Secure Program Execution via Dynamic Information Flow Tracking

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

Abstract

Dynamic information flow tracking is a hardware mechanism to protect programs against malicious attacks by identifying spurious information flows and restricting the usage of 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 disallowing the spurious data to be used as instructions or jump target addresses. Our scheme is also transparent to applications because it does not require any modification of executables. Unlike software-only protection methods, which may cause significant overheads, our scheme only requires additional memory space of 4.5% and result in less than 2% performance degradation on average.

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. Since hijacking a single privileged program gives attackers full access to the system, vulnerable programs 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.

To be effective for a broad range of security exploits, attacks can be thwarted by preventing the final step, namely, the malevolent transfer of control. In order to be successful, every attack has to change a program's control flow so as to execute malicious code. Unlike memory overwrites, there are only a few ways to change program's control flow. Attacks may change a pointer to 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. The challenge is to distinguish malicious control transfers from many legitimate ones.

We make the observation that potentially malicious input channels, i.e., channels from which instructions (data) may not be safe to be executed (used as a jump target), are rather easy to identify. Operating systems manage input channels for a program, and can mark spurious inputs so that these inputs are not allowed to be used as instructions or jump targets. However, spurious input data are used in various ways at run-time to generate new spurious data that may result in malicious control transfers. Therefore, only restricting the use of spurious *input* data is not sufficient to prevent many attacks.

Dynamic information flow tracking is a simple hardware mechanism to track spurious information flows at run-time. 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 pro-

cessor can easily check whether an instruction or a branch target is spurious or not, which prevents changes of control flows 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. In most cases, our restrictions do not cause any false alarms for real applications in the SPEC CPU2000 suite. At the same time, the space and performance overhead of our scheme is minimal. It only incurs 4.5% memory space overhead and less than 2% performance degradation, on average.

We describe our attack model and general approach for protection in Section 2. Section 3 presents architectural mechanisms to track information flow and Section 4 details how these mechanisms are used under various security policies. Practical considerations in making our scheme efficient are discussed in Section 5, and our scheme is evaluated in Section 6. Finally, we compare our approach with related work in Section 7 and conclude the paper in Section 8.

2 Security Exploits and Protection

This section discusses the types of security exploits that we target for prevention, and gives an overview of our protection scheme. We present an attack model for targeted security exploits and explain our approach to stop attacks under this model. We also present a simple example.

2.1 Attack Model

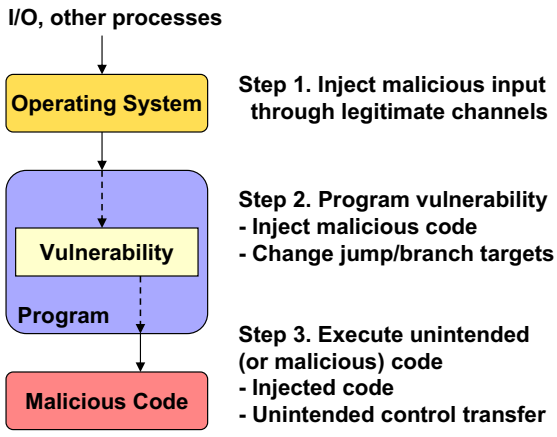

Figure 1. Our security attack model.

Figure 1 illustrates how attacks which attempt to take control of a vulnerable program work. A program has *legitimate* communication channels to the outside world, which are either managed by the operating system as in most I/O channels or set up by the operating system as in interprocess communication. An attacker can control an input to one of these channels.

Knowing a vulnerability in the program, attackers provide a malicious input that exploits the bug. This malicious input makes the program change values in its address space, in a way that is not intended in the original program functionality.

Two frequently exploited bugs are buffer overflows and format strings. The buffer overflow vulnerability occurs when the bound of an input buffer is not checked. Attackers can provide an input that is longer than an allocated buffer size, and overwrite memory locations near the buffer. For example, a stack smash attack can change a return address stored in the stack [10] by overflowing a buffer allocated in the stack.

The format string vulnerability [9] occurs when the format string of the `printf` family is given by input data. Using the `%n` flag in the format, which stores the number of characters written so far in the memory location indicated by an argument, attackers can potentially modify any memory location to any value.

Finally, the modified values in the memory cause the program to perform unintended operations. This final step of an attack can happen in two ways. First, attackers may inject malicious code exploiting the vulnerabilities and make the program execute the injected code. Second, attackers can simply reuse existing code and change the program's control flow to execute code fragments that otherwise would not have been executed by modifying one of the program pointers in the memory.

For example, in the stack smash attack, attackers inject malicious code into the overflown buffer as well as modify a return address in the stack to point to the injected code. When a function returns, the victim program jumps to the injected code and executes it.

2.2 Protection

We protect vulnerable programs from malicious attacks by restricting *executable instructions* and *control transfers*. In order to take control of a program, *every* attack should either make a processor execute injected malicious code or change a program's control flow to execute unintended code. Attackers may still be able to make programs produce incorrect results, for instance, due to a buffer overflow in Step 2 of Figure 1. However, they will not be able to gain unauthorized access to a system as long as executable instructions and control transfers are properly protected by blocking program execution in Step 3 of Figure 1.

The key question in this approach is how to distinguish malicious code from legitimate code, or malicious program

pointers from legitimate pointers. Because there are many legitimate uses of dynamically generated instructions such as just-in-time compilation, and legitimate uses of indirect jumps, the question does not have a straightforward answer.

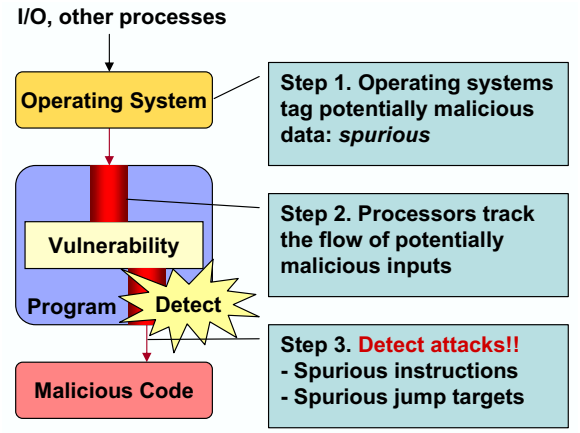

Figure 2. Our protection scheme against security exploits.

Figure 2 shows our approach to identify and prevent malicious instructions and control transfers. Since the operating system manages communication channels for a program, it identifies potentially malicious channels such as network I/O, and tags all data from those channels as *spurious*. On the other hand, other instructions and data including the original program when it gets loaded are marked as *authentic*. The operating system also specifies how the spurious instructions may be used. Normally, spurious instructions will not be allowed to be executed and spurious data will not be used as a jump target address.

During an execution, malicious data may be processed by the program before being used as an instruction or a jump target address. Therefore, the processor also tags the data generated from spurious data as spurious. We call this technique *dynamic information flow tracking*. There can be different types of dependencies, and the types of dependencies to be tracked can be specified as a part of a security policy (cf. Section 4).

Finally, if the use of spurious data or execution of spurious instructions violates a specified policy, the processor detects it and generates an exception, which will be handled by the operating system. In general, the exception indicates an intrusion, and the operating system needs to terminate the victimized process.

In the rest of the paper, we show that this approach is very efficient in detecting many types of security attacks without producing false positives.

2.3 Example 1: Stack Smashing

A simple example of the stack smashing vulnerability 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;
}
```

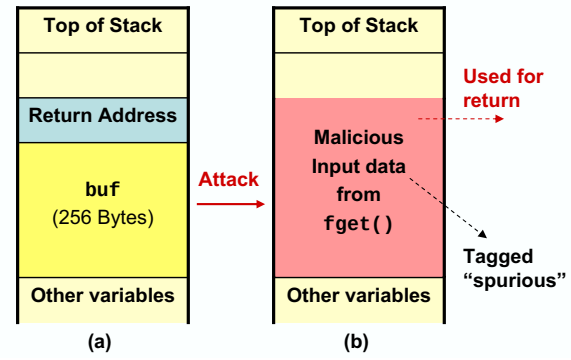

Figure 3. The states of the program stack before and after a stack smashing attack.

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 3 (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 3 (b). An attacker can modify the return address pointer arbitrarily, and change the control flow when the function returns.

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`, these

values may be copied and processed to be put into the buffer. Dynamic information flow tracking tags these processed values as spurious (cf. *computation dependency* in Section 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 pointer, and generates an exception since the pointer is spurious.

2.4 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 [9].

```
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 can provide `''aaaa%n''` to overwrite the address `0x61616161` with 4. First, the `snprintf` copies the first four bytes `aaaa` of the input into `buf`. Then, it encounters `%n`, which is interpreted as a format string to store the number of characters written so far in the memory location indicated by an argument. The number of characters written at this point is four, and without an argument specified, the pointer to `buf` is used as the argument. The first four bytes of `buf` has the value `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 conrol 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. *direct copy dependency* and *store dependency* in Section 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 branch target.

3 Dynamic Information Flow Tracking

The effectiveness of our protection scheme largely depends on the processor's ability of tracking flows of spurious data. An attack can be detected only if a malicious information flow is tracked by the processor. This section discusses the types of information flows that are relevant to attacks under our attack model and explains how they can be efficiently tracked in the processor.

3.1 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 there are many types or sources of data. However, since we only have to distinguish two types of data, one bit is sufficient for this particular setting. In the following discussion, tags with zero indicate authentic data and tags with one indicate spurious data.

For our purposes, the term *authenticity* is used to indicate whether the value is under a program's control or not. For example, a return address stored by the processor is under the program's control and safe to be used as a jump target. On the other hand, a program cannot predict a value from an I/O channel, and it will cause unpredictable behavior if the value is used as a jump target.

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 5 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 operating system 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 sepa-

rate tag space for each process, just as it manages a separate virtual memory space per process.

3.2 Information Flow Types

Spurious data can affect the authenticity of other registers or memory locations in many different ways. We categorize these dependencies into four types: *direct copy*, *computation dependency*, *load dependency*, and *store dependency*.

- *direct copy dependency*: If a spurious value is simply 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, in an arithmetic instruction `ADD Rd, Rs1, Rs2`, the value in `Rd` directly depends on the values of `Rs1` and `Rs2`. If either of the inputs are spurious, the output data is considered spurious.
- *load dependency*: When a spurious value is used to specify the address to access, the loaded value is considered spurious. Unless the bound of the value is explicitly checked by the program, the result could be any value since it is from an unpredictable address.
- *store dependency*: Just as in the load dependency, the stored value becomes spurious if the store address is determined by a spurious value. If a program does not know where it is storing a value, it would not expect the value in the location to be changed when it loads from that address in the future.

In general, a value can propagate to other locations through a control dependency as well. For example, in the following example, the value of `v2` is determined by the value of `v1`.

```
if (v1 == 1) {
    v2 = 1;
} else {
    v2 = 0;
}
```

However, we deem that the authenticity of `v2` is not affected by the authenticity of `v1` because the value of `v2` is completely controlled by the program itself (there are only two possible values). This is true for most control dependencies and hence we do not consider control dependency currently in our scheme.

In general, we note that a program can implement operations with spurious information flows so that the flow can only be detected through the control dependency. For example, many arithmetic operations such as additions and copying can be implemented with loops and comparisons which do not cause actual computation dependency or direct copy dependency. However, these implementations are extremely inefficient compared to the ones that use the instructions for computations or data movement. Therefore, reasonably good compilers are very unlikely to generate such programs.

There is one case that we know of that cannot be done efficiently with other dependencies and may cause spurious information flows: counting. For example, counting the number of zeroes in an array will require going through an array and checking each element. The result will only have control dependency from the array, which may contain spurious data. Fortunately, even attacks using this type of untracked data can be detected by other dependencies. In the format string attack (cf. Section 2.4), a spurious value is not tracked because it is generated by counting the number of characters written at a particular point. However, to use this value for an attack, the value should be written to the memory location specified by a spurious input. As a result, the store dependency detects the attack.

3.3 Tracking Information Flows

Processors dynamically track spurious information flows by tagging the result of an operation as spurious if it has a dependency from spurious data. If a spurious data is directly copied, a copy is always tagged as spurious. On the other hand, for flexibility, other dependencies to be tracked are specified by the operating system in a bit vector `Mask`. For example, computation dependencies are tracked only if the first bit in the vector `Mask[0]` is set. Similarly, `Mask[1]` and `Mask[2]` indicate whether the processor tracks the load dependency and the store dependency, respectively.

Table 1 summarizes how a new security tag is computed for different operations. For arithmetic or logical operations, the result is spurious if any of the inputs are spurious and computation dependency is specified to be tracked. For load or store operations, the security tag of the source always propagates to the destination since the value is directly copied. In addition, the result may also become spurious if the accessed address is spurious.

Some operations may use processor states or immediate values encoded in a instruction to obtain the result. For example, jump-and-link instructions update a register with the program counter plus four. In these cases, the processor states and immediate values are considered authentic. In our scheme a processor only executes authentic instructions, therefore the immediate values must be authentic. Processor states are authentic because attackers do not have any way to directly modify them.

Operation	Example		Meaning	Tag Propagation
Computation	ADD	R1, R2, R3	<R1>←<R2>+<R3>	T[R1]←(T[R2]|T[R3])&Mask[0]
	ADDI	R1, R2, #Imm	<R1>←<R2>+Imm	T[R1]←T[R2]&Mask[0]
Load	LW	R1, Imm(R2)	<R1>←Mem[<R2>+Imm]	Temp←T[Mem[<R2>+Imm]]; T[R1]←Temp|(T[R2]&Mask[1])
Store	SW	Imm(R1), R2	Mem[<R1>+Imm]←<R2>	Temp←T[R2]|(T[R1]&Mask[2]); T[Mem[<R1>+Imm]]←Temp
Branch/Jump	JALR	R1	<R31>←PC+4; PC←<R1>	T[R31]←0

Table 1. Tag computations for tracking each type of dependencies. <Ri> represents the value in a general purpose register. Mem[] represents the value stored in the specified address. T[] represents the security tag for a register or a memory location specified.

3.4 Implementation Complexity

It is trivial to implement our information tracking scheme in a processing core. In addition to security tags for registers, our scheme only requires a simple one-bit manipulation for computing a new tag, which can be done in parallel with the corresponding operation. Since this can be performed completely separately from regular operations, tag manipulation will not increase the latency of other operations.

4 Security Policies

The presented information tracking mechanism provides a powerful tool to make strong security guarantees. Potentially malicious data can be identified, and restricted to be used only for safe operations. *Security policies* specify what should be identified as spurious, and what operations are allowed (or not allowed) with the spurious data. In our scheme, the security policy consists of 3 parts: *spurious input channels*, *dependencies to be tracked*, and *restrictions*.

Good security policies are essential for the effectiveness of our protection scheme. If the policy is too restrictive, it may cause a lot of false alarms without intrusion. On the other hand, if the policy is too loose, many attacks will not be detected. This section discusses the possible design space and other considerations for security policies.

In this paper, we assume the security policy is specified in the operating system and enforced mostly by the processor. It is also possible to have other software layers such as program shepherding [8] to enforce more complicated security policies using information from the flow tracking mechanism. However, the flexibility provided by an additional software layer comes with increased space and performance overheads (cf. Section 7).

4.1 Input Channels

The security policy first needs to specify which input channels should be tagged as spurious. A program can have many different input channels such as network I/O, disk I/O, user interfaces, and shared pages among processes. For most privileged applications such as daemons, attacks are mainly from network I/O and it will be sufficient to tag the network input as spurious. However, it should be noted that no attack through an input channel can be detected unless the channel is specified as spurious.

4.2 Information Flow Types

The types of dependencies to be considered as spurious information flow should also be specified in the security policy. As discussed in the previous section, there are four dependencies, each of which will be needed to detect different types of attacks.

- *direct copy*: Simple attacks that directly modify values to be used as instructions or jump targets can be detected with just direct dependency. For example, simple forms of *stack smashing* attacks, which are some of the most popular buffer overflow attacks, will be detected.

- *computation dependency*: Detecting attacks that modify values before its malicious use requires that computation dependency be tracked. For example, if an overflown buffer gets processed before the value is used, the dependency from direct copy alone will not detect the attack.

- *load dependency*: Load dependency should be tracked for attacks that modify a data pointer. For example, an attack to corrupt a pointer in a double-linked list of `malloc` header is reported in [7]. When `free` is called, random memory locations are modified. In order to track this malicious flow, load dependency is required.

- *store dependency*: To detect `%n` format string attacks, store dependency needs to be tracked. In this attack,

a malicious value can be generated without visible dependency on spurious inputs. However, the location to store the value is given by the spurious inputs.

4.3 Restrictions for Use

To defeat attacks that attempt to take control of a program, we propose to restrict the use of spurious values as executable instructions and jump targets. These restrictions are enforced by a processor, and violations are reported as exceptions.

- *executable*: Do not allow spurious instructions to be executed. This restriction prevents attacks that inject malicious code into a program's address space.

- *jump target*: Do not allow spurious data to be used as a branch/jump target address. This stops attacks from changing the control flow.

The operating system can further restrict the use of spurious data in order to stop attacks with more subtle objectives than simply hijacking a program.

- *arguments of system calls*: Do not allow spurious data as arguments of selected system calls. On a system call, the operating system can verify if the arguments are authentic.

  This restriction can limit the impact of attacks even if they only modify data without attempting to change the control flow. For example, the operating system may not allow a program to make a system call `open` with spurious arguments. If there is a malicious attack that tries to corrupt the system's files by changing file names in the program, it will be detected.

- *program results*: Do not accept the results of a program if they are spurious. If channels that should not affect a certain result of a program can be identified, those inputs are tagged spurious. The result will be accepted only if tagged authentic. By checking the security tag, a user can be more confident of the integrity of the results.

5 Efficient Tag Management

Dynamic information flow tracking only requires minimal modification to the processing core as discussed in Section 3. On the other hand, managing a tag for each byte in memory can result in up to 12.5% storage and bandwidth overhead if implemented naively. This section discusses how security tags for memory can be managed efficiently.

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

Table 2. Example type values for security tags and their meaning.

5.1 Multi-Granularity Security Tags

Even though a program may 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 the entire 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 in each page table 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 2.

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 with 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 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.

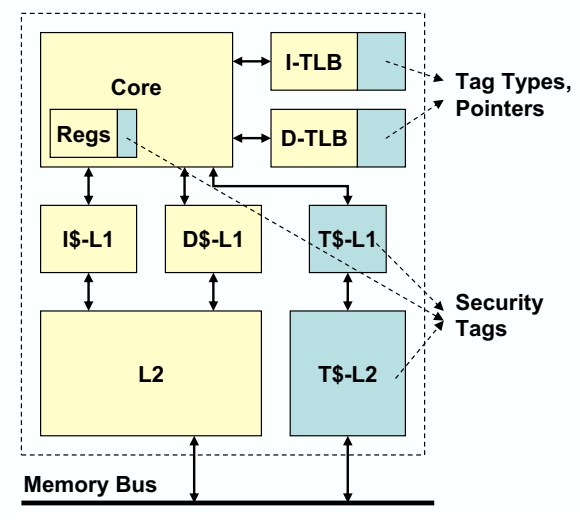

Figure 4. On-chip structures to manage security tags. Dark (blue) boxes represent additional structures.

5.2 On-Chip Structures

Figure 4 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 in the caches, 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.

TLBs contain the information about security tags in addition to regular virtual to physical translations. First, the 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 base address of the tags. Based on this information, the processor can issue an access to the tag cache.

6 Evaluation

This section evaluates our protection scheme through detailed simulations. We first study the functional effectiveness of the scheme, and then discuss memory space overhead and performance overheads.

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, 1MB, 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	10 cycles
Memory latency (first chunk)	80 cycles
I/D TLBs	4-way, 128-entries
TLB latency	160
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 3. Architectural parameters.

Our simulation framework is based on the SimpleScalar 3.0 tool set [1]. For the functional evaluation and 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 3. SimpleScalar is configured to execute Alpha binaries, and all benchmarks are compiled on EV6 (21264) for peak performance.

Policy	Tracked information flows
1	`Direct`
2	`Direct+Comp`
3	`Direct+Comp+LD`
4	`Direct+Comp+ST`
5	`Direct+Comp+LD+ST`
6	`Direct+LD`
7	`Direct+ST`
8	`Direct+LD+ST`

Table 4. Security policies used for simulations. `Direct`, `Comp`, `LD`, and `ST` represents direct copying dependency, computation dependency, load dependency, and store dependency.

We define eight security policies shown in Table 4 based on the types of information flows to be tracked, and use them in our simulations. In each security policy, all input channels to a program are considered potentially malicious. Thus, all input data from a system call are tagged spurious. Note that this makes each policy as conservative as possible, which implies potentially greater likelihood of false alarms and overheads. The use of spurious values is not allowed in

executable instructions and branch/jump target addresses.

6.1 Effectiveness

To evaluate the effectiveness of our approach in detecting malicious software attacks, we tested a set of benchmarks with various vulnerabilities such as buffer overflows and format strings.

- **Stack buffer overflows**: Stack smashing attacks based on a "cookbook" [10] and Tripbit Secure Code Analizer are detected and stopped by our protection scheme. The experiments show that all policies with direct copying dependency and computation dependency (Policy 2 - 5) successfully detect the attacks.

- **Heap buffer overflows**: Attackers can also exploit buffer overflows in the heap area. In most cases, the attack involves injecting malicious code in the heap. Thus, these attacks are stopped by not executing spuriuos instructions in our scheme with Policy 2 - 5. The following instances that are reported at SecurityFocus^{TM} are studied.

  WSMP3, *Tinyproxy*, and *Solaris xlock*: Attackers can inject the shell code and a program pointer into the heap by providing long inputs. Attackers can execute arbitrary code with effective privileges of the vulnerable programs.

  Null HTTPd: By passing a negative content length value to the server, attacks can modify the allocation size of the read buffer, which results in a heap overflow.

- **vudo**: As a heap buffer overflow attack, overwriting a file of a linked list of `malloc` is suggested [7]. The spurious values are tracked by load dependency and the attack is stopped by our scheme with Policy 3 or 5.

- **Format string attacks**: Format string attack based on Newsham's document [9] is studied. This attack is detected for the policies with store dependency (Policy 4 and 5) because it overwrites the memory location specified by spurious input.

The other concern for the effectiveness of a protection scheme is whether it causes false alarms without intrusion. To evaluate the false alarms, we simulated SPEC CPU2000 benchmarks [6] with our security policies. Each benchmark is simulated for at least 100 billion instructions to obtain the results.

Table 5 summarizes the number of false alarms that our scheme causes in SPEC CPU2000 benchmarks. For all policies, there are no false alarms from executing spurious instructions. All false alarms were due to using spurious values for branch/jump target addresses.

	Security policies							
Benchmark	1	2	3	4	5	6	7	8
ammp	0	0	0	0	0	0	0	0
applu	0	0	4	0	4	0	0	0
apsi	0	0	8	0	8	0	0	0
art	0	0	0	0	0	0	0	0
crafty	0	0	0	0	0	0	0	0
eon	0	0	16	0	16	0	0	0
equake	0	0	0	0	0	0	0	0
gzip	0	0	0	0	0	0	0	0
mcf	0	0	0	0	0	0	0	0
mesa	0	0	0	0	0	0	0	0
mgrid	0	0	4	0	4	0	0	0
parser	0	0	0	0	0	0	0	0
sixtrack	0	0	18	0	18	0	0	0
swim	0	0	4	0	4	0	0	0
twolf	0	0	0	0	0	0	0	0
vpr	0	0	2	0	2	0	0	0
wupwise	0	0	4	0	4	0	0	0
total	0	0	42	0	42	0	0	0
ave	0	0	2.5	0	2.5	0	0	0

Table 5. The number of false alarms for SPEC CPU2000 benchmarks with various security policies. All false alarms result from using spurious values as branch target addresses.

As shown in the table, there are no false alarms if load dependency is ignored (policy 1-2, 4, 6-8). The most restrictive policy (policy 5) can be enforced without false alarms for nine out of seventeen benchmarks. In general, the results demonstrate our security policies are reasonable to enforce for most programs.

The load dependency can result in false alarms because spurious data can be often used legitimately to access jump tables. For example, to implement `switch` statements in C, a compiler may generate code that simply checks the bound of the case value and converts it to an index into a jump table. In this case, the index is tagged spurious if the case value is spurious, and with load dependency a pointer from the jump table will also be tagged spurious.

Given the ways spurious values are used to load a program pointer, it is straightforward for compilers or operating systems to distinguish false alarms from legitimate uses. To enforce the restrictive policies with load dependencies for applications that have false alarms, compilers or operating systems can specify those particular loads as *legitimate* so that the spurious tag bit does not propagate. This will eliminate all the false alarms of Table 5.

6.2 Memory Space Overhead

Dynamic information tracking only requires minimal modifications to the processing core. The only noticeable space overhead comes from storing security tags for memory. This subsection evaluates our tag management scheme described in Section 5 in terms of actual storage overhead for security tags compared to regular data.

Table 6 summarizes the space overhead of security tags for two policies. Policy 1 has the least space overhead since it only tracks the direct copying dependency. On the other hand, Policy 5 has the most space overhead since it tracks all four types of dependencies.

For security policies without the computation dependency, the amounts of spurious data are often very limited. As a result, most pages have per-page tags, and the space overhead of security tags is almost negligible. For example, Policy 1 in the table results in over 97.5% pages with per-page tags and only 0.2% space overhead on average.

On the other hand, with the computation dependency tracked, the amount of spurious data is often significant. In fact, there may be more spurious data than authentic data as indicated by only 27% per-page tags. However, even in this case, most memory accesses are done in quadword granularity and the space overhead of tagging can be kept small. For example, ten out of seventeen benchmarks had less than 2% space overhead. On average, for Policy 5, the space overhead is 4.5% for the chosen SPEC benchmarks.

6.3 Performance Overhead

Finally, we evaluate the performance overhead of our scheme compared to the baseline case without any protection mechanism. Unlike the functional studies where we simulated seventeen benchmarks, we selected 8 benchmarks that have various characteristics of tag space overhead and memory bandwidth usage. For each benchmark, the first 1 billion instructions are skipped, and the next 100 million instructions are simulated. We will report results on a more complete set in the final version of this paper (if accepted) – however, the provided results are representative.

In the experiments, the same cache sizes are used for both our mechanism and the baseline case. We note that our scheme has on-chip logic overhead of additional tag caches. However, it is also not accurate to simply increase the caches for the baseline to compensate it because larger caches will have longer access latencies. At the same time, the simulation framework did not allow us to increase the cache size by just 12.5%. Therefore, we have ignored the difference in cache sizes. Given diminishing performance returns for larger caches, the error from this approximation is unlikely to be significant.

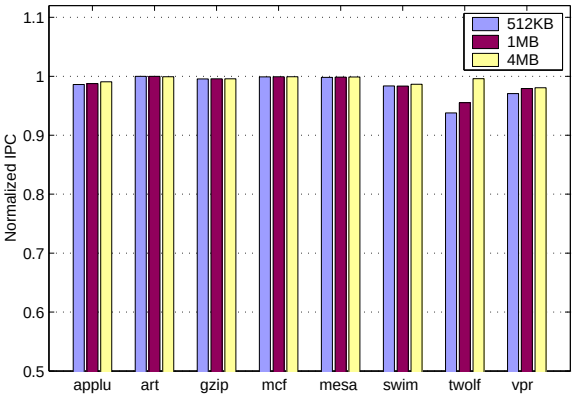

Figure 5. Performance overhead of our security mechanism over various L2 cache sizes. The worst-case performance degradation is about 6 percent (`twolf`).

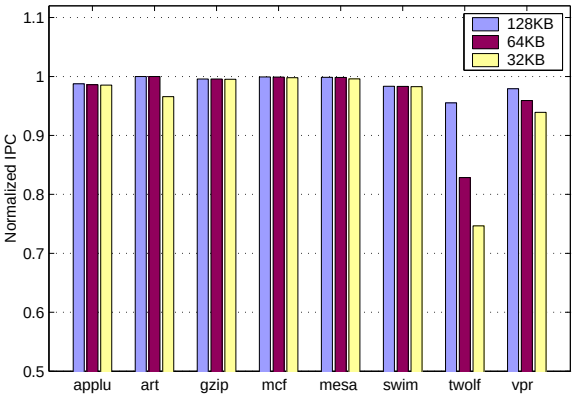

Figure 6. Performance degradation with small L2 tag caches. All IPCs were normalized to the IPC for the baseline case without any security mechanism (with 2 MB unified L2 cache).

Figure 5 shows that performance overhead of our mechanism is modest with 1.2 percent on average over various L2 cache sizes. Note that even the benchmarks causing large space overhead such as `twolf` and `vpr` have acceptable degradation of IPCs (about 3 % on average).

Performance is also 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 not to introduce a new overhead caused by tag cache miss even in case of data/instruction cache hit. However, our simulation results in figure 6 demonstrate that smaller tag caches works in most cases without imposing a significant performance cost. In the graph, we observe that IPCs are stable with the size of L2 tag cache in the range of 1/8 (128 KB) to 1/32 (32 KB) of the unified L2 cache size. Only when the program being executed has a large number of

Benchmark	Policy 1 (%)				Policy 5 (%)			
	Per-Page	Per-QWord	Per-Byte	Overhead	Per-Page	Per-QWord	Per-Byte	Overhead
ammp	99.85	0.00	0.15	0.02	1.58	4.58	93.84	11.80
applu	99.99	0.00	0.01	0.00	0.94	99.02	0.03	1.55
apsi	99.97	0.00	0.03	0.00	60.38	39.55	0.07	0.63
art	82.46	0.00	17.54	2.19	6.45	75.60	17.94	3.42
crafty	99.02	0.00	0.98	0.12	97.70	0.00	2.30	0.29
eon	97.97	0.00	2.03	0.25	79.73	6.76	13.51	1.79
equake	99.71	0.00	0.29	0.04	3.44	31.93	64.64	8.58
gzip	82.25	14.10	3.65	0.68	52.53	43.72	3.75	1.15
mcf	99.99	0.00	0.01	0.00	0.09	99.88	0.03	1.56
mesa	99.62	0.00	0.38	0.05	46.26	0.08	53.66	6.71
mgrid	99.94	0.00	0.06	0.01	1.61	98.27	0.12	1.55
parser	99.82	0.00	0.18	0.02	1.55	0.11	98.34	12.29
sixtrack	99.69	0.00	0.31	0.04	79.13	19.66	1.21	0.46
swim	99.98	0.00	0.02	0.00	0.49	99.47	0.04	1.56
twolf	98.60	0.00	1.40	0.18	22.11	5.61	72.28	9.12
vpr	99.74	0.00	0.26	0.03	1.04	1.47	97.49	12.21
wupwise	99.98	0.00	0.02	0.00	0.53	99.43	0.04	1.56
ave	97.56	0.83	1.61	0.21	26.80	42.66	30.55	4.48

Table 6. Space overhead of security tags. For each policy, 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.

per-byte tags as in `twolf`, the size of tag cache affects the performance significantly.

7 Related Work

There have been a number of other efforts to provide automatic detection and protection against buffer overflow and format string attacks. We summarize some of the successful ones for each type of possible approaches.

StackGuard [3], StackShield [14] are both compiler patches that are targeted to prevent stack smashing attacks. StackGuard places a "canary" next to a return address in a stack, and ensures that it's value is unchanged. Similarly, StackShield keeps a copy of the actual return address separately, and checks the address before using it. Both techniques only work for specific type of buffer overflow attacks that modify a return address in a stack, and require recompilation.

StackGhost [5] is a kernel patch that performs an `xor` operation on the return address before it is written to the stack and before it is used for control transfer. Return address corruption results in a transfer unintended by the attacker.

Enforcing non-executable permissions on IA-32 via kernel patches has been done for stack pages [4] and for data pages in PaX [11]. However, these techniques cannot be used for applications with legitimate use of dynamically generated code such as just-in-time compilation. Moreover, attacks can simply reuse existing code and bypass these protection techniques.

FormatGuard [2] is a library patch for eliminating format string vulnerabilities. It provides wrappers for the `printf` functions that count the number of arguments and match them to the specifiers. It is applicable only to functions that use the standard library functions directly, and it also requires recompilation.

Program shepherding [8] monitors control flow transfers during program execution and enforces a security policy. Our scheme also restrict 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 has high overheads. The space overhead is reported to be 16.2% on average and 94.6% in the worst case, compared to 4.5% and 12.5% in our case. Program shepherding also incurs up to 7.6X performance slowdown.

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 as part of a security policy in program shepherding.

Our tagging mechanism is similar to the ones used for information flow control [12]. 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 [12]. 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. Recent work uses a processor to remember and check a return address to thwart stack smashing and related attacks [15]. This approach only works for very specific types of stack smashing attacks that modify return addresses.

Static approaches for information flow control are proposed as a more powerful way of ensuring information flow security [13]. However, static analysis cannot be used for our purpose because attacks exploit program bugs at runtime.

8 Conclusion

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.

To enhance the security our mechanism, we plan to add some forms of control flow dependencies in the flow tracking. Specifically, simply tracking control dependencies within a loop will cover most cases that can generate spurious results.

In our current implementation, there are some legitimate use of jumps that depends on spurious data such as an implementation of `switch` statements. To be remain transparent to an application, automated analysis of executable code that can identify the legitimate jumps with potential false alarms is required. Our current scheme is also limited to only one source since it only has one-bit tag. Studies on more detailed security policies with multiple sources remain to be done.

We have only discussed how the information flow tracking can prevent attacks that try to take control of a vulnerable program. However, the technique to identify spurious information flow can be used to enhance other aspects of security such as data integrity. For example, the current approach only detects attacks with malicious control transfers. If we can disallow changing a security-sensitive memory segment based on spurious data, it will be also possible to protect the integrity of that segment. We plan to investigate other applications of information flow tracking with more complicated security policies. We believe that this is a promising direction for future research.

References

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

[2] 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.

[3] 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.

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

[5] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proc. 10th USENIX Security Symposium*, Washington, D.C., Aug. 2001.

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

[7] M. Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 8(57), Aug. 2001.

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

[9] T. Newsham. Format string attacks. Guardent, Inc., September 2000. `http://www.guardent.com/docs/FormatString.PDF`.

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

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

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

[13] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.

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

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