Integration of Static Instruction Analysis with Dynamic Information Flow Tracking

Ivan Beschastnikh, Ian Post, Joshua Schwartz, Benedict Singer

{ivan,ianp,vegan,singerb} @uchicago.edu

March 13, 2006

Abstract

Computer security is a constant issue for computer systems; thus, many schemes have been proposed to stop specific attacks. In this paper, we present a software implementation of the Dynamic Information Flow Tracking (DIFT) system which is designed to stop a broad range of control flow attacks [9]. We modified the SimpleScalar/PISA simulator (v.3.0) to include an extended instruction set implementing tag operations. Due to the software nature of our scheme we can apply static analysis of assembly code.

1 Introduction

With the growth of the internet, computer security vulnerabilities have become an increasing problem, and researchers have devoted much effort to the problem of designing secure software and preventing exploitation in insecure software [10, 6]. In particular, Suh et al. [9, 10] have proposed a hardware scheme called dynamic information flow tracking, or DIFT, that prevents many common forms of attack with minimal false positives or negatives and small overhead.

Suh et al. note that, regardless of the attack's specific form, the majority of exploits use bugs to somehow overwrite program control data with user input, which can then be used to hijack the program as soon as it uses the malicious control data. Almost no programs are designed to work in this fashion: they rarely intend to execute input or use an address derived from input as a jump target. Note that all input is managed via the operating system. The basic idea of DIFT is to exploit these observations to identify security attacks. DIFT introduces tags which mark registers or bytes in memory as spurious. The operating system tags all bytes from input as spurious, and spurious tags propagate across operations if either of the operands is spurious. The program is killed if it ever tries to execute code at a spurious address or use a spurious value as an instruction.

DIFT produces impressive results. On a SimpleScalar simulation, it successfully blocked a variety of attacks tested against it with no false positives or negatives. Suh et al. lay out several security policies, which define how tags are propagated across operations. For example, in the case of a load instruction, a basic security policy would mark the destination register as spurious if its value came from spurious memory, while stricter policy would also tag the destination register as spurious if its new value was not spurious but the source *address* was spurious because the spurious address may have been used to load an unexpected value. DIFT added only about 1.5% overhead to both performance and memory in the most strict tagging policy [9]. In an updated version of the paper, they investigated different security policies and improved checking of pointer arithmetic by detecting bounds checking of the pointer, allowing safe pointer arithmetic with spurious data to pass and preventing unbounded spurious pointer arithmetic. The simplest security policy tested required only .25% memory overhead and negligable performance cost, while the most comprehensive policy averaged 4.5% memory and 1% performance cost [10].

One issue with DIFT is that it requires specialized, dedicated hardware that does nothing except update and test tags. Although this hardware improves performance by allowing tag propagation to be executed in parallel with the regular instructions, many processors are moving away from specialized hardware and toward microcode or software implementations. Accordingly, we have implemented the original DIFT scheme using explicit tag-updating and checking instructions added to the ISA. Although the additional instructions result in a large overhead in performance and code size, they serve as a crude substitute for a microcode implementation, which we were unable to simulate. This ISA implementation opens up the possibility of more flexible or different forms of tag checking and allows us to investigate how many of the tag updates are actually necessary and how many can be optimized away. We propose to examine both of these issues using static analysis of assembly code.

2 Motivation

Existing software contains a wide array of bugs that allow a malicious user to trick the program into transferring control to a piece of arbitrary code [7, 8, 1, 3]. Although the form and method of exploitation on the part of these bugs vary greatly, each centers around a single critical step: a piece of control data—that is, data used in determining a jump address such as function return addresses, function pointers, `longjmp` buffers, and various system hooks—is overwritten based on user input. Subsequently, when the program makes the appropriate jump, it transfers control to an exploit pointed to by the corrupt control. We discuss several types of control flow attack below.

2.1 Stack Smashing

The best-known control flow attack is a form of buffer overflow known as stack smashing[7], which takes advantage of inadequate bounds-checking in local arrays to overwrite a function's return address. On many systems, to allow resumption of execution when a function returns, each function call pushes the current program counter onto the stack before transferring control to the new function, which then expands the stack to make room for local variables. When the function returns, it pops the old program counter off the stack and jumps to that address. As a result, if the return address has been overwritten during execution, the program will jump to an arbitrary memory location.

For instance, in the following code `scanf()` has no way of knowing `buf`'s length. As a result, if it reads more than 256 bytes of input, it will write past the end of `buf` and corrupt adjacent memory.

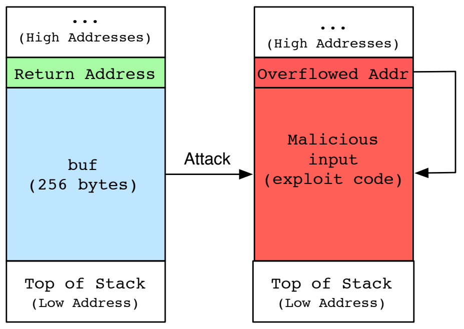

Figure 1: Program stack before and after buffer overflow

```
void foo()
{
    char buf[256];
    ...
    scanf("%s", buf);
    ...
    return;
}
```

As shown in Figure 1, the return address is stored after the buffer and is vulnerable to being overwritten. The user can then hijack the program by injecting exploit code elsewhere in the program, such as `buf` and overwriting the return address with the location of this code. Buffer overflows are described in more detail by an anonymous hacker[7].

2.2 Format String Attacks

Format string vulnerabilities arise when a program directly uses input as a format string in functions such as `printf()` or `sprintf()`, allowing a malicious user to overwrite arbitrary memory locations through careful use of formating commands. For example, in the following code, `buf` is meant to be interpreted as a text string, as in `printf("%s", buf)`.

```
void foo(char *str)
{
    char buf[256];
    strncpy(buf, str, 256);
    ...
    printf(buf);
    ...
}
```

A problem arises because `printf(buf)` will interpret any %'s in `buf` as formatting instructions.

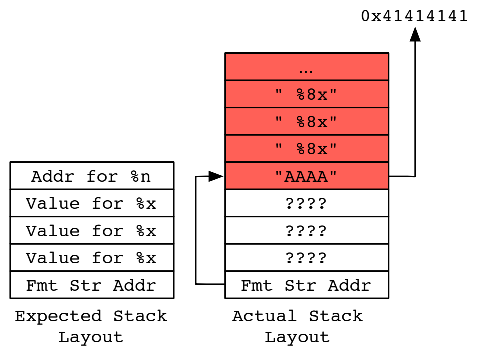

Figure 2: Placement of `printf()`'s arguments on the stack

For example, if `str` is `"AAAA %8x %8x %8x %n"`, `printf()` will assume the next address after its first argument will contain the value for the first `%x`. After reading through the return address and old frame pointer on the stack, it will come to `buf`, which lies at the top of the previous stack frame, and it will read `AAAA` as the argument for `%n`, which instructs it to record the number of characters printed so far (See Figure 2). As a result, `printf()` will store 32 in `0x41414141` (the address corresponding to `AAAA`, assuming a big endian machine). Through careful crafting of such a string, an attacker can overwrite an arbitrary location, such as a return address or function pointer, with an arbitrary value. Scut [8] discusses format string attacks at length.

2.3 Heap Corruption Attacks

Heap overflows come in a wide variety of forms and depend heavily on a number of factors including program memory use patterns and the implementation of `malloc()` used. One of the simpler forms overwrites heap metadata in the GNU C library implementation of `malloc` to write to arbitrary locations. Consider the following code:

```
void foo()
{
    char *buf;
    buf = (char *) malloc(256);
    ...
    scanf("%s", buf);
    ...
    free(buf);
    ...
}
```

As in the stack overflow example, `scanf` will overflow `buf` if given too much input, but in this case there is no vulnerable control data following `buf`. However, there is most likely another chunk of memory owned by `malloc` immediately following `buf`, and in the `glibc` implementation it will be prefaced by a header containing, among other things, an in-use flag and 2 pointers `bk` and `fd` pointing to the previous and next `malloc()` chunks respectively. When `buf` is freed, it will check whether this next chunk is in use, and if so, it will merge the two chunks, which involves the assignment `fd->bk = bk`. The user can therefore overwrite the next chunk's header so that the in-use flag is clear, `fd` is such that `fd->bk` is a piece of control data, and `bk` is its desired value, causing the program to be hijacked when `buf` is released. An anonymous author [1] lays out the details of several `malloc()` attacks.

3 Previous Work

A number of schemes have been proposed to thwart various control data attacks. Some focus on preventing the final transfer of control to exploit code. Modern compilers, for example, may mark stack and heap pages as non-executable to thwart the injection of exploit code. However, this defense may be bypassed using the technique of returning into libc. In this technique, control is transferred to a libc function such as `system()` rather than to code on the stack [3]. Several operating system kernels support security features such as address space randomization, in which the stack and shared libraries are stored in randomized locations, making it difficult to successfully guess the location of code needed for exploitation.

Other schemes seek to provide mechanisms to protect control data or prevent its corruption. StackGuard, for instance, places random canary values on the stack just after return addresses. If the value changes during function execution, the return address may have been corrupted. StackGuard also supports a similar technique of XORing of return addresses with random values. StackShield goes a different route and stores return addresses off of the stack entirely. However, such schemes, at best, protect against a narrow class of attacks and can often be bypassed [2]. Similarly, PointGuard encrypts pointers in memory, which protects some types of control data, but does nothing to protect other forms such as return addresses [5].

The limitations of software schemes have led to several proposals for implementing protection in hardware. These methods have the advantage of stopping wide classes of attacks and not requiring recompilation of existing software. Our work is based on the DIFT framework developed by Suh et al.[9], which we discussed in Section 1. Crandall and Chong developed a very similar scheme called Minos [6], that also tracks the spread of input to protect control data. Minos has similar performance to Suh et al.'s version, with complete success stopping attacks and minimal overhead, but Crandall and Chong emphasize that Minos is implemented in physical memory and is independent of the memory model used by the architecture or kernel.

Finally, Chen et al. [4] argue that overwriting control data is not necessary for all attacks, and they propose a more comprehensive security policy based on the notion of pointer taintedness: a pointer is tainted if its value can depend on user input, and any dereference of a tainted pointer is assumed to be an attack. Their tests stop all attacks with no false positives or negatives, and they claim low overhead.

4 ISA Design

4.1 Requirements

Our additions to the ISA support several tag operations required by the scheme proposed in Suh et. al [9, 10]; additionally, the existing instructions in PISA influenced our design (for example, we needed to be able to set the high and low result registers used in multiply and divide computations). In particular, we needed to be able to:

- Set the tag of a register based on:
  - The tag of another register or pair of registers
  - The tag of a memory location
  - An immediate value (1 or 0)
- Set the tag of a memory location based on:
  - The tag of a register
  - An immediate value (1 or 0)
- Test the tag of a register

These requirements correspond to types of instructions in PISA, i.e. computations, loads, stores, jumps, etc. A full list of these instructions is in Table 2, with accompanying legend in Table 1.

4.2 Additional Considerations

Additionally, we included instructions to perform multiple tag setting operations in one instruction, allowing us to set multiple registers at once, or set the tag of a register based on more than two other registers. These instructions provide the potential for greater tag-setting efficiency.

Table 1:
New Instructions for PISA: Legend

Abbreviation	Meaning
`Rn`	Register #n
`HI`	High result register
`LO`	Low result register
`B`	Byte length
`T[ref]`	Tag of object ref
`T[]`	All tags
`Mem[addr]`	Memory location at address addr
`O`	Memory offset
`I`	Immediate 1 or 0
`M`	16-bit mask, in hexadecimal

5 Naïve Assembly Annotation

Because the instructions in our new ISA are not known by an existing compiler, we needed to design a scheme to insert these instructions into existing assembly code. We term this process *assembly annnotation.* The simplest such scheme, which we term *naïve annotation*, is to insert the appropriate tag-propogating or tag-checking instruction directly before every instruction. For instance, consider the following assembly code:

```
add $1, $2, $2
sw $1, 0($sp)
lw $1, 4($sp)
```

Since each instruction requires a tag-bit to be set for its destination, a naïve annotation yields:

```
sett $1, $2, $2
add $1, $2, $2
setmt $1, 0($sp)
sw $1, 0($sp)
sett $1, 4($sp)
```

Table 2:
New Instructions for PISA: Full List

Instruction	Semantics	Opcode

`setmt I, (R1+R2), B`	`T[Mem[R1 + R2]] = I`
	`...`
	`T[Mem[R1 + R2 + B - 1]] = I`
		0xb0

`setmt I, O(R1), B`	`T[Mem[R1 + O]] = I`
	`...`
	`T[Mem[R1 + O + B - 1]] = I`
		0xe0

`sett R1, I`	`T[R1] = I`	0xb1

`setmt R1, (R2+R3), B`	`T[Mem[R2 + R3]] = R1`
	`...`
	`T[Mem[R2 + R3 + B - 1]] = R1`
		0xb2

`setmt R1, O(R2), B`	`T[Mem[R2 + O]] = R1`
	`...`
	`T[Mem[R2 + O + B - 1]] = R1`
		0xe2

`sett R1, (R2+R3), B`	`T[R1] = T[Mem[R2 + R3]] | ... | T[Mem[R2 + R3 + B - 1]]`	0xb3
`sett R1, O(R2), B`	`T[R1] = T[Mem[R2 + O]] | ... | T[Mem[R2 + O + B - 1]]`	0xe3
`sett R1, R2, R3`	`T[R1] = T[R2] | T[R3]`	0xb4
`sett16 M, I`	`for R1 in M: T[R1] = I`	0xb5
`sett32 M, I`	`for R1 in M: T[R1 + 16] = I`	0xb6
`sett16 M, R1`	`for R2 in M: T[R2] = T[R1]`	0xb7
`sett32 M, R1`	`for R2 in M: T[R2 + 16] = T[R1]`	0xb8
`ldtag R1`	`T[] = R1`	0xb9
`sttag R1`	`R1 = T[]`	0xba
`testt R1`	`raise exception if: (T[R1] is spurious)`	0xbb
`testnt R1`	`raise exception if: (T[R1] is not spurious)`	0xbc
`testtr R1`	`raise exception if: (T[R1] is spurious | if T[Mem[R1]] is spurious)`	0xbd
`setth R1`	`T[HI] = T[R1]`	0xe4
`settl R1`	`T[LO] = T[R1]`	0xe5
`setthr R1`	`T[R1] = T[HI]`	0xe6
`setttl R1`	`T[R1] = T[LO]`	0xe7

`setthl R1, R2`	`T[HI] = T[R1] | T[R2]`
	`T[LO] = T[R1] | T[R2]`
		0xe8

`settor16 R1, M`	`for R2 in M: T[R1] |= T[R2]`	0xe9
`settor32 R1, M`	`for R2 in M: T[R1] |= T[R2 + 16]`	0xea

```
        lw $1, 4($sp)
```

Such an annotation scheme will result in a substantial increase in instruction count (IC), as the majority of instructions require some sort of tag-setting operation. As described in Section 12, our initial experiments show roughly a 59% increase in IC. However, this scheme may insert extraneous instructions. In the above example, because the tag of `$1` is set twice, the first `sett` could be removed, reducing the code to:

```
        add $1, $2, $2
        setmt $2, 0($sp)
        sw $1, 0($sp)
        sett $1, 4($sp)
        lw $1, 4($sp)
```

In such situations, we wish to identify and remove such redundant operations. However, to do so, we must first parse the assembly code into manageable units. In particular, we note that straight-line code provides the simplest environment for optimization, since branches present control flow complexities and, within the DIFT framework, require that the spurious status of certain registers and addresses be known in order to determine branch-validity. Thus, the first stage in attempting to optimize annotation is the breaking of code into straight-line blocks.

6 Basic Blocks

A basic block is a straight-line sequence of instructions, such that control always arrives at the start of the block, and always leaves from the end, i.e. a block where there are no jumps out until the end and any jump in arrives at the beginning of the block. Following is a simple example of a basic block:

```
        label1:
                add $1, $2, $3
                sub $1, $4, $1
                sw $1, 0($sp)
                bnez $1, label2
```

Because this example uses labels, it is clear that no jumps have targets anywhere other than at the start of the block. This observation is a key part of the strategy for decomposing an assembly file into basic blocks.

6.1 Identifying Basic Blocks

The algorithm for identifying basic blocks proceeds as shown in Figure 6.1.

A complete reconstruction of the file can be achieved easily; each basic block is output in turn, along with its name if its name is a label. Notice that a series of labels with no code will produce a series of empty basic blocks, but these are not an issue; empty basic blocks with non-label names do not contribute to the output, and an empty basic block with a label name just produces that label, which is the correct behavior.

7 Optimized Annotation

Given a basic block, we wish to identify an optimal tagging scheme, where optimal means using as few instructions as possible. Observe that because a basic block terminates at any jump or branch instruction, the status of the spurious bits in the system at the end of the block is a function of both the initial state of the spurious bits and the logic propagating spuriousness across operations. So, for instance, consider the example code below:

```
        add $1, $2, $2
        sw $1, 0($sp)
        lw $1, 4($sp)
```

In this code, the tag of `$1` is dependent on the initial tag of `4($sp)` and the tag of `0($sp)` is dependent on the initial tag of `$2`; all other tags remain constant throughout the block. Thus, we examine the possibility of statically analyzing the logic within a basic block to reduce tag-setting instruction overhead.[1]

7.1 Basic Infection Graphs

Let $G = (V, E)$ be a graph defined as follows. Given a block of n instructions, for each bit b_i in the system (including registers and all memory addresses) define vertices $v_{i,0}, \ldots, v_{i,n}$. Let there be a directed edge between $v_{i,k-1}$ and $v_{j,k}$ if, based on instruction k, the value of bit b_j depends upon the value of bit b_i. In addition, let there be an edge between $v_{i,k-1}$ and $v_{i,k}$ if instruction k does not change the value of b_i. So, for instance, given a block beginning with

[1]It is important to observe that there are unfortunate limits to static analysis. For instance, if all bits in a system are non-spurious at the beginning of a block, then, by necessity, they are all non-spurious at the end of a block – no new spurious bits can be injected, except by I/O operations which lie in their own basic blocks. Thus, in such as situation we would like to skip all tag-setting logic, since it cannot possibly result in a bit becoming spurious. However, without having knowledge of the state of spurious bits – knowledge not available until run-time – this logic cannot be easily avoided.

```
procedure PARSE(file)                              ▷ Decompose an assembly file into basic blocks
    $bblist \leftarrow \emptyset$
    $currblock \leftarrow \emptyset$
    for all lines L in file do
        if L is a label then
            $bblist \leftarrow bblist + (currname, currblock)$
            $currname \leftarrow L$
            $currblock \leftarrow \emptyset$
        else if L is a jump then
            $currblock \leftarrow currblock + L$
            $bblist \leftarrow bblist + (currname, currblock)$
            $currname \leftarrow \emptyset$
            $currblock \leftarrow \emptyset$
        else if currblock = $\emptyset$ then
            $currname \leftarrow \emptyset$
            $currblock \leftarrow currblock + L$
        else
            $currblock \leftarrow currblock + L$
        end if
    end for
    return $bblist$
end procedure
```

Figure 3: Algorithm for decomposing an assembly file into basic blocks, where bblist holds the list of blocks, currname holds the name of the current block, and currblock holds the contents of the current block.

the instruction `add $1,$2,$3`, edges would be placed from $v_{\$2,0}$ to $v_{\$1,1}$ and from $v_{\$3,0}$ to $v_{\$1,1}$, since $b_{\$1}$ will depend on $b_{\$2}$ and $b_{\$3}$. Then, a directed path from one vertex to another represents the propagation of a spurious tag across instructions. If we call a vertex *infected* when the spurious bit it represents is set, then all vertices that lie along a path out from an infected vertex will become infected. Note that, as defined, a traversal of this graph, setting the tag-bit of each vertex when required, is equivalent to the naïve annotator described above. An example of such a graph is contained in Figure 4.

7.2 Bipartite Graph Model

To compress the information in a basic infection graph, observe that the bit values b_i we wish to know at the end of a block of n instructions are represented by the vertices $v_{i,n}$. Furthermore, at the beginning of a block, the values are $v_{i,0}$ are known. As pointed out above, any infection in a later node must be the (possibly indirect) result of a node that is infected at the beginning of the block. As such, we can reduce this infection graph to a bipartite graph containing only vertices representing the start and the end of the block, such that there is an edge between two vertices $v_{i,0}$ and $v_{j,n}$ if there exists a path $v_{i,0}, \ldots, v_{j,n}$ between these two vertices. As shown in Figure 4, such a graph represents the dependencies between tag-bits. By setting the tag-bit for a register to 0 and then using the `settor16` and `settor32` instructions defined in Table 2, all register-register dependencies can be resolved in at most three instructions for a given register, by OR-ing with all infecting tags. For instance, consider the following code:

```
add $1, $2, $3
add $1, $1, $4
add $1, $1, $5
add $1, $1, $26
add $1, $1, $27
```

The tag of `$1` is dependent on `$2`, `$3`, `$4`, `$5`, `$26`, and `$27`. Thus, the following tag setting operations would be used to set the tag for `$1`:

```
sett $1, 0
settor16 $1, 0000000000111100
settor32 $1, 0000110000000000
```
[2]

Note that this scheme has two major limitations:

[2]Note that, in actual assembly, these masks would be specified in hexadecimal. We have used binary to make explicit the tag-bits which are being used.

```
add $1,$2,$2
sw $1,0($sp),4
lw $1,4($sp),4
```

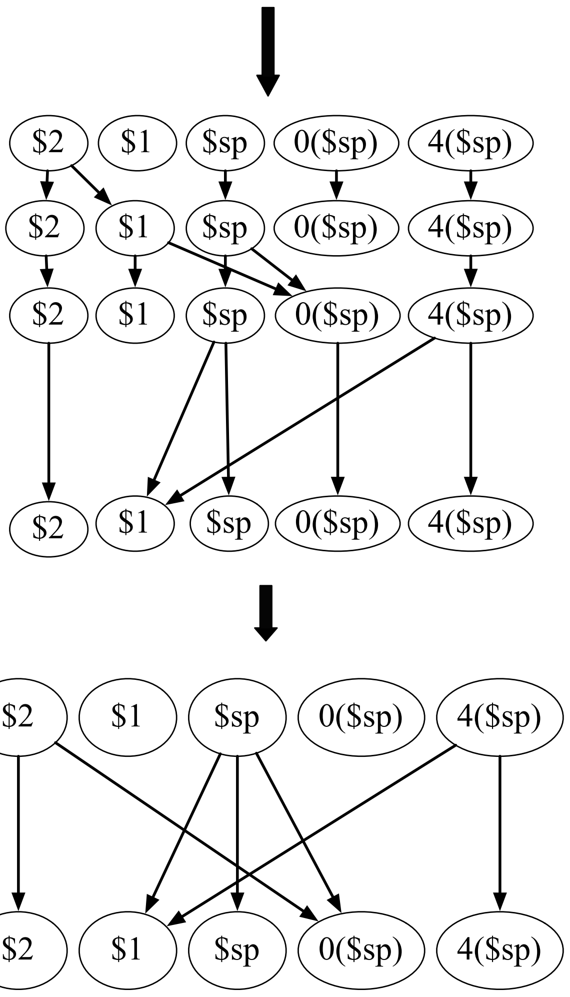

Figure 4: A basic block, its basic infection graph, and the bipartite graph representing the relationship between the final values of tag-bits and the initial values of tag-bits within the block. Only those bits that change or affect other bits are shown. Note that, despite the apparent complexity of the graph, only two nodes on the bottom half of the graph have edges into them from different nodes on the top half.

1. Memory references cannot be resolved or used by the `settor` instructions. Thus, tag-setting involving memory must be resolved separately.

2. The lack a temporary workspace to manipulate tags in means that dependencies can make tag-setting using `settor` instructions impossible. For instance, if the final tag of `$1` is dependent on the original tag of `$2`, and the final tag of `$2` dependent on the original tag of `$1`, we could not simply set the tags of these registers in arbitrary order.

7.3 Multi-graph Model

Consider the following assembly code:

```
lw $1, 4($sp)
add $1,$1,$2
mov $3,$1
add $sp,$sp,$4
lw $2,4($sp)
add $1,$2,$2
```

Several conflicts exist in this block. For instance, the value of the stack pointer changes in the midst of the block. Thus, the two references to `4($sp)` refer to two different spurious bits. The second reference must, thus, be delayed until after the value of `$sp` changes. Additionally, the values of $2 and $1 are used and set at various states, and ordering must be kept between them. In situations like these, the only solution is to effectively divide the basic block into several sub-blocks, and resolve each block, in order. So, in the case of this block, we would break the block into two - one containing the first thee instructions, and the other containing the final three. A representation of how this graph must thus be divided based on conflicts can be found in Figure 5. Each graph could then be parsed, and the needed tag-setting instructions inserted into the program code at the end of each sub-block. This yields the algorithm shown in Figure 6. While the complexity of this algorithm is high, annotation need only be done once after compilation, so the run-time of the annoator is irrelevant to program performance. Note that, however, compile time (including annotation) will increase.

8 Toolchain Flow

To implement our new ISA, the existing toolchain had to be either extended or modified to handle the new instructions we introduced. For reference, the

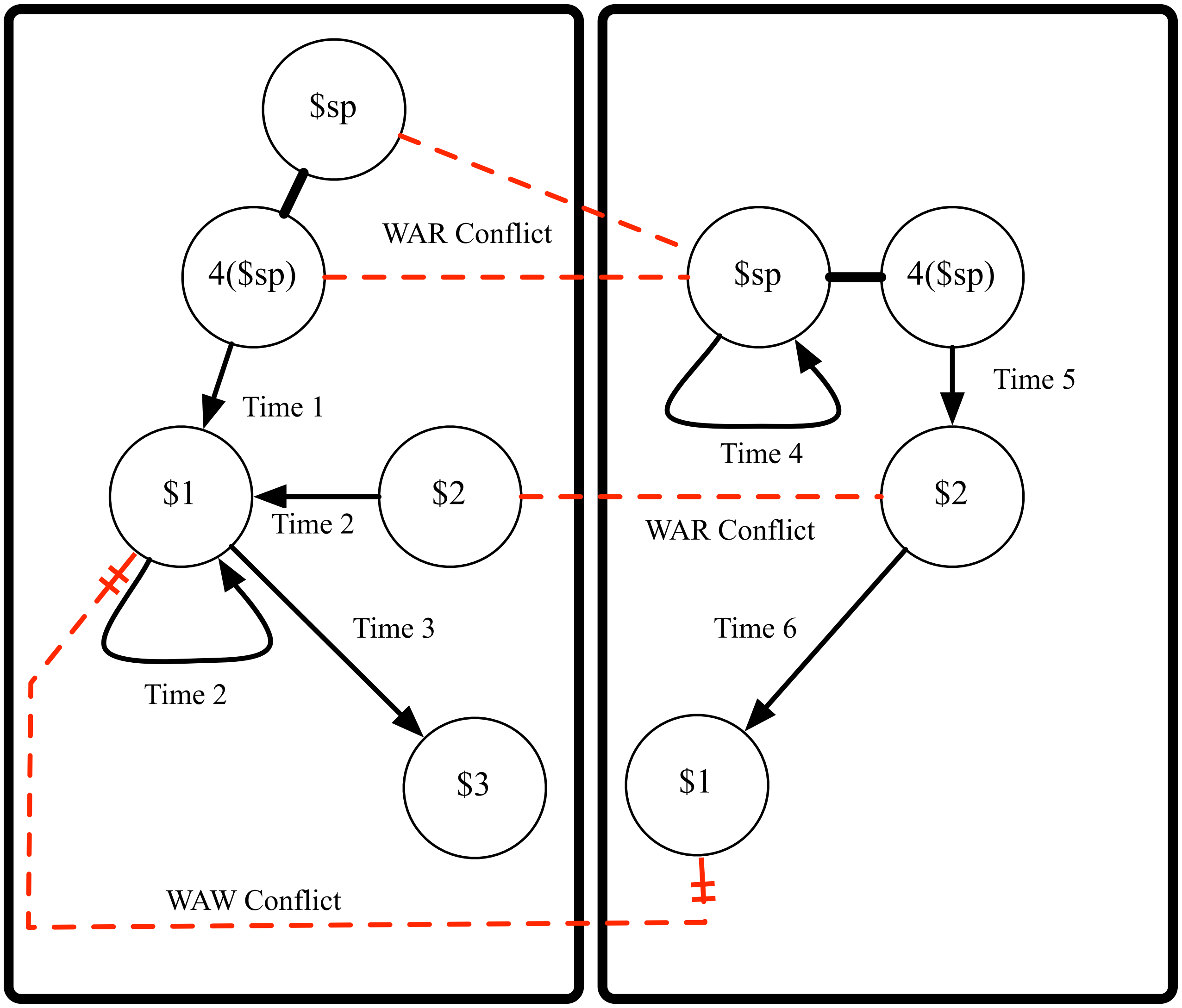

Figure 5: A graph representing how dependencies can be eliminated by cutting a basic infection graph into multiple graphs.

```
procedure Resolve($fwd, time, MaxTime$)                      ▷ Resolve all tags in a block
    $L \leftarrow \emptyset$                                 ▷ List of tuples (victim, infectors)
    for $i \leftarrow 1, MaxTime$ do
        for all vertices f in fwd do
            for all victims v of f at time i do
                $t \leftarrow (v, \emptyset)$
                for all victim $v'$ of v before i do
                    $t.infectors \leftarrow t.infectors + GetInfectors(v', forward, time)$   ▷ GetInfectors returns all infectors of $v'$
                end for
                $L \leftarrow L + t$
            end for
        end for
    end for
    $T \leftarrow \emptyset$
    for all $\ell \in$ L do
        $T \leftarrow T + Tags(\ell)$        ▷ Tags($\ell$) returns the appropriate tag-setting operation for victim and its infectors
    end for
    return $T$
end procedure
```

Figure 6: Algorithm for breaking dependencies, where fwd is a representation of the dependencies of eac individual bit, time is a representation of the basic infection graph, and MaxTime is the number of instructions in the block

standard process is in Figure 7. Our available options were thus modifying the compilation, assembly, and linking steps, or inserting new stages between existing stages. For this project, we chose the latter, due primarily to the difficulty of modifying the `gcc` suite of tools. With the new stages inserted, the flow of the extended toolchain is shown in Figure 8, with the stages and processes we added in dashed stroke.

The new actions are analysis, annotation, and preprocessing. The front end to all these actions is called `BBApp`, which coordinates the parsing and analysis of the initial file, the saving of intermediate results, the addition of new instructions, the preprocessing of the instructions, and the rewriting of the final file. These actions are described in more detail in the following sections. `BBApp` is implemented in Python, which enabled rapid development, in part by providing a rich standard library for both back-end features (i.e. regular expressions, dictionaries, lists, etc.) as well as front-end features (i.e. command line option parsing).

8.1 `BBApp` Actions

8.1.1 Parsing and Caching

Initially, `BBApp` parses the input assembly file into basic blocks. Once performed, `BBApp` caches this analysis so that subsequent actions on the same file can reuse the initial analysis, rather than re-creating it every run. Basic blocks are stored as a list of lines, along with starting and ending line numbers, and a name. Each assembly file is converted into a collection of these basic blocks (a list of blocks, along with supporting information). These structures are represented as Python classes. Caching is then easily accomplished by using Python's `pickle` module, which allows for pickling and unpickling of arbitrary Python objects to and from files. The main benefit of caching is that it allows `BBApp` to modify the assembly without changing the original input file but to still operate in distinct stages—only the cache is changed (until the final rewriting stage).

8.1.2 Annotation and Replacement

`BBApp` does annotation via a separate collection of functions, which take as input a basic block of as-

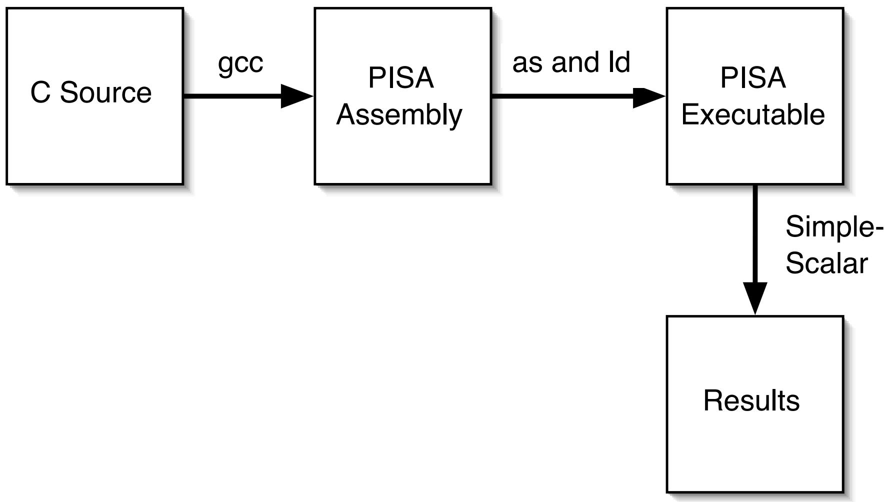

Figure 7: Original Toolchain Flow

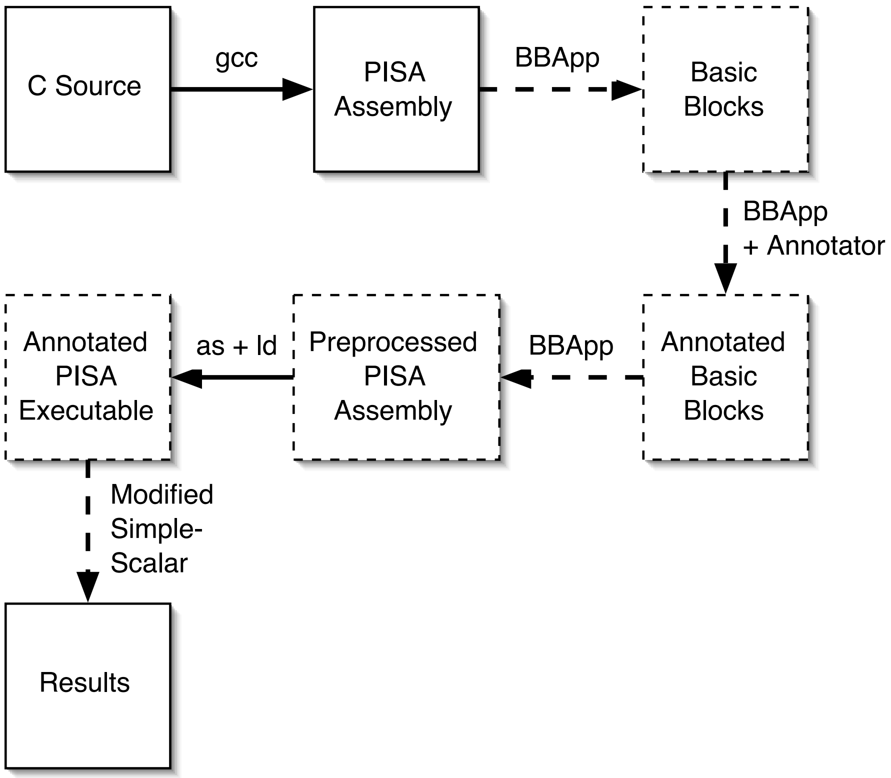

Figure 8: Modified Toolchain Flow

sembly code and return that basic block with new instructions interspersed. Full annotation of a file is thus accomplished by passing each basic block in turn to these annotation functions. The returned assembly code replaces the old contents of the original basic block, and line numbers are adjusted appropriately.

8.1.3 Preprocessing and Rewriting

For the preprocessing step, an instance of the preprocessing class is created. Upon construction, this class creates a collection of regular expressions for identifying our new instructions and their operands. These regular expressions are dynamically generated from a table specifying parameters such as opcode, arguments, argument types, etc. This approach allows us to easily modify the surface format and the encoding of each instruction without changing the preprocessing code directly; these encodings are described in Section 9. Preprocessing the full assembly file is then done by using an instance of this preprocessing class on each basic block.

Finally, the annotated and preprocessed file needs to be generated. Rewriting simply consists of writing the basic blocks in order, overwriting the original file (which is automatically backed up beforehand). Overwriting the original file simplifies the integration of our tools into existing automated build tools, such as Makefiles; `BBApp` can simply rewrite the assembly files before the assembly and linking stage, and the build tool does not have to deal with new file names.

9 ISA Implementation

This section will discuss the specific encoding scheme for our instructions, as well as some issues raised by our scheme.

9.1 Encoding

PISA instructions are double words. We make use of this fact to encode some additional flags on each instruction without compromising the space used for operands. For example, while registers, offsets, and immediates are encoded in the normal RS, RT, RD, IMM, etc. positions (PISA is a MIPS derivative), we use the annotation field to encode both the byte length for memory instructions (as seen in Table 2) as well as whether the instruction is modifying floating point or integer registers. In this way, we avoid having separate instructions for byte, half word, word, and double word variations, as well as integer and floating point variations, which would substantially increase the number of opcodes and instructions required.

The specifics of our scheme are the following: the annotation field is 16 bits wide, and we use the top 8 bits for the byte length specifier, and the bottom 8 for the integer/float/double distinction. The distinction between integer, single precision float, and double precision float operations is encoded using a value of 0 for integer, 1 for single precision floats, and 2 for double precision floats. Byte lengths are encoded directly; legal values are 1, 2, 4 and 8. Note that in the case of floating point memory operations, the specified byte length must match the floating point type; if not, the preprocessor flags the instruction as an error. An example encoding with all fields labeled is shown in Figure 9.

9.2 Preprocessing

As described in Section 8, the assembly files containing our new instructions must somehow be preprocessed to avoid the problem that the existing toolcain does not know about them. This preprocessing is achieved by directly encoding our instructions into hexadecimal and including this hexadecimal in the assembly file. The remainder of the toolchain then simply includes this data in the final output, and our modified SimpleScalar interprets them just as any other instruction in the program. An example of an instruction sequence before and after preprocessing can be seen in Table 3.

Table 3:
Preprocessing example.

Original Instructions	After Preprocessing
`add $1,$2,$3`	`add $1,$2,$3`
`sett $1,$2,$3`	`.long 0x000000b4`
	`.long 0x02030100`
`sub $1,$1,$4`	`sub $1,$1,$4`
`sett $1,$1,$4`	`.long 0x000000b4`
	`.long 0x01040100`
`sw $1,8($sp)`	`sw $1,8($sp)`
`setmt $1,8($sp),4`	`.long 0x040000e2`
	`.long 0x011d0008`

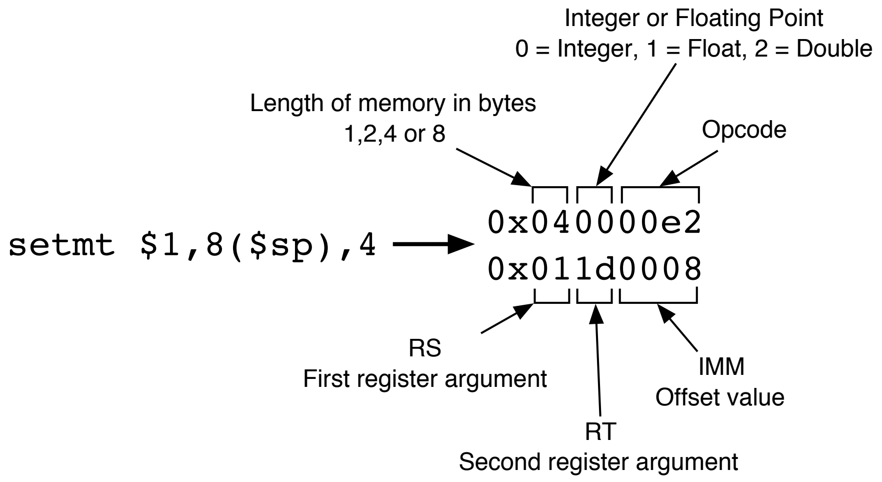

Figure 9: Example instruction encoding.

9.3 Issues

Because our tools are not part of the traditional toolchain, they face some unique issues, which were unfortunately not realized until late in the process. The two main issues are dealing with symbolic names in the assembly and dealing with libraries. Both of these, unfortunately, currently cause security holes in our scheme.

Symbolic names (for example, `stdout`) are a problem for our tools because they are not resolved to addresses until the assembly and linking stages of compilation. Since our tools operate one stage before these processes, they do not have the address information available to them. An assembler modified to handle our instructions would not have this problem — it would simply resolve names to addresses and encode our instructions as it does for any other instruction. One method we proposed for dealing with this problem was to compile the program normally first, and then use the `nm` program to lookup names in the final executable. We anticipated that this would work because most symbolic names are global variables in the data sections of the program, and thus their location would not change between an un-annotated binary and an annotated binary. Symbolic function names could change, but this could be resolved via counting inserted instructions.

Libraries are also a problem for our scheme; as discussed in Section 10.2.1, we have modified the `read()` system call to tag input data as spurious. However, this input data is moved, processed, and returned to the application via library functions such as `gets()` or `scanf()`. Because these functions are not included in the executable until the linking stage, they are not available for our toolchain to annotate. Instead, `glibc` would need to be annotated, and this annotated version could then be (statically) linked with compiled executables. There are several problems with this approach, however. The first is that compiling `glibc` is very tricky, even without modifying it at all. The second is the symbolic name problem again; however, it is harder to solve in this instance, because symbolic names in the library do not get resolved until the library is linked into a program (since the layout of the various functions and data is not known until that time). This issue is harder to solve using `nm`, since the libraries would have to be re-annotated on a per-program basis. As of now, we have no better proposal for solving this problem than abandoning our current infrastructure and hacking the `gcc/as/ld` toolchain.

10 SimpleScalar Modifications

We have modified the `sim-safe` version of the SimpleScalar simulator version 3.0. We chose `sim-safe` both because it is the simplest of the SimpleScalar family to modify and experiment with, and because it relieves us from worrying about how our new instructions are scheduled. We chose to extend PISA since RISC architectures lend themselves well to modifications. Another useful feature of PISA is its use of two words to encode instructions, which allows us to encode extra information (see Section 9).

Our modifications to the simulator fall into two categories—hardware, and software.

10.1 Hardware

To support tagging of general purpose registers, floating point registers and the HI and LO registers, we have added two 32-bit registers and two single bit registers to the register file. Although general purpose register zero is always set to zero and does not need a spurious tag, there is no other register with a constant tag value. Therefore register tagging has to spill to include at least one more bit tag for either the HI or LO registers.

Our current memory modifications implement support for tagging by allocating enough extra memory to associate a bit with each byte of main memory. This naïve implementation incurs a 12% memory overhead. Potential improvements include structuring the tag memory in a hierarchical fashion, which reduces the memory overhead to an average of 1.0% [9]. The hierarchichal tagging would, at the lowest resolution, tag virtual pages and, at the highest resolution, tag bytes, with optional levels of multi-byte tagging in between.

10.2 Software

The modular nature of SimpleScalar allowed for most modifications to be relegated to the machine.def file which defines and implements the PISA instructions. A total of 24 new opcodes were defined (each opcode implementing a single addressing mode, and each opcode supporting both floating point and general purpose register encodings). The instructions corresponding to these new opcodes are shown in Table 2. Original instructions were left unmodified.

10.2.1 System Calls and Library Functions

System calls are the entry point for user data which may in turn be propogated by library functions. To tag spurious input data, modifications must be made at the system call level. We have modified the `read()` system call in SimpleScalar. `read()` reads data and also tags all the read bytes as spurious in memory.

11 Extending the system to real hardware

We decided to add additional hardware to the simulator to isolate and simplify our modifications. The system can actually be implemented with a standard hardware setup. For example, three general purpose registers can be reserved for tagging of registers, and a piece of memory can be reserved to store tagging information for the rest of the memory. These modifications however constrain the compiler to use a restricted register set. To implement this scheme we would have to modify the PISA compiler to avoid the tag registers, modify a key set of system calls that get input from the user, and modify standard libraries such as `libc` that provide wrappers for the above system calls. One merit of our system is that the modified simulator effectively implements the same framework with minimal modification to the existing toolchain.

The full system could be implemented at the compiler level with small modifications to the operating system and no modifications to the hardware. Library and operating system modifications would only take effect for those programs that have been annotated by conditioning the modified sections on whether the proccess is an annotated binary or not.

12 Experiments

To understand the performance overhead incurred by our scheme, we have run portions of the SPEC-CPU2000 benchmark suite on our modified version of SimpleScalar. Because we chose to modify `sim-safe`, rather than the more powerful `sim-outorder`, the information we can glean from these tests is somewhat less than what would ideally be available. In particular, the main performance metric available to us is instruction count, whereas instructions-per-cycle would likely be a more discriminating metric. For each SPEC benchmark, we have run an unannotated version, a naïvely annotated version, and an optimally annotated version. The results of these runs can be found in Table 4. We present the unannotated instruction count, the percent of unannotated IC made up by loads and stores, the percent that IC increased with annotation, and the percent that annotated IC was reduced by optimization.

Given our results, it appears that instruction count increases roughly 60% from unannotated to

Table 4: Experimental Results

Benchmark	Original IC	% Loads/Stores	% Added by Annotation	% Removed by Optimitzation
gzip, log-input	33642800312	29%	61%	11%
gzip, graphic-input	80085017764	32%	59%	9%
mcf	202016915	39%	59%	5%

naïvely tagged, and that an optimized tagger reduces this overhead by several percent. The variance in reduction through optimization appears to be correlated with the number of loads and stores, which is expected, since most loads require the breaking of a basic block into sub-blocks (note that `sim-safe` does not provide information on loads alone, which are more significant). Given more time, it would be beneficial to run on more benchmarks to better understand the relationship between loads and optimization.

13 Conclusion

Dynamic Information Flow Tracking, proposed by Suh et. al. [9], is effective in stopping many of the most common exploit methods in use today, while causing very small performance and memory overheads. However, it requires specialized hardware to compute and check tags in parallel with regular instructions. In this paper, we have explored options for implementing the same security policy, but at the level of the instruction set architecture.

The potential advantage of our system is the ability to secure specific programs with minimal modifications. The idea of selectively annotating an existing binary to make it safe to run is a powerful one. The performance of existing applications is *not affected* and only programs requiring extra security can be annotated. In this way, performance loss is traded for security *selectively* and existing third party programs can be made secure without the use of the original source code.

Were this scheme to be implemented by a compiler, it would have the further advantage of the compiler's global knowledge of the program's execution flow. So, for instance, rather than being restricted to basic blocks, a compiler could identify a block as secure – one in which no data from I/O could possibly enter – and avoid tag-propogating and checking instructions altogether. Further, because tag-setting instructions operate outside of the critical path of normal instructions, it would likely be possible for a compiler to schedule these instructions to fill in stall cycles and thus increase IPC to make up for the added number of instructions.

References

[1] Anonymous. Once upon a free(). *Phrack*, (57), 2001.

[2] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, (56), 2000.

[3] c0ntex. Bypassing non-executable-stack during exploitation using return-to-libc. *www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf*, unknown.

[4] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. 2005.

[5] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. 1998.

[6] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. 2004.

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

[8] Scut. Exploiting format string vulnterabilities. *http://www.team-teso.net/articles/verbformatstring*, 2001.

[9] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[10] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. 2005.