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. We describe two different security policies that track differing sets of dependencies. Implementing the first policy only incurs, on average, a memory overhead of 0.26% *and a performance degradation of* 0.02%*. This policy does not require any modification of executables. The stronger policy incurs, on average, a memory overhead of 4.5% and a performance degradation of* 0.8%*, and requires binary annotation.*

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 a 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 malicious attacks may come, are managed by operating systems. Therefore, operating systems can mark inputs from those channels as spurious so that they are not allowed to be used as instructions or jump targets. Unfortunately, spurious inputs 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 processor 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. Our restrictions do not cause any false alarms for applications in the SPEC CPU2000 suite. We describe two different security policies that track differing sets of dependencies. Implementing the first policy only requires, on average, a memory overhead of 0.26% and a performance degradation of 0.02%. The stronger policy requires, on average, additional memory space of 4.5% and a performance overhead of 0.8%. The stronger policy also requires annotation of executables prior to execution.

We describe our security model and general approach for protection in Section 2 and Section 3, respectively. Section 4 presents architectural mechanisms to track spurious information flow at run-time. The two security policies we consider are also defined. Practical considerations in making our scheme efficient are discussed in Section 5. We evaluate the first security policy in Section 6. Our second security policy is described in detail and evaluated in Section 7. Finally, we compare our approach with related work in Section 8 and conclude the paper in Section 9.

2 Security Attack Model

In this paper, we consider attacks whose goal is to gain unauthorized access to a computer system by taking control of a vulnerable privileged program. Security attacks can also try to crash programs, make programs produce incorrect results, read program's execution state, etc. However, attackers will not be able to obtain unauthorized access unless they hijack a privileged program. Therefore, we focus on this specific type of attacks.

We assume that attackers can exploit a vulnerability that allows them to modify an arbitrary memory location with an arbitrary value. Thus, attackers effectively have write permission to any stored program address. The only restriction for attackers is that any initial input from attackers should be through a communication channel that can be identified by an operating system. This is a reasonable assumption since all I/O channels are managed by the operating system in modern computer systems.

Protected programs and compilers that generated the programs are assumed to not be malicious. For example, we do not prevent programs from being compromised if a back door is implemented as a part of original program functionality. The protected programs, however, can be buggy and contain vulnerabilities. To achieve the goal of taking control, attackers should either change a control flow of a program in an unintended way, or inject its own code.

Given the above assumptions, our protection scheme targets the prevention of any attack that tries to take control of a protected program. In the rest of the section, we explain how security attacks work in more detail. In the next section, we show how we can defeat the attacks.

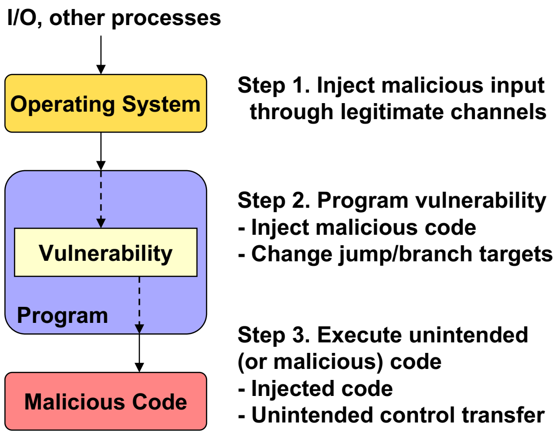

I/O, other processes
Operating System
Step 1. Inject malicious input through legitimate channels
Vulnerability
Program
Step 2. Program vulnerability
- Inject malicious code
- Change jump/branch targets
Malicious Code
Step 3. Execute unintended (or malicious) code
- Injected code
- Unintended control transfer

Figure 1. Security attack scenario.

Figure 1 illustrates attacks which attempt to take control of a vulnerable program. 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 inter-process 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 [12] by overflowing a buffer allocated in the stack.

The format string vulnerability [11] 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.

3 Protection Scheme

This section explains our approach to stop attacks under the security model presented in the previous section. We also provide two examples to illustrate our protection scheme.

3.1 Overview

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 a program produce incorrect results, for instance, by overwriting the program's states 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 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 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*. Note that the operating system can always be conservative and consider all I/O channels as spurious. Thus, identifying potentially malicious channels is not a major problem.

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

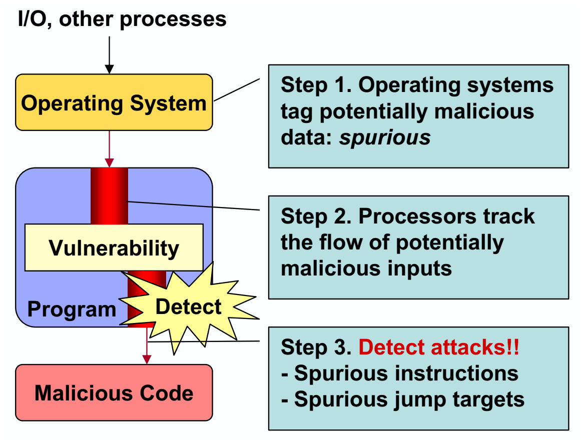


Figure 2. Our protection scheme against security exploits.

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.

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*.

Finally, if the processor detects the use of spurious data as jump target addresses or execution of spurious instructions, it 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.

3.2 Security Policies

Since programs and systems will have different malicious I/O channels and different security requirements, the protection scheme should be flexible enough to handle this variance. For this purpose, *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*.

The security policy first specifies which input channels should be tagged as spurious. 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, one should be careful in placing absolute trust in an untracked channel since no attack from this channel can be detected.

Spurious information can propagate in various ways during a program execution. Section 4 discuss the types of dependencies in detail. The security policy specifies which dependencies should be tracked by the processor. As noted above, one can always be conservative and track all input channels and all possible dependencies. The experiments show that our protection scheme does not cause false alarms even in this case. However, tracking unnecessary input channels and dependencies will incur higher memory space and performance overhead.

Finally, the security policy determines what kind of operations are allowed for spurious data. In this paper, we assume that spurious data is allowed for all operations *except when used as an instruction or jump target addresses*. These restrictions are enough to prevent attackers from gaining control of protected programs. The operating system may be able to provide security against a broader class of attacks if it further restricts the use of spurious information. We do not address this here.

In this paper, we assume the security policy is specified in the operating system and enforced by the processor – the processor throws an exception when it detects a security violation. 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 8).

3.3 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 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

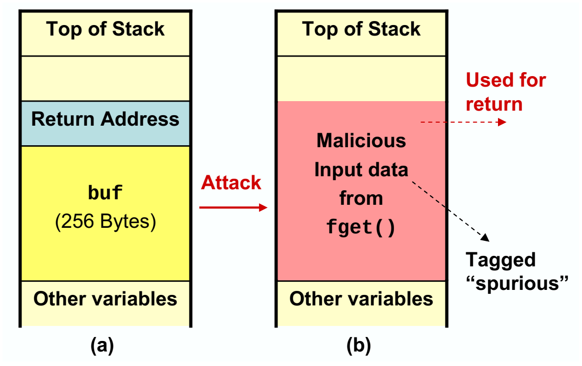


Figure 3. The states of the program stack before and after a stack smashing attack.

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`, 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 4). 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.

3.4 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 [11].

```
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 4). 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.

4 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.

4.1 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.

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

4.2 Tracking Information Flows

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

- *Copy dependency*: If a spurious value is simply copied into a different location, the value of the new location is also spurious.
- *Computation (Comp) 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-address (LDA) dependency*: When a spurious value is used to specify the address to access, the loaded value is considered spurious. Unless the bound of the spurious value is explicitly checked by the program, the result could be any value since it is from an unpredictable address.
- *Store-address (STA) 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.

Processors dynamically track spurious information flows by tagging the result of an operation as spurious if it has

Operation	Example	Meaning	Tag Propagation
Computation (Copy)	`ADD R1, R2, R0`	`<R1>←<R2>`	`T[R1]←T[R2]&Mask[0]`
	`ADD R1, R0, R2`		
	`ADDI R1, R2, #0`		
Computation (Others)	`ADD R1, R2, R3`	`<R1>←<R2>+<R3>`	`T[R1]←(T[R2]|T[R3])&Mask[1]`
	`ADDI R1, R2, #Imm`	`<R1>←<R2>+Imm`	`T[R1]←T[R2]&Mask[1]`
Load	`LW R1, Imm(R2)`	`<R1>←Mem[<R2>+Imm]`	`Temp←T[Mem[<R2>+Imm]];` `T[R1]←(Temp&Mask[0])|(T[R2]&Mask[2])`
Store	`SW Imm(R1), R2`	`Mem[<R1>+Imm]←<R2>`	`Temp←(T[R2]&Mask[0])|(T[R1]&Mask[3]);` `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, and `R0` is a constant zero register. `Mem[]` represents the value stored in the specified address. `T[]` represents the security tag for a register or a memory location specified.

a dependency on spurious data. For flexibility, dependencies to be tracked are specified by the operating system in a bit vector `Mask[0:3]`. For example, copy dependency is tracked only if the first bit in the vector `Mask[0]` is set. Similarly, `Mask[1]`, `Mask[2]`, and `Mask[3]` indicate whether the processor tracks computation, load-address, and store-address dependencies, 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. In some cases, computations are considered as a copy if they are used to produce the result that is identical to an input. For example, `ADDI R1, R2, #0` or `OR R1, R2, R0`, where `R0` is the constant zero register, is considered as a copy. 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.

4.3 Two Security Policies

We define two security policies based on the types of information flows to be tracked, and use them in our simulations. Policy 1 tracks copy, load-address, and store-address dependencies, but not computation dependency. This policy represents a light-weight version that provides security against known types of attacks with low overhead. Policy 2 tracks all four dependencies to provide security against a broader class of attacks incurring more overhead.

Note that we do *not* track any form of control dependency in this work. We believe that control dependency is not essential to detecting attacks of the kind considered in this paper. This is because while control may decide what values are assigned to a variable, the authenticity/spuriousness of each value that is assigned is determined by tracking computation and copy dependency on other authentic/spurious values.

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 Efficient Tag Management

Dynamic information flow tracking requires the modifications to the processing core for tag propagation as described in Table 1. Further, 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.

5.1 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 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 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 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 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.

5.2 On-Chip Structures

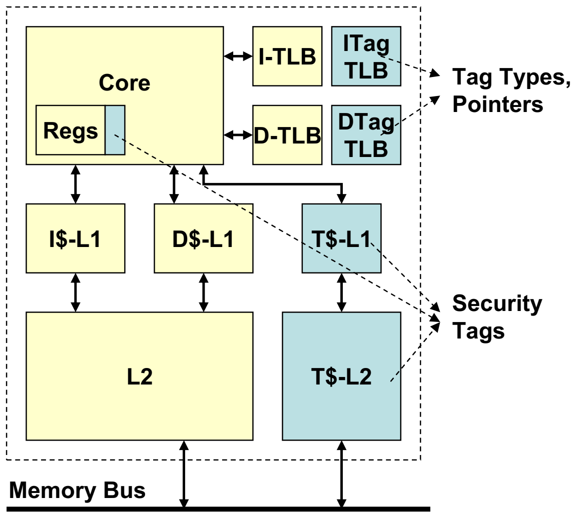


Figure 4. On-chip structures to manage security tags. Dark (blue) boxes represent additional structures.

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 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.

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 base 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 Sections 6 and 7, we discuss the impact of security tags on performance in detail.

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 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 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 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.

In both security policies 1 and 2 (cf. Section 4.3), 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" [12] and Tripbit Secure Code Analizer are tested with simulations.
- **Heap buffer overflows**: Attacks can also exploit buffer overflows in the heap area. In most cases, the attack involves injecting malicious code in the heap. 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**: This is a special type of a heap buffer overflow attack suggested in [7]. Attacks overwrite a field of a double-linked list in `malloc`. On a subsequent call to `free`, the list update will overwrite an arbitrary location. We consider a case where this overwrite modifies function pointers or return addresses.
- **Format string attacks**: A basic format string attack is described in Newsham's document [11]. Here, we study example cases of *QPOP 2.53*, *bftpd*, and *wu-ftpd 2.6.0* in a document from the TESO security group [15]. *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 our example.

Table 4 summarizes the effectiveness of our protection scheme against various attacks. All attacks are detected and stopped with either Policy 1 or Policy 2. In examples of buffer overflow attacks, malicious inputs are simply copied and used as instructions or jump targets. Similarly, in *wu-ftpd*, the I/O input is simply copied into a buffer and a part of it is used as a malicious store address. Therefore, stack/heap buffer overflows, *QPOP*, and *bftpd* are detected with just copy dependency, and attacks on *wu-ftpd* are detected with copy and store-address dependencies. *vudo* requires three dependencies to detect since it reads a spurious pointer to a node of a double-linked list (copy), reads the `prev` field using a proper offset with the pointer to the node (load-address), and updates the `prev->next` field (store-address). In general, the experiments indicate that our protection scheme is very effective against a large range of security attacks.

Attacks	Policy 1	Policy 2	Dependencies
Stack BO	Yes	Yes	Copy
Heap BO	Yes	Yes	Copy
vudo	Yes	Yes	Copy+LDA+STA
`QPOP`, `bftpd`	Yes	Yes	Copy
`wu-ftpd`	Yes	Yes	Copy+STA

Table 4. Effectiveness of our protection scheme against various security exploits.

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 17 SPEC CPU2000 benchmarks [6] with our security policies. Each benchmark is simulated for 100 billion instructions or to completion to obtain the results.

The simulations showed that our scheme does *not* cause any false alarms for the SPEC benchmarks. Policy 1 did not have any use of spurious values as instructions or jump target addresses for any of the benchmarks. In Policy 2, there are legitimate uses of spurious information after bound checking, however, these do not cause false alarms because they are successfully detected and allowed by our mechanisms described in Section 7.

6.2 Memory Space Overhead for Policy 1

Dynamic information tracking only 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 5 in terms of actual storage overhead for security tags compared to regular data. Table 5 summarizes the space overhead of security tags for Policy 1.

For security policies without 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 results in over 95.7% pages with per-page tags and only 0.26% space overhead on average.

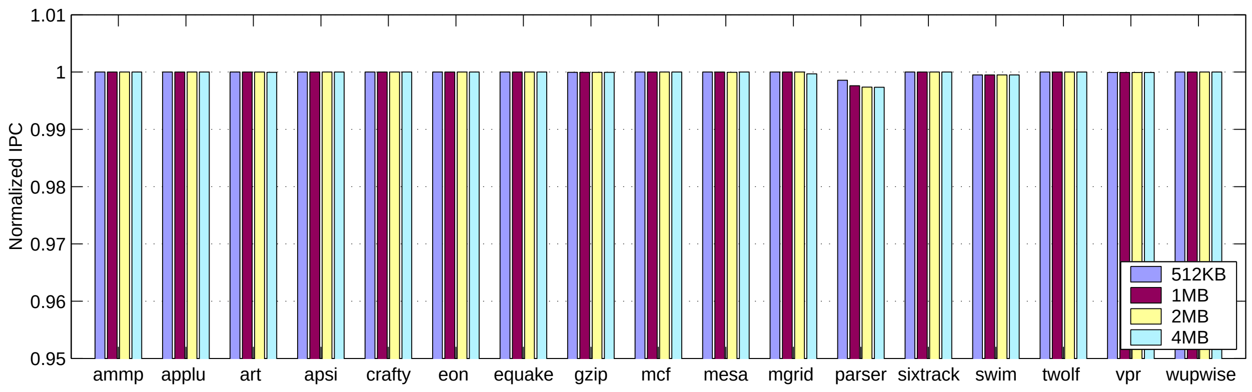


Figure 5. Performance overhead of Policy 1 for various L2 cache sizes (1/8 tag caches).

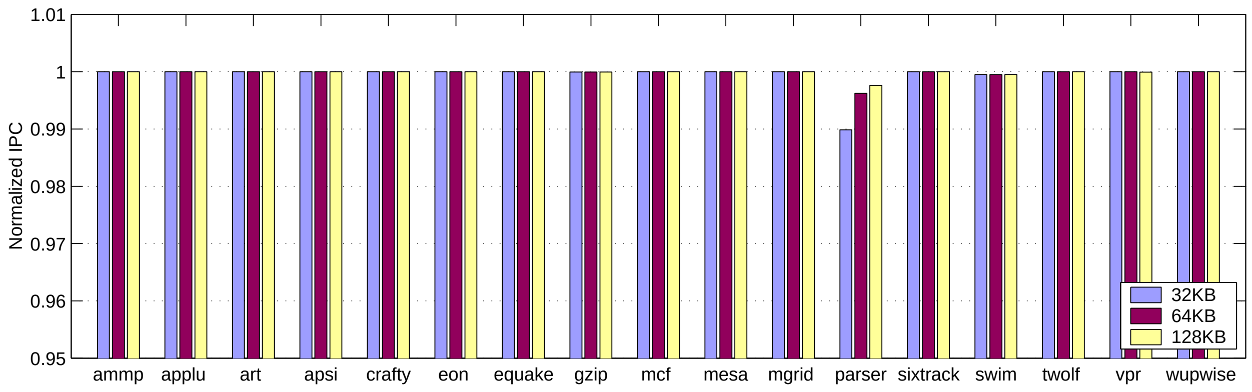


Figure 6. Performance degradation of Policy 1 with small L2 tag caches (1-MB unified L2 cache).

6.3 Performance Overhead for Policy 1

Finally, we evaluate the performance overhead of our 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.

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 the tag overhead 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%. Given diminishing performance returns for larger caches, the error from this approximation is unlikely to be significant.

Our protection scheme can affect the performance in two ways. First, accessing security tags consumes additional off-chip bandwidth. 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 5 shows the performance overhead of our scheme for various L2 cache sizes. 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. With Policy 1, the performance degradation is negligible. There is only 0.3% degradation in the worst case. The benchmark `parser` is an extreme case with the least amount of per-page tags and therefore suffers the highest performance degradation.

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. How-

Benchmark	Policy 1 (%)			
	Per-Page	Per-QWord	Per-Byte	Overhead
ammp	99.85	0.00	0.15	0.02
applu	99.99	0.00	0.01	0.00
apsi	99.97	0.00	0.03	0.00
art	82.46	0.00	17.54	2.19
crafty	99.02	0.00	0.98	0.12
eon	97.97	0.00	2.03	0.25
equake	99.71	0.00	0.29	0.04
gzip	82.25	14.10	3.65	0.68
mcf	99.99	0.00	0.01	0.00
mesa	99.62	0.00	0.38	0.05
mgrid	99.94	0.00	0.06	0.01
parser	69.53	26.75	3.72	0.88
sixtrack	99.69	0.00	0.31	0.04
swim	99.98	0.00	0.02	0.00
twolf	98.60	0.00	1.40	0.18
vpr	99.74	0.00	0.26	0.03
wupwise	99.98	0.00	0.02	0.00
avg	95.78	2.40	1.82	0.26

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.

ever, given that we have only 0.21% overhead for security tags, small tag caches do not hurt the performance for Policy 1 as shown in Figure 6.

7 Tracking Computation Dependency

In Policy 2, we track computation dependency in addition to the other dependencies. For the attacks that we tested, Policy 1 was sufficient because these attacks do not use any computation dependency. In fact, we could not find any real world attacks that require computation dependency for detection. However, we believe that some vulnerable programs will need the stronger policy with computation dependency for protection against buffer overflow and format string attacks.

For example, consider the following buffer overflow vulnerability.

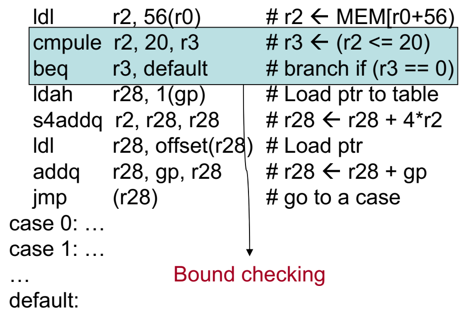
```
int filter(void)
{
    char buf[256];
    int i = 0;

    fgets(buf);

    while (buf[i]) {
        buf[0] = 0.5*buf[i] + 0.5*buf[i+1];
        i++;
    }
    ...
}
```

In this example, the function gets an input string and filters the string. Unfortunately, the bound of `buf` is not checked in `fgets` and attackers can overwrite the stack near the buffer. What is different in this example is that the program performs computation on the buffer, *again, without checking the bound of the buffer*. As a result, the overwritten return address will not be detected without computation dependency.

Tag propagation basics for computation dependency were described in Table 1. In addition, some instructions need to be treated specially. We also require additional software support to detect legitimate uses of spurious data.

7.1 Legitimate Use of Spurious Data

In general, programs should not use spurious data as jump targets and load/store addresses since they cannot predict the resultant behavior. However, *after explicit bound checking*, programs may use spurious data for these purposes to implement efficient control structures. For example, the `switch` statement in C, jump tables, and dynamic function pointer tables often compute pointers to `case` statements or table entries directly from spurious inputs. In these cases, spurious data effectively becomes authentic because its bound is explicitly checked.



```
    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: …
…
default:
```

Figure 7. A switch statement with a jump table.

As an example, Figure 7 shows a code segment from a SPEC benchmark, which implements a switch statement using a jump table. 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 for jump. Since this is a legitimate use of spurious information, tracking the spurious flows for computation and load-address dependency will result in a false alarm.

One possible solution for this problem is to have compilers mark these legitimate uses as *safe*. However, this approach will require re-compilation of each program. Instead, the tag propagation scheme is slightly modified to identify *safe computations* on spurious data, and a simple software algorithm is used to statically inspect the binary executable and mark *safe loads* (binary annotation).

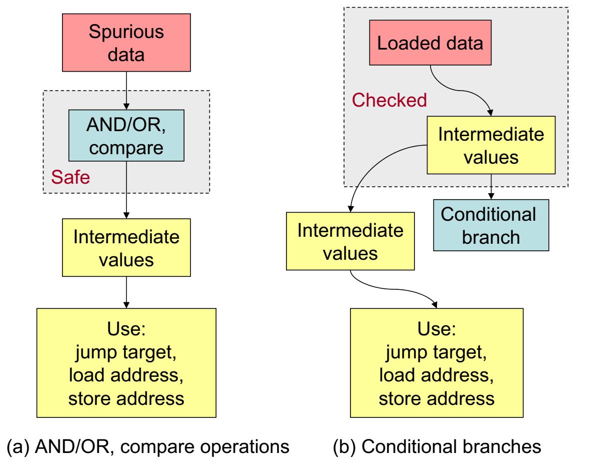


Figure 8. Legitimate uses of spurious data.

There are two possible ways for programs to ensure that spurious data is safe to use. First, before the use, programs can perform special operations on the data that only produce restricted results. Figure 8 (a) shows this case. For example, `AND R1, R2, #7` can only produce the result between 0 and 7, and *compare operations* only have two possible results: true and false. Thus, these operations effectively produce a bound-checked result.

We deem two types of operations as bound checking: *AND/OR* and *compare*. For these operations, the tag propagation is slightly modified. The processor marks the results as *spurious* only if both inputs are spurious; the output is *authentic* if an input is authentic or an immediate.

The second way to ensure the safety of spurious data is to check the bound using *conditional branches* as shown in the switch example. Figure 8 (b) illustrates the case. To detect these bound checks, we use a simple software algorithm to inspect and annotate the binary before execution. Load instructions are modified to have one hint bit (*safety bit*) indicating whether the bound of the value loaded will be checked with conditional branches or not. At run-time, if the processor reads a spurious value using a load instruction with the safety bit set, the value is tagged *authentic*.

The annotation can be done once when the binary is installed to a system and saved in the hard disk. Or the operating system can annotate a program in memory as a part of a load process. In the latter case, the scheme is transparent to executables.

Figure 9 summarizes the software algorithm to annotate the binary. First, the algorithm walks through the program ignoring any jump or branch. If a conditional branch is found, the algorithm backtracks the dependency chain for

```
For every conditional branch (Bxx Ri, target) in the program,
  1. $S = \{$Ri$\}$;
  2. $addr$ = the address of the branch - 4;
  3. while (($S \neq \emptyset$) && ($S \neq \{$R0$\}$) && ($addr \geq$text_base)) {
          $inst$ = the instruction at address $addr$;
          if ($inst$ == jump or branch) break;
          if (($inst$ == Op Rd, Rs1, Rs2) &&
             (Rd $\in S$) && (Rd $\neq$ R0)) {
               $S$ = $S$ - {Rd} + {Rs1, Rs2};
          }
          if (($inst$ == Op Rd, Rs, #Imm) &&
             (Rd $\in S$) && (Rd $\neq$ R0)) {
               $S$ = $S$ - {Rd} + {Rs};
          }
          if (($inst$ == load Rd, Imm(Rs)) && (Rd $\in S$) {
               Set the safety bit of the load;
               $S$ = $S$ - {Rd};
          }
          $addr = addr - 4$.
     }
```

Figure 9. Safe load annotation algorithm. Some details omitted for clarity.

the branch condition and marks all the loads that contribute to it as *safe*.

The backtracking is quite simple. The annotation algorithm uses a set S to track all the registers that contribute to the branch condition, and attempts to find all the load instructions that can affect the value in the branch register. Moving backward from the instruction right before the conditional branch, the algorithm adds the source registers of an instruction to S and removes the destination reigster from S when the instruction is a part of the dependency chain to the conditional branch. When the algorithm encounters a load that reads a value into a register in S, the load is marked as *safe*.

Finally, if the algorithm encounters a jump or a branch, it aborts backtracking for the current conditional branch and finds the next conditional branch. This is a conservative approach to ensure that we annotate loads only if we can guarantee that the conditional branch is always reached after the loads. In our SPEC benchmarks, this conservative scheme is enough to detect all legitimate uses of spurious values since the values are checked right after they are loaded from memory.

7.2 Memory Space Overhead for Policy 2

The additional hardware cost of Policy 2 relative to Policy 1 is the extra safety bit required for the load instructions.

We now evaluate Policy 2 for memory space overhead. The results are summarized in Table 6. 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 the 27% per-page tags average. 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 2, the space overhead is 4.5% for the chosen SPEC benchmarks.

Benchmark	Policy 2 (%)			
	Per-Page	Per-QWord	Per-Byte	Overhead
ammp	1.58	4.58	93.84	11.80
applu	0.94	99.02	0.03	1.55
apsi	60.38	39.55	0.07	0.63
art	6.45	75.60	17.94	3.42
crafty	97.70	0.00	2.30	0.29
eon	79.73	6.76	13.51	1.79
equake	3.44	31.93	64.64	8.58
gzip	52.53	43.72	3.75	1.15
mcf	0.09	99.88	0.03	1.56
mesa	46.26	0.08	53.66	6.71
mgrid	1.61	98.27	0.12	1.55
parser	1.55	0.11	98.34	12.29
sixtrack	79.13	19.66	1.21	0.46
swim	0.49	99.47	0.04	1.56
twolf	22.11	5.61	72.28	9.12
vpr	1.04	1.47	97.49	12.21
wupwise	0.53	99.43	0.04	1.56
avg	26.80	42.66	30.55	4.48

Table 6. 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.

7.3 Performance Overhead for Policy 2

We evaluate the performance overhead of Policy 2 compared to the baseline case without any protection mechanism. As before, experimental results (IPCs) in figures are normalized to the IPC for the baseline case without any security mechanism.

Figure 5 shows the performance overhead of Policy 2 for various L2 cache sizes. As before, the tag caches are always one-eighth of the corresponding caches for instruction and data. For Policy 2, the protection scheme only incurs 0.8% performance degradation on average, and 6% in the worst case of `twolf`.

Figure 11 shows the performance under different tag cache sizes. Even for Policy 2, smaller tag caches work in most cases without imposing a significant performance cost. In the figure, 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 has a large number of per-byte tags as in `ammp`, `parser` and `twolf`, the size of a tag cache can affect the performance significantly.

This degradation is mainly due to the fact that we delay the use of data until the corresponding tag is ready. If we assume a mechanism to decouple data and tag computations, even `twolf` with a 32 KB tag cache has only 5% performance degradation. Other benchmarks such as `ammp`, `parser`, and `vpr` show less than 1% degradation if the tag access latency is ignored.

8 Related Work

Recent works have proposed hardware mechanisms to prevent stack smashing attacks [18, 9]. In these approaches, a processor stores a return address in a separate memory location and check the value in the stack on a return. Unfortunately, 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.

Our tagging mechanism is similar to the ones used for hardware information flow control [14]. 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 [14]. 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.

Secure processors such as XOM [10] and AEGIS [16] target secure execution environments for programs. However, they assume that protected programs are well-written, and are still vulnerable to attacks exploiting software bugs such as buffer overflows and format strings. Because a process within a secure compartment overwrites a memory location, these attacks are not prevented.

There have been a number of software approaches to provide automatic detection and protection against buffer overflow and format string attacks. We briefly summarize some successful ones below.

StackGuard [3], StackShield [17] are both compiler patches that are targeted to prevent stack smashing attacks. Both techniques only work for specific type of buffer overflow attacks that modify a return address in a stack, and require recompilation. Kernel patches such as StackGhost [5] and the patches described in [4] and in PaX [13] have been

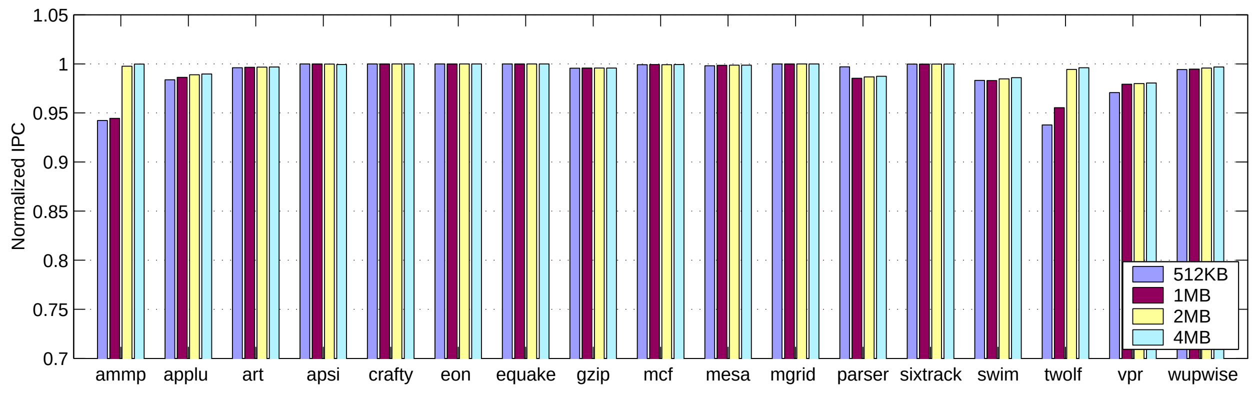


Figure 10. Performance overhead of Policy 2 for various L2 cache sizes (1/8 tag caches).

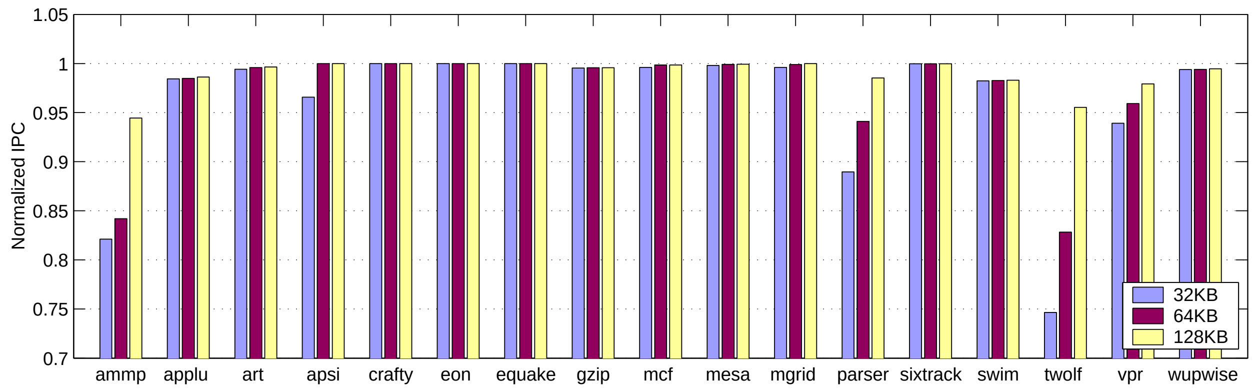


Figure 11. Performance degradation for Policy 2 with small L2 tag caches (1-MB unified L2 cache).

successful against preventing some types of attacks. However, these techniques cannot be used for applications with legitimate use of dynamically generated code such as just-in-time compilation. FormatGuard [2] is a library patch for eliminating format string vulnerabilities. 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 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 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 allows 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.

9 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.

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.

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] 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.

[10] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9^{th} Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

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

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

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

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

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

[16] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17^{th} Int'l Conference on Supercomputing*, June 2003.

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

[18] 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.