overview of our protection scheme.

The execution monitor is a software module that orchestrates our protection scheme and enforces the security policy. First, the module configures the protection mechanisms in the processor so that they track proper information flows and trap on certain uses of spurious values. Second, the I/O interface in the module marks inputs from untrusted I/O channels as spurious. Finally, if the processor generates a trap, the handler checks if the trapped operation is allowed in the security policy. If so, the handler returns to the application. Otherwise, the violation is logged and the application is terminated. This module can be either in the operating system, or in a layer between the application and the operating system.

There are two mechanisms added to the processor core; *dynamic information tracking* and *security tag checking*. On each instruction, the information tracker determines whether the result should be spurious or not based on the authenticity of input operands and the security policy. In this way, the mechanism tracks spurious information flow. Section 3.3 describes flow tracking in detail.

At the same time, the tag checker monitors the tags of input operands for every instruction that the processor executes. If spurious values are used for the operations specified in the security policy, the checker generates a security trap so that the operation can be checked by the execution monitor.

The execution monitor and the two hardware mechanisms provide a framework to check and restrict the use of spurious I/O inputs. The security policy determines how this framework is used by specifying *the untrusted I/O channels*, *information flows to be tracked*, and *the restrictions on spurious value usage*. One can have a general security policy that prevents most common attacks, or one can fine-tune the policy for each system or even for each application based on its security requirements and behaviors. Section 3.5 discusses the issues in writing a security policy and describes our policy to prevent attacks from gaining total control of the victim process.

3.2 Security Tags

We use a one-bit tag to indicate whether the corresponding data block is authentic or spurious. It is straightforward to extend our scheme to multiple-bit tags if it is desirable to further distinguish the values; for example, it may be helpful to distinguish the I/O inputs from the values generated from them. In the following discussion, tags with zero indicate authentic data and tags with one indicate spurious data.

In the processor, each register needs to be tagged. In the memory, data blocks with the smallest granularity that can be accessed by the processor are tagged separately. We assume that there is a tag per byte since many architectures support byte granularity memory accesses and I/O. Section 4 shows how the per-byte tags can be efficiently managed with minimal space overhead.

The tags for registers are initialized to be zero at program start-up. Similarly, all memory blocks are initially tagged with zero. The execution monitor tags the data with one only if they are from a potentially malicious input channel.

The security tags are a part of program state, and should be managed by the operating system accordingly. On a context switch, the tags for registers are saved and restored with the register values. The operating system manages a separate tag space for each process, just as it manages a separate virtual memory space per process.

Operation	Control	Examples		Meaning	Tag Propagation
Pointer addition (reg+reg)	PCR[1:0]	ADD	R1, R2, R3	R1←<R2>+<R3>	Temp1←(T[R2]|T[R3])&PCR[0] Temp2←(T[R2]&[R3])&PCR[1] T[R1]←Temp1|Temp2
Other ALU operations	PCR[1]	MUL	R1, R2, R3	R1←<R2>*<R3>	T[R1]←(T[R2]|T[R3])&PCR[1]
		MULI	R1, R2, #Imm	R1←<R2>*Imm	T[R1]←T[R2]&PCR[1]
Load	PCR[2]	LW	R1, Imm(R2)	R1←Mem[<R2>+Imm]	Temp←T[Mem[<R2>+Imm]]; T[R1]←Temp|(T[R2]&PCR[2])
Store	PCR[3]	SW	Imm(R1), R2	Mem[<R1>+Imm]←<R2>	Temp←(T[R2])|(T[R1]&PCR[3]); T[Mem[<R1>+Imm]]←Temp
Jump & link	-	JALR	R1	R31←<PC>+4; PC←<R1>	T[R31]←0
Explicit tag manipulation					
Set a tag	-	SETT	R1, Imm	-	T[R1]←Imm
Move a tag	-	MOVT	R1, R2	-	T[R1]←T[R2]
Exceptional operations (architecture specific)					
Clear a reg	-	xor	eax, eax (x86)	eax←0	T[eax]←0

Table 1: Tag computations for different types of operations. <Ri> represents the value in a general purpose register Ri. Mem[] represents the value stored in the specified address. T[] represents the security tag for a register or a memory location specified. R0 is a constant zero register, and R31 is a link register.

3.3 Tracking Information Flows

In our context, a spurious value is the one that may have unexpectedly changed by I/O inputs due to bugs in the program. Once injected, spurious values can again cause unexpected changes to other values through many different dependencies. We categorize the possible dependencies for spurious information flows into five types: *copy dependency*, *computation dependency*, *load-address dependency*, *store-address dependency*, and *control dependency*.

- *Copy dependency*: If a spurious value is copied into a different location, the value of the new location is also spurious.
- *Computation dependency*: A spurious value may be used as an input operand of a computation. In this case, the result of the computation directly depends on the input value. For example, when two spurious values are added, the result depends on those inputs.
- *Load-address (LDA) dependency*: If a spurious value is used to specify the address to access, the loaded value depends on the spurious value. Unless the bound of the spurious value is explicitly checked by the program, the result could be any value.
- *Store-address (STA) dependency*: If stores use spurious data pointers, the stored value may become spurious because the program would not expect the value in the location to be changed when it loads from that address in the future.
- *Control dependency*: If a spurious value determines the execution path, either as a code pointer or as a branch condition, all program states are effectively dependent on that spurious value.

Processors dynamically track spurious information flows by tagging the result of an operation as spurious if it has a dependency on spurious data. To be able to enforce various security policies, the dependencies to be tracked are controlled by a bit vector in the Propagation Control Register (PCR). The PCR is set to the proper value by the execution monitor based on the security policy.

Table 1 summarizes how a new security tag is computed for different operations. First, the ALU operations can propagate the spurious values through computation dependency. Therefore, for most ALU instructions, the result is spurious if any of the inputs are spurious and PCR[1] is set indicating the computation dependency should be tracked.

Additions that can be used for pointer arithmetic operations are treated separately unless PCR[0] is set because they can often be used to legitimately combine authentic base pointers with spurious offsets. For special instructions used for pointer arithmetic (such as s4addq in Alpha), we only propagate the security tag of the base pointer, not the tag of the offset. If it is not possible to distinguish the base pointer and the offset, the result is spurious only if both inputs are spurious. If PCR[0] is set, the pointer additions are treated the same as other computations.

For load and store instructions, the security tag of the source propagates to the destination since the value is directly copied. In addition, the result may also become spurious if the accessed address is spurious and the corresponding PCR bits are set to track the load-address or store-address dependencies.

We introduce two new instructions so that software modules, either the execution monitor or the application itself can explicitly manage the security tags. The SETT instruction sets the security tag of the destination register to an immediate value. The MOVT instruction copies the security tag from the source to destination.

Finally, there can be instructions that require special tag propagations. In the x86 architecture, XOR'ing the same register is the default way to clear the register. Therefore, the result should be tagged as authentic in this case. Common RISC ISAs do not require this special propagation because they have a constant zero register.

Note that we do not track any form of control dependency in this work. We believe that tracking control dependency is not useful for detecting malicious software attacks consider herein. For control transfer that can compromise the program, such as register-based jumps, the use of spurious values should simply be checked and stopped. Tracking a control dependency will only make the entire program state

Operation	Example	Trap condition
Instruction fetch	-	T[Inst]&TCR[0]
Loads	LD R1, Imm(R2)	T[R2]&TCR[1]
Stores	ST Imm(R1), R2	T[R1]&TCR[2]
Register jumps	JR R1	T[R1]&TCR[3]
Cond. branches	BEQ R1, ofst	T[R1]&TCR[4]

Table 2: Security tag checks by the processor.

become spurious, causing traps on every consequent operation. We also believe that it is difficult for attacks to exploit control dependencies to bypass our protection scheme because programs do not use control dependencies to generate pointers (See Section 5).

3.4 Checking the Tags

For each instruction, the processor checks the security tag of the input operands and generates a trap if the spurious value is used for certain operations.

Table 2 summarizes the tag checks performed by the processor. The processor checks the five types of spurious values: *instructions*, *load addresses*, *store addresses*, *jump targets*, *branch conditions*. The Trap Control Register (TCR) that is set by the execution monitor based on the security policy determines whether a trap is generated.

In addition to the five checks that the processor performs, we add one explicit tag check instruction, `BRT R1, offset`. The instruction is a conditional branch based on the security tag. The branch is taken if the security tag of the source register is zero. Using this instruction, programs can explicitly check that critical values are not corrupted.

3.5 Security Policies

The security policy defines legitimate uses of I/O values by specifying the *untrusted I/O channels*, *information flows to be tracked* (PCR), *trap conditions* (TCR), and *software checks on a trap*. If the run-time behavior of a program violates the security policy, the program is considered to be attacked. Ideally, the security policy should only allow legitimate operations of the protected program.

The policy can be based either on a general invariant that should be followed by almost all programs or on the invariants for a specific application. Also, the restrictions defining the security policy can be based either on where spurious values can be used or on general program behavior.

For example, we use a general security policy generated from the following invariants based on the use of spurious values in this paper: *No instruction can be generated from I/O inputs*, and *No I/O inputs and spurious values based on propagated inputs can be used as pointers unless they are bound-checked and added to an authentic pointer*.

Our policy tags all I/O inputs except the initial program from disk as spurious. All spurious flows are tracked except that pointer additions are treated leniently (`PCR[0] = 0`) as discussed in Section 3.3. The processor traps if spurious values are used as instructions, store addresses, or jump targets. In this case, the handler does not have to perform any more checks since all traps indicate the violation of the security policy.

This exception in the flow tracking described above is made so as to not trap on legitimate uses of I/O inputs. For example, the `switch` statement in C, jump tables, and dynamic function pointer tables often use I/O inputs to compute the addresses of `case` statements or table entries *after explicit bound checking*.

```
ldl     r2, 56(r0)          # r2 ← MEM[r0+56)
cmpule  r2, 20, r3          # r3 ← (r2 <= 20)
beq     r3, default         # branch if (r3 == 0)
ldah    r28, 1(gp)          # Load ptr to table
s4addq  r2, r28, r28        # r28 ← r28 + 4*r2
ldl     r28, offset(r28)    # Load ptr
addq    r28, gp, r28        # r28 ← r28 + gp
jmp     (r28)               # go to a case
case 0: …
case 1: …
…                           Bound checking
default:
```

Figure 4: A switch statement with a jump table.

Figure 4 shows a code segment from a SPEC benchmark, which implements a switch statement. A potentially spurious value is loaded into `r2`, and checked to be smaller or equal to 20 (bound check). If the value is within the range, it is used to access an entry in a jump table; `r28` is loaded with the base address of a table, and `r2` is added as an offset. Finally, a pointer in the table is loaded into `r28` and used as a jump target.

This code fragment and other legitimate uses of spurious data do not produce traps under our policy. In the instruction `s4addq` in Figure 4, the spurious bit on `r2` is not propagated to `r28` because the original value of `r28` is authentic. Here we assume that adding non-constant offset to the base pointer implies that the program performed a bound-check before.

By restricting the use of spurious values as instructions, code pointers, and data pointers for stores, our policy can prevent attackers from injecting malicious code, arbitrary control transfers and arbitrary data corruption. Therefore, the attacks cannot obtain total control of the victim process.

While we crafted our policy so that a security trap indicates a violation of the security policy, traps may only indicate potential violations. For example, the policy can strictly track all dependencies, causing traps even on some legitimate uses. Then, the software handler can explicitly check if a bound-check has been performed or enforce any restrictions on the program behavior such as a) programs can only jump to the original code loaded from disk, or b) programs cannot modify a certain memory region based on spurious I/O, etc. We do not use these complex security policies to evaluate the security of our scheme because even the simple policy detects all attacks tested. The performance implication of tracking all dependencies is discussed in Section 6.

If more specific knowledge about the application behavior is available, stricter (and more secure) security policies can be enforced. For example, if the protected application does not use any jump tables, the security policy can strictly track all dependencies and flag all uses of spurious pointers. Also, if the application only copies the I/O inputs, but is not supposed to use them to determine its actions, conditional branches based on spurious values may be flagged.

4. EFFICIENT TAG MANAGEMENT

Managing a tag for each byte in memory can result in up to 12.5% storage and bandwidth overhead if implemented

Type value	Meaning
00	all 0 (per-page)
01	per-quadword tags
10	per-byte tags
11	all 1 (per-page)

Table 3: Example type values for security tags and their meaning.

naively. This section discusses how security tags for memory can be managed efficiently.

4.1 Multi-Granularity Security Tags

Even though a program can manipulate values in memory with byte granularity, writing each byte separately is not the common case. For example, programs often write a register's worth of data at a time, which is a word for 32-bit processors or a quadword for 64-bit processors. Moreover, a large chunk of data may remain authentic for an entire program execution. Therefore, allocating memory space and managing a security tag for every byte is likely to be a waste of resources.

We propose to have security tags with different granularities for each page depending on the type of writes to the page. The operating system maintains two more bits for each page to indicate the type of security tags that the page has. One example for 64-bit machines, which has four different types, is shown in Table 3.

Just after an allocation, a new authentic page holds a per-page tag, which is indicated by type value 00. There is no reason to allocate separate memory space for security tags since the authenticity is indicated by the tag type.

Upon the first store operation with a non-zero security tag to the page, a processor generates an exception for tag allocation. The operating system determines the new granularity of security tags for the page, allocates memory space for the tags, and initializes the tags to be all zero. If the granularity of the store operation is smaller than a quadword, per-byte security tags are used. Otherwise, per-quadword tags, which only have 1.6% overhead, are chosen.

If there is a store operation with a small granularity for a page that currently has per-quadword security tags, the operating system reallocates the space for per-byte tags and initializes them properly. Although this operation may seem expensive, our experiments indicate that it is very rare (happens in less than 1% of pages).

Finally, the type value of 11 indicates that the entire page is spurious. This type is used for I/O buffers and shared pages writable by other processes that the operating system identifies as potentially malicious. Any value stored in these pages is considered spurious even if the value was authentic before.

4.2 On-Chip Structures

Figure 5 illustrates the implementation of the security tag scheme in a processor. Dark (blue) boxes in the figure represent new structures required for the tags. Each register has one additional bit for a security tag. For cache blocks, we introduce separate tag caches (`T$-L1 and T$-L2`) rather than tagging each cache block with additional bits.

Adding security tags to existing cache blocks will require a translation table between the L2 cache and the memory in order to find physical addresses of security tags from physical addresses of L2 blocks. Moreover, this approach will require per-byte tags for every cache block, which is wasteful in most cases. Similarly, sharing the same caches between data and security tags is also undesirable because it would prevent parallel accesses to both data and tags unless the caches are dual-ported.

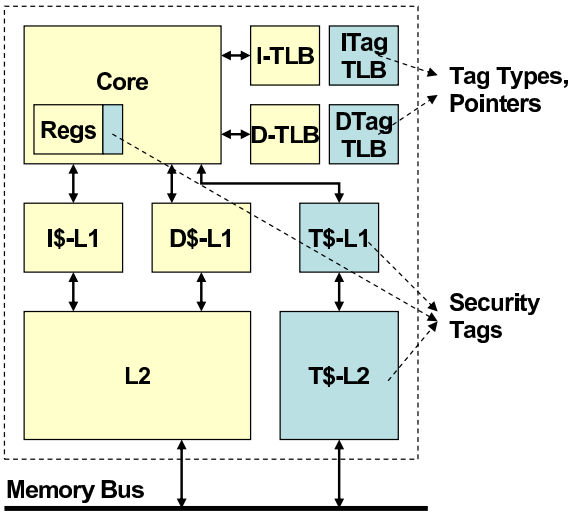

Figure 5: On-chip structures to manage security tags. Dark (blue) boxes represent additional structures.

Finally, the processor has additional TLBs for security tags. For a memory access, the tag TLB returns two bits for the tag type of a page. If the security tags are not per-page granularity tags, the TLB also provides the physical address of the tags. Based on this information, the processor can issue an access to the tag cache.

Note that new structures for security tags are completely decoupled from existing memory hierarchy for instructions and data. Therefore, latencies for on-chip instruction/data TLBs and caches are not affected. In Section 6 we discuss the impact of security tags on performance in detail.

5. SECURITY EVALUATION

This section evaluates the effectiveness of our approach in detecting malicious software attacks. There are two aspects to effectiveness. First, the protection scheme should be able to detect malicious attacks. On the other hand, the protection should not cause any false alarms without attacks.

We first evaluate our scheme through functional simulations of various applications. The result shows that our scheme with a simple security policy can effectively detect all attacks tested without false alarms. Then, we discuss the potential weaknesses of our scheme with the chosen security policy.

5.1 Simulation Framework

We developed two functional simulators to evaluate our scheme. For SPEC2000 benchmarks, `sim-fast` in the SimpleScalar 3.0 tool set [2] is modified to incorporate tagging.

To evaluate the effectiveness of our approach for real applications on the Intel x86 architecture, we modified the Bochs open-source x86 emulator [12]. We applied a 1-bit security

tag to every byte in memory and registers. Every byte on the hard drive is also tagged to accommodate paging and program load: on the first boot, all tags were authentic to represent authentic programs. Keyboard and network I/O was tagged as spurious to present standard sources of spurious data. At boot-time, all memory except for BIOS ROM was tagged as spurious.

As described in the security policy, copy, load/store address, and computation dependencies are tracked. In addition, several tag propagation rules specific to the x86 instruction set were added. First, for the instructions that are commonly used to zero a register such as `XOR R1, R1`, `SUB R2, R2` and `AND R3, 0`, the result is always tagged as authentic.

Also, if the immediate displacement is specified for memory addressing such as `[4*R1+Disp]` and larger than a 2-byte signed integer, the address is considered authentic. In the x86 architecture, the displacement can be 4-bytes and often used as a base pointer to jump tables for switch statements. For other addressing modes, the pointer addition rules described in Table 1 apply.

5.2 Experimental Validation

To evaluate the security of our scheme, we tested the testbed of 20 different buffer overflow attacks developed by John Wilander [24], and format string attacks based on the document from the TESO security group [20].

The buffer overflow testbed includes 20 different attacks based on the technique to overwrite (direct overwrite or pointer redirection), the locations of the buffer (stack or heap/BSS/data), and the attack targets (return address, base pointer, function pointer, and longjmp buffers). The testbed covers all possible combinations in practice. The original testbed simulated the input to the vulerable buffer internally rather than getting them from I/O. We modified the testbed so that the input is tagged as spurious.

The best protection tool tested in Wilander's paper only detected 50% of attacks. Our scheme detected and stopped all 20 attacks. Because all attacks injected either a code pointer or a data pointer, tracking copy and load/store address dependencies was sufficient to detect the attacks.

We also tested the format string attacks used in the cases of *QPOP 2.53*, *bftpd*, and *wu-ftpd 2.6.0*. *QPOP 2.53* and *bftpd* cases use a format string to cause buffer overflows, while the *wu-ftpd 2.6.0* case uses the technique described in Section 2.2. In all cases, the attacks were detected and stopped by copy and store-address dependencies.

The other concern for the effectiveness of a protection scheme is whether it causes false alarms without any attack. To verify that our scheme does not incur false alarms, we ran various real-world application on Bochs as well as 20 SPEC CPU2000 benchmarks [8] on `sim-fast`.

Under our security policy, Debian Linux (3.0r0) successfully boots and all the system commands including `ls`, `cp`, and `vi` can be run with no false alarms. We also successfully ran network applications such as the OpenSSH server and client, `scp`, and `ping` with no false alarms. Our scheme does not cause any false alarm for the SPEC benchmarks, either. Even though there are lots of data marked as spurious, those values from I/O are never used directly as an instruction or a pointer.

To test our protection scheme for dynamically generated code, we ran a sample http server, called TinyHttpd2, on SUN's Java SDK 1.3 HotSpot virtual machine that has a JIT (Just-In-Time) compiler. Our protection scheme did not cause any false alarms even when the input class files are marked as spurious. The generation of dynamic code typically involves control dependency, which is not tracked in our security policy.

5.3 Security Analysis

Our protection scheme detects *all* attacks that *directly* inject malicious code or pointers through I/O because copies are always tracked. All existing attacks that we know of fall into this category. In order to bypass our protection, attacks must find a way to make the victim process malicious inputs through untracked dependencies before being used.

Fortunately, tracking control dependency is not essential for detecting the type of attacks we consider. Attacks cannot cause specific changes to specific locations solely through control dependency. First, all attacks that inject a pointer to overwrite a specific location can be detected without tracking control dependency. Programs, even with bugs, only use additions and subtractions for manipulating pointers. Therefore, all injected pointers and values accessed using the pointers get marked as spurious if computation and load/store-address dependencies are tracked. Even when attacks do not need to inject pointers, control dependency alone often does not provide enough freedom for attacks to generate a specifc value that they need.

For example, let us consider the format string vulnerability discussed in Section 2.2. The vulnerability counts the number of characters to print before the `%n` directive, and overwrites the location pointed to by a value in the stack. If the attacker injects a data pointer in the stack, the overwrite is marked as spurious by store-address dependency. If there is a suitable pointer already in the stack, the attacker does not have to inject the malicious data pointer. However, even in this case, the malicious value will be marked as spurious by computation dependency. To generate a large enough value to be used as a pointer, the attacks need to use format directives such as `%256d` to specify the number of characters, which involves computation. Otherwise, the value can only be as large as the length of an input buffer.

The lenient propagation rule for pointer additions may leave some kinds of programs vulnerable. First, if I/O inputs are translated using a look-up table, the propagation is not tracked. This does not mean that all look-up tables can be exploited because the program normally will not overwrite sensitive information such as pointers with the result of a table look-up. However, it is conceivable that the table look-up combined with buffer overflow bugs can leave the program vulnerable.

Second, if the I/O inputs are added to an authentic value using the instructions for pointer additions, this spurious flow will not be tracked. Again, to be exploitable, this untracked propagation should be present with other bugs that cause the untracked values to overwrite pointers.

In summary, control dependency is not essential in detecting the type of attacks we are considering. On the other hand, the lenient treatment of pointer additions may leave some programs vulnerable. However, these vulnerabilities must be combined with other bugs to be exploitable. One way to enhance the security of our scheme is to identify explicit bound checks rather than assume that all pointer additions make I/O inputs safe.

6. PERFORMANCE EVALUATION

This section evaluates the memory space and performance overheads of our protection scheme through detailed simulations.

Architectural parameters	Specifications
Clock frequency	1 GHz
L1 I-cache	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line
L2 cache	Unified, 4-way, 128B line
L1 T-cache	8KB, 2-way, 8B line
L2 T-cache	1/8 of L2, 4-way, 16B line
L1 latency	2 cycles
L2 latency	Varies (from CACTI)
Memory latency (first chunk)	80 cycles
I/D TLBs	4-way, 128-entries
TLB miss latency	160 cycles
Memory bus	200 MHz, 8-B wide (1.6 GB/s)
Fetch/decode width	4 / 4 per cycle
issue/commit width	4 / 4 per cycle
Load/store queue size	64
Register update unit size	128

Table 4: Architectural parameters.

Our simulation framework is based on the SimpleScalar 3.0 tool set [2]. For the memory space overhead, `sim-fast` is modified to incorporate our information flow tracking mechanism. For performance overhead study, `sim-outorder` is used with a detailed memory bus model. The architectural parameters used in the performance simulations are shown in Table 4. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled on EV6 (21264) for peak performance.

For the experiments, we use the security policy described in Section 3.5 as the default case. All input channels are considered potentially malicious. During an execution, the processor tracks copy, computation, load-address, and store-address dependencies with the lenient propagation for pointer additions (`PCR[3:0]` is set to $1110_{(2)}$). We also simulated and reported the results for the most restrictive policy that tracks all dependencies strictly (`PCR[3:0]` is set to $1111_{(2)}$).

6.1 Memory Space Overhead

Dynamic information flow tracking requires small modifications to the processing core. The only noticeable space overhead comes from storing security tags for memory.

We now evaluate our tag management scheme described in Section 4 in terms of actual storage overhead for security tags compared to regular data. Table 5 summarizes the space overhead of security tags.

For our security policy, the amount of spurious data are very limited. As a result, most pages have per-page tags, and the space overhead of security tags is very small. Over 85% of pages have per-page tags, and the space overhead is only 1.44% on average. Even for the tagging policy generating the most spurious tags (`PCR[3:0]` $= 1111_{(2)}$), the space overhead on average is less than 3.7%. `ammp` is an exceptional case where many pages have per-byte tags even though there are very small amount of spurious bytes. In most pages, only 16–31 bytes out of 8 KB are spurious.

6.2 Performance Overhead

Finally, we evaluate the performance overhead of our protection scheme compared to the baseline case without any protection mechanism. For each benchmark, the first 1 billion instructions are skipped, and the next 100 million instructions are simulated. Experimental results (IPCs) in figures are normalized to the IPC for the baseline case without any security mechanisms.

Benchmark	Spurious data (%)	Tag Granularity (%)			Overhead
		Page	QWord	Byte	
ammp	0.33	9.68	0.12	90.20	11.28
applu	0.00	99.99	0.01	0.00	0.00
apsi	0.01	99.97	0.00	0.02	0.00
art	18.91	66.94	23.79	9.27	1.53
crafty	0.01	99.22	0.65	0.13	0.03
eon	0.00	99.94	0.05	0.01	0.00
equake	0.10	100.00	0.00	0.00	0.00
gap	0.03	99.72	0.06	0.22	0.03
gcc	0.18	81.88	0.43	17.69	2.22
gzip	33.30	70.04	29.87	0.09	0.48
mcf	0.00	99.99	0.00	0.01	0.00
mesa	0.08	99.85	0.00	0.15	0.02
mgrid	0.00	99.97	0.03	0.00	0.00
parser	8.76	41.99	6.37	51.65	6.56
sixtrack	0.24	99.03	0.69	0.28	0.05
swim	0.00	99.98	0.00	0.01	0.00
twolf	0.32	98.74	0.00	1.26	0.16
vpr	0.07	99.45	0.54	0.01	0.01
vortex	6.87	44.64	5.27	50.09	6.34
wupwise	0.01	99.96	0.01	0.03	0.00
avg	3.46	85.55	3.39	11.06	1.44

Table 5: Space overhead of security tags. The percentages of pages with per-page tags, per-quadword tags, and per-byte tags are shown. Finally, `Overhead` represents the space required for security tags compared to regular data. All numbers are in percentages. `PCR[3:0]` is set to $1110_{(2)}$.

Since our mechanism uses additional on-chip space for tag caches, we increased the cache size of the baseline case by the amount used by the tag caches in each configuration. The access latency for a larger cache is estimated using the CACTI tool [22]. For some benchmarks, increased cache latency results in worse performance, even though the cache is larger. In our experiments we report the worst-case performance degradation by choosing the baseline performance to be the maximum of the performances for the two different cache sizes.

Our protection scheme can affect the performance in two ways. First, accessing security tags consumes additional off-chip bandwidth. Table 6 shows the six applications in which the bus bandwidth is polluted by tag accesses the most significantly. The rest of applications have only negligible bandwidth consumption (less than 0.1%) by security tags. Second, in the simulation framework, we assume that the dispatch of an instruction waits until both data and security tags are ready. Therefore, memory access latency seen by a processor is effectively the maximum of the data latency and the tag latency. This is a rather pessimistic assumption since there is no dependency between the regular computation and the security tags; it is possible to have more complicated logic for tag computations that allow regular computations to continue while waiting for security tags.

Figure 6 shows the performance overhead of our scheme for various L2 cache sizes when `PCR[3:0]` is set to $1110_{(2)}$. In this case, the tag caches are always one-eighth of the corresponding caches for instruction and data. The figure demonstrates that the overhead is modest for all benchmarks for various cache sizes. There is only 1.1% degradation on average while the worst case degradation is 23% for `art`. The performance of `art` is very sensitive to the L2 cache size, and having 12.5% additional cache space significantly

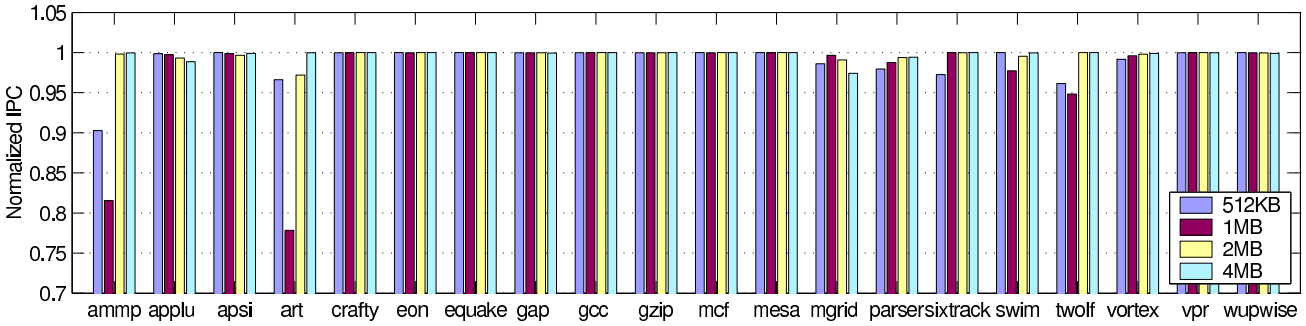

Figure 6: Performance overhead for various L2 cache sizes (1/8 tag caches).

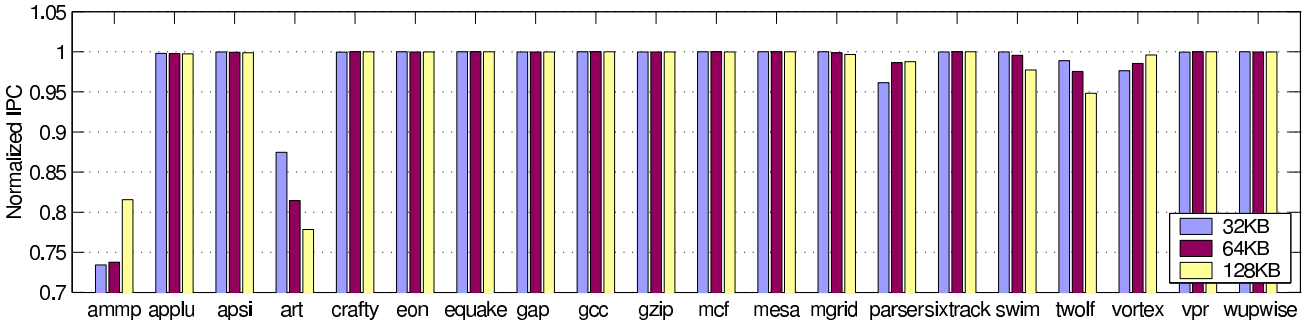

Figure 7: Performance degradation with small L2 tag caches (1-MB unified L2 cache).

Benchmark	Bandwidth Pollution (%)
ammp	24.86
vortex	11.83
parser	2.90
gzip	1.11
eon	0.83
gcc	0.21
average	**2.10**

Table 6: Bandwidth pollution defined as the extra bus bandwidth required for accessing security tags. The average is for all 20 benchmarks.

improves the baseline performance. If we consider the same size L2 caches for both baseline and our scheme, the performance degradation is less than 0.1% in the worst case. The performance degradation on average increases to 1.6% for the most strict tagging policy (`PCR[3:0]` $= 1111_{(2)}$).

Performance can also be affected by the size of tag caches. In the worst case, we should have a tag cache whose size is one-eighth of the corresponding data/instruction cache in order to avoid the additional latency caused by tag cache misses in case of data/instruction cache hits. However, given that we have only 1.44% overhead for security tags, small tag caches do not hurt the performance for most applications as shown in Figure 7.

We can categorize the behavior of those applications with variable L2 tag caches into three different classes: L2 data/instruction cache-limited, L2 tag cache-limited, and insensitive (to any L2 cache sizes). For the first class of applications such as `art`, `swim`, and `twolf`, their performance is limited by L2 *data/instruction* cache. For these applications, smaller tag caches result in less performance degradation, because the baseline case gets less additional data/instruction cache space. The second class of applications such as `ammp`, `parser`, and `vortex`, are sensitive to the size of L2 *tag* cache. We can easily identify them as the three applications which go through the largest amount of bandwidth pollution by tag accesses as shown in Table 6. In this class, a larger L2 tag cache leads to better overall performance, because the tag cache is a performance bottleneck. The third class of applications such as `crafty`, `eon`, `equake`, and so on, are not sensitive to either L2 data/instruction cache size or L2 tag cache size, so that their performance does not fluctuate for variable L2 cache sizes.

7. RELATED WORK

There have been a number of software and hardware approaches to provide automatic detection and prevention of buffer overflow and format string attacks. We briefly summarize some successful ones below, and compare them with our approach. In summary, our proposal provides protection against all known types of attacks without causing considerable overheads or requiring re-compilation.

7.1 Safe Languages and Static Checks

Safe languages such as Java, and safe dialects of C such as CCured [14] and Cyclone [9] can eliminate most software vulnerabilities using strong type systems and run-time checks. However, programs must be completely rewritten in a safe language or ported to safe C in order to take advantage of the safe languages. Moreover, the safe languages are often less flexible and result in slower code compared to C.

Various static analysis techniques are proposed to detect potential buffer overflows [7] or format string vulnerabilities [21]. While these techniques can detect many errors

at compile time, they all suffer from two weaknesses; they often cannot detect errors due to the lack of run-time information, and they tend to generate considerable false errors that need to be inspected by programmers. Run-time mechanisms are still required to protect undetected errors even after the programs are inspected by static analysis tools.

7.2 Dynamic Checks in Software

Many compiler patches are developed to automatically generate binaries with dynamic checks that can detect and stop malicious attacks at run-time. Early compiler patches such as StackGuard [5] and StackShield [23] checked a return address before using it in order to prevent stack smashing attacks. While these techniques are effective against stack smashing with near-zero performance overhead, they only prevent one specific attack.

PointGuard [4] is a more recent proposal to protect all pointers by encrypting them in memory and decrypting them when the pointers are read into registers. While PointGuard can prevent a larger class of attacks, the performance overhead can be high causing about 20% slowdown for openSSL.

Bound checking provides *perfect* protection against buffer overflows in instrumented code because all out-of-bound accesses are detected. Unfortunately, the performance overhead of this perfect protection while preserving code compatibility is prohibitively high; checking all buffers incurs 10-30x slowdown in the Jones & Kelly scheme [10], and checking only string buffer incurs up to 2.3x slowdown [18].

On top of considerable performance overhead, compiler patches have the following weaknesses. First, they need recompilation which requires access to source code. Therefore, they cannot protect libraries without source codes. Also, some techniques such as PointGuard require users to annotate the source program for function calls to uninstrumented code.

Program shepherding [11] monitors control flow transfers during program execution and enforces a security policy. Our scheme also restricts control transfers based on their target addresses at run-time. However, there are significant differences between our approach and program shepherding. First, program shepherding is implemented based on a dynamic optimization infrastructure, which is an additional software layer between a processor and an application. As a result, program shepherding can incur considerable overheads. The space overhead is reported to be 16.2% on average and 94.6% in the worst case. Shepherding can also cause considerable performance slowdown: 10-30% on average, and 1.7x-7.6x in the worst cases.

The advantage of having a software layer rather than a processor itself checking a security policy is that the policies can be more complex. However, a software layer without architectural support cannot determine a source of data since it requires intervention on every operation. As a result, the existing program shepherding schemes only allow code that is originally loaded, which prevents legitimate use of dynamic code. If a complex security policy is desired, our dynamic information flow tracking mechanism can provide sources of data that can be used by a software layer such as program shepherding.

7.3 Library and OS Patches

Library patches such as FormatGuard [3] and Libsafe [1] replaces vulnerable C library functions with safe implementations. They are only applicable to functions that use the standard library functions directly. Also, FormatGuard requires re-compilation.

Kernel patches enforcing non-executable stacks [6] and data pages [17] have been proposed. AMD and Intel also recently announced their plans to have architectural support for non-executable data pages. However, code such as `execve()` is often already in victim program's memory space as a library function. Therefore, attacks may simply bypass these protections by corrupting a program pointer to point to existing code. Moreover, non-executable memory sections can also prevent legitimate uses of dynamically generated code. Our approach detects the pointer corruption no matter where the pointer points to.

7.4 Hardware Protection Schemes

Recent works have proposed hardware mechanisms to prevent stack smashing attacks [26, 13]. In these approaches, a processor stores a return address in a separate return address stack (RAS) and checks the value in the stack on a return. This approach only works for very specific types of stack smashing attacks that modify return addresses whereas our mechanism is a general way to prevent a broad range of attacks.

Mondrian memory protection (MMP) [25] provides fine-grained memory protection, which allows each word in memory to have its own permission. MMP can be used to implement non-executable stacks and heaps. It can also detect writes off the end of an array in the heap. However, MMP implementations that place inaccessible words before and after every malloc'ed region incur considerable space and performance overheads. Even then, MMP cannot prevent many forms of attacks such as stack buffer overflows and format string attacks.

Our tagging mechanism is similar to the ones used for hardware information flow control [19]. The goal of the information flow control is to protect private data by restricting where that private data can flow into. In our case, the goal is to track a piece of information so as to restrict its use, rather than restricting its flow as in [19]. Although the idea of tagging and updating the tag on an operation is not new, the actual dependencies we are concerned with are different, and therefore our implementation is different.

8. CONCLUSION AND FUTURE WORK

The paper presented a hardware mechanism to track dynamic information flow and applied this mechanism to prevent malicious software attacks. In our scheme, the operating system identifies spurious input channels, and a processor tracks the spurious information flow from those channels. A security policy concerning the use of the spurious data is enforced by the processor. Experimental results demonstrate that this approach is effective in automatic detection and protection of security attacks, and very efficient in terms of space and performance overheads.

In this paper, we have only focused on attacks that try to take total control of a vulnerable program. However, our technique to identify spurious information flows can be used to defeat other types of attacks as well, and there are many other possible security policies to explore. For example, we can disallow reading or writing a security-sensitive memory segment based on spurious values, which provides additional privacy and integrity.

Currently, our main weakness arises from the fact that we use lenient tracking rule for pointer additions. Ideally, automatically detecting a bound-check, rather than pointer additions, and sanitizing the value after the bound-check will provide the security without incurring unnecessary traps. Either binary analysis or compiler extensions can identify bound checking operations. To indicate bound checking, the `SETT` instruction can be inserted to explicitly clear the security tag, or instructions can be annotated with a control bit that overrides the default tag propagation rules.

Our protection scheme is completely transparent to applications. However, exposing our mechanisms to the application layer can enable even stronger security guarantee. Because compilers and application programmers know exactly what the intended uses of I/O values are, they can provide the best security policy. We plan to investigate the possible use of a compiler technique with programmer annotation to enhance the security of our scheme.

Dynamic information tracking also has interesting applications for debugging. For example, programs can use our mechanisms to detect the illegal uses of *uninitialized* values as pointer or branch conditions by using the tags to indicate whether the value is initialized or not. In general, the debuggers can monitor how certain values get used in the program with very low overhead.

9. REFERENCES

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

[2] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[3] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities, 2001. In 10th USENIX Security Symposium, Washington, D.C., August 2001.

[4] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, 2003.

[5] 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 *Proc. 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, Jan. 1998.

[6] S. Designer. Non-executable user stack. `http://www.penwall.com/linux/`.

[7] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security*, 2003.

[8] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[9] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[10] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the* 3^{rd} *International Workshop on Automatic Debugging*, 1997.

[11] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. 11th USENIX Security Symposium*, San Francisco, California, Aug. 2002.

[12] K. Lawton, B. Denney, N. D. Guarneri, V. Ruppert, and C. Bothamy. Bochs user manual. `http://bochs.sourceforge.net/`.

[13] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the 2003 International Conference on Security in Pervasive Computing*, 2003.

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

[15] T. Newsham. Format string attacks. Guardent, Inc., September 2000. `http://www.securityfocus.com/guest/3342`.

[16] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov. 1996.

[17] PaX Team. Non executable data pages. `http://pageexec.virtualave.net/verb`pageexec.txt .

[18] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the* 11^{th} *Annual Network and Distributed System Security Symposium*, 2004.

[19] H. J. Saal and I. Gat. A hardware architecture for controlling information flow. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, 1978.

[20] Scut. Exploiting format string vulnerabilities. TESO Security Group, September 2001. `http://www.team-teso.net/articles/verb`formatstring .

[21] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proceedings of the* 10^{th} *USENIX Security Symposium*, 2001.

[22] P. Shivakumar and N. J. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, WRL Research Report, Feb. 2001.

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

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

[25] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, 2002.

[26] J. Xu, Z. Kalbarczjk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. In *Proceedings of the* 2^{nd} *Workshop on Evaluating and Architecting System dependability (EASY)*, 2002.