2008 IEEE Symposium on Security and Privacy

# Preventing memory error exploits with WIT

Periklis Akritidis * Cristian Cadar* Costin Raiciu* Manuel Costa Miguel Castro
Microsoft Research, Cambridge UK

## Abstract

*Attacks often exploit memory errors to gain control over the execution of vulnerable programs. These attacks remain a serious problem despite previous research on techniques to prevent them. We present* Write Integrity Testing *(WIT), a new technique that provides practical protection from these attacks. WIT uses points-to analysis at compile time to compute the control-flow graph and the set of objects that can be written by each instruction in the program. Then it generates code instrumented to prevent instructions from modifying objects that are not in the set computed by the static analysis, and to ensure that indirect control transfers are allowed by the control-flow graph. To improve coverage where the analysis is not precise enough, WIT inserts small guards between the original program objects. We describe an efficient implementation with optimizations to reduce space and time overhead. This implementation can be used in practice because it compiles C and C++ programs without modifications, it has high coverage with no false positives, and it has low overhead. WIT's average runtime overhead is only 7% across a set of CPU intensive benchmarks and it is negligible when IO is the bottleneck.*

## 1. Introduction

Attackers can exploit software errors to control vulnerable programs. Programs written in unsafe languages like C and C++ are particularly vulnerable because of memory errors, for example, buffer overflows and underflows [5, 35], dangling pointers [7], and double frees [26]. Despite previous research on techniques to prevent these attacks (see Section 8), at least 40% of the vulnerabilities published by US-CERT in the last six months are memory errors. We believe there are two reasons for this: techniques that are used to prevent these attacks fail to prevent many attacks; and most techniques are not used because they have high overhead or they require non-trivial changes to the source code or the language runtime.

We present *Write Integrity Testing* (WIT), a new technique to prevent memory error exploits that addresses the problems with previous approaches. WIT can be applied to C and C++ programs without modifications, it does not require changes to the language runtime, it has high coverage with no false positives, and it has low overhead.

At compile time, WIT uses interprocedural points-to analysis [23] to compute the control-flow graph and the set of objects that can be written by each instruction in the program. At runtime, WIT enforces *write integrity*, that is, it prevents instructions from modifying objects that are not in the set computed by the static analysis. Additionally, WIT inserts small *guards* between the original objects in the program. Since the guards are not in any of the sets computed by the static analysis, this allows WIT to prevent sequential overflows and underflows even when the static analysis is imprecise. WIT also enforces *control-flow integrity* [6, 28], that is, it ensures that the control flow transfers at runtime are allowed by the control-flow graph computed by the static analysis.

WIT uses the points-to analysis to assign a *color* to each object and to each write instruction such that all objects that can be written by an instruction have the same color. It instruments the code to record object colors at runtime and to check that instructions write to the right color. The color of memory locations is recorded in a *color table* that is updated when objects are allocated and deallocated. Write checks look up the color of the memory location being written in the table and check if it is equal to the color of the write instruction. This ensures write integrity.

WIT also assigns a color to indirect call instructions and to the entry points of functions that may be called indirectly such that all functions that may be called by the same instruction have the same color. WIT instruments the code to record function colors in the color table and to check indirect calls. The indirect call checks look up the color of the target address in the table and check if it matches the color of the indirect call instruction. These checks together with the write checks ensure control-flow integrity. Control-flow integrity prevents the attacker from bypassing our checks and provides an effective second line of defense against at-

*Work done while an intern at MSR Cambridge. Periklis Akritidis is affiliated with the University of Cambridge, Cristian Cadar with Stanford University, and Costin Raiciu with University College London.

978-0-7695-3168-7 /08 $25.00 © 2008 IEEE
DOI 10.1109/SP.2008.30

tacks that are not detected by the write checks.

We developed several optimizations to reduce the space and time overhead of our implementation. First, we use static analysis to determine memory accesses and objects that are *safe*, that is, accesses that cannot violate write integrity and objects that only have safe accesses. We only instrument unsafe writes and we assign the same color to all safe objects. This reduces the number of write checks and also the overhead to maintain the color table. Additionally, it reduces the number of bits required to represent colors. One byte was sufficient to represent colors in all our experiments. Second, we use a compact representation for the color table that can be looked up efficiently. The color table maintains one byte to represent the color of an eight byte chunk of memory, which reduces space overhead to approximately 12.5%. Third, we reduce the cost of updating color table entries on function calls. Since most local variables are safe, we only update entries for guards and unsafe variables on function entry and we reset these entries to the color of safe objects on function exit.

We evaluated the coverage of WIT using a suite of attacks to test buffer overflow prevention techniques [43] and five real attacks on SQL server, libpng, ghttpd, nullhttpd, and stunnel. WIT can prevent all these attacks.

We also evaluated the overhead introduced by WIT using SPEC CPU and Olden benchmarks. WIT's average overhead is 7% and the maximum overhead is 25%. In a Web server running the SPEC Web 1999 benchmark, the overhead is even lower: response times increase by 0.2% and peak throughput decreases by 4.8%. We believe that WIT can be used in practice to protect software from attacks that exploit memory errors.

## 2. Overview

WIT has both a compile-time and a runtime component. We will use the example in Figure 1 to illustrate how both components work. The example is a simplified Web server with a buffer overflow vulnerability. It is inspired by a vulnerability in nullhttpd [2] that can be exploited to launch a non-control-data attack [15].

When the Web server in Figure 1 receives a CGI command, it calls `ProcessCGIRequest` with the message it received from the network and its size as arguments. The function copies the command from the message to the global variable `cgiCommand` and then calls `ExecuteRequest` to execute the command. The variable `cgiDir` contains the pathname of the directory with the executables that can be invoked by CGI commands. `ExecuteRequest` first checks that `cgiCommand` does not contain the substring `"\\.."` and then it concatenates `cgiDir` and `cgiCommand` to obtain the pathname of the executable to run. The problem is that there is a buffer

```
1:  char cgiCommand[1024];
2:  char cgiDir[1024];
3:
4:  void ProcessCGIRequest(char* msg, int sz) {
5:    int i=0;
6:    while (i < sz) {
7:      cgiCommand[i] = msg[i];
8:      i++;
9:    }
10:
11:   ExecuteRequest(cgiDir, cgiCommand);
12: }
```

**Figure 1. Example vulnerable code: simplified Web server with a buffer overflow vulnerability.**

overflow vulnerability in lines 5 — 9: if the message is too long, the attacker can overwrite `cgiDir`. This allows the attacker to run any executable (for example, a command shell) with the arguments supplied in the request message. This is a non-control-data attack [15]: it does not violate control-flow integrity.

We start by using points-to analysis [23] to compute the set of objects that can be modified by each instruction in the program. For the example in Figure 1, the analysis computes the set $\{$`i`$\}$ for the instructions at lines 5 and 8, and the set $\{$`cgiCommand`$\}$ for the instruction at line 7.

To reduce space and time overhead at runtime, we also perform a *write safety* analysis to compute instructions and objects that are *safe*. An instruction is safe if it cannot violate write integrity and an object is safe if all instructions that can modify the object (according to the points-to analysis) are safe. In our example, the write safety analysis determines that instructions 5 and 8 are safe because they can only modify `i` and, therefore, `i` is safe. It also determines that the arguments to `ProcessCGIRequest` are safe. In contrast, instruction 7 is not safe because it may modify objects other than `cgiCommand` depending on `i`'s value.

The results of the points-to and write safety analysis are used to assign a color to each write instruction and a color to each object in the program. We attempt to assign distinct colors to each unsafe object under the constraint that each instruction must have the same color as the objects it can write. We assign color 0 to all safe objects and all safe instructions to reduce the number of bits required to represent colors. In our example, variables `msg`, `sz`, and `i` and instructions 5 and 8 are assigned color 0 because they are safe. We assign color 3 to variable `cgiCommand` and instruction 7, and color 4 to variable `cgiDir`.

To reduce the false negative rate due to imprecision of the points-to analysis, we insert small guards between the unsafe objects in the original program. Guard objects have color 0 or 1. These colors are never assigned to unsafe instructions in the program to ensure that WIT can detect attempts to overwrite guards or safe objects.

We also use the points-to analysis to compute the functions that can be called by each indirect call instruction in

the program. We assign colors to indirect call instructions and to the functions they can call. We attempt to assign distinct colors to each function while ensuring that each instruction and the functions it can call have the same color. The set of colors assigned to functions is disjoint from the set of colors assigned to unsafe objects, to safe objects, and to guards. This prevents unsafe instructions from overwriting code and prevents control transfers outside code regions.

WIT adds extra compilation phases that insert instrumentation to enforce write integrity and control flow integrity. There are four types of instrumentation: to insert guards, to maintain the color table, to check writes, and to check indirect calls. Guards are eight bytes long. In our example, we instrument the code to add guards just before `cgiCommand`, between `cgiCommand` and `cgiDir`, and just after `cgiDir`. We do not insert guards around the arguments to `ProcessCGIRequest` and local variable `i` because they are safe.

WIT uses the color table to record the color of each memory location. When an object is allocated, the instrumentation sets the color of the storage locations occupied by the object to its color. In our example, WIT adds instrumentation at the beginning of main to set the color of the storage locations occupied by `cgiCommand` to 3, the color of the storage for `cgiDir` to 4, and the color of the storage for the guards around them to 0.

We use an optimization to reduce the cost of updating the color table. We initialize the color table to 0 for all memory locations and we do not update the color table when safe objects are allocated on the stack. Instead, we only update the colors for locations corresponding to unsafe objects on function entry. On function exit, we reset color table entries that we updated on function entry to 0. Therefore, there is no instrumentation to update the color table on function entry or exit for `ProcessCGIRequest`.

The checks on writes compare the color of the instruction performing the write to the color of the storage location being written. If the colors are different, they raise a security exception. The color of each instruction is known statically and write checks use the color table to lookup the color of the location being written. We do not insert write checks for safe instructions to improve performance. In the example in Figure 1, WIT adds write checks only before instruction 7 to check if the location being written has color 3. It does not add write checks before lines 5 and 8 because these instructions are safe.

WIT also records the color of each function that can be called indirectly in the color table. It inserts instrumentation to update the color table at program start-up time and to check the color table on indirect calls. The indirect call checks compare the color of the indirect call instruction and its target. If the colors are different, they raise an exception. There are no indirect calls in our example.

WIT can prevent all attacks that violate write integrity but the number of attacks that violate this property depends on the precision of the points-to analysis. For example if two objects have the same color, we may fail to detect attacks that use a pointer to one object to write to the other. Our results show that the analysis is sufficiently precise to make this hard. Additionally, WIT can prevent many attacks regardless of the precision of the points-to analysis. For example, it prevents: attacks that exploit buffer overflows and underflows by writing elements sequentially until an object boundary is crossed (which are the most common); attacks that overwrite any safe object (which include return addresses, exception handler pointers, and data structures for dynamic linking); and attacks that corrupt heap management data structures.

Control-flow integrity provides an effective second line of defense when the write checks fail to detect an attack. WIT prevents all attacks that violate control-flow integrity but the number of attacks that violate this property also depends on the precision of the points-to analysis. For example, if many functions have the same color as an indirect call instruction, the attacker may be able to invoke any of those functions. In the worst case, the analysis may assign the same color to all functions that may be called indirectly. Even in this worst case, an attacker that corrupts a function pointer can only invoke one of these functions. Furthermore, these functions do not include library functions invoked indirectly though the dynamic linking data structures. Therefore, the attacker cannot use a corrupt function pointer to jump to library code, to injected code or to other addresses in executable memory regions. This makes it hard to launch attacks that subvert the intended control flow, which are the most common.

WIT does not prevent out-of-bounds reads. These can lead to disclosure of confidential data but it is hard to exploit them to execute arbitrary code without violating write integrity or control-flow integrity in the process. Therefore, we chose not to instrument reads to achieve lower overhead.

WIT can prevent attacks on our example Web server. The write check before line 7 fails and raises an exception if an attacker attempts to overflow `cgiCommand`. When `i` is 1024, the color of the location being written is 0 (which is the color of the guard) rather than 3 (which is the color of `cgiCommand`). Even without guards, WIT would be able to detect this attack because the colors of `cgiCommand` and `cgiDir` are different.

## 3. Static analysis

We implemented the points-to and the write safety analysis using the Phoenix compiler framework [30]. These analysis operate on Phoenix's medium level intermediate representation (MIR), which enables them to be applied to differ-

ent languages and target architectures. Figure 2 shows the MIR for the vulnerable C code in Figure 1.

```
     _i     = ASSIGN 0                              #1
$L6: t273   = COMPARE(LT) _i, _sz                   #2
     CONDITIONALBRANCH(True) t273, $L8, $L7         #3
$L8: t278   = ADD _msg, _i                          #4
     t276   = ADD &_cgiCommand, _i                  #5
     [t276] = ASSIGN [t278]                         #6
      _i    = ADD _i, 1                             #7
     GOTO $L6                                       #8
$L7: CALL &_ExecuteRequest,&_cgiDir,&_cgiCommand
```

**Figure 2. Example vulnerable code in medium-level intermediate representation (MIR).**

We use an inter-procedural points-to analysis due to Andersen [8] that is flow and context insensitive but scales to large programs. It computes a points-to set for each pointer, which is the set of logical objects the pointer may refer to. The logical objects are local and global variables and dynamically allocated objects (for example, allocated with `malloc`). We use a single logical object to represent all objects that are dynamically allocated at the same point in the program but we do cloning of simple allocation wrappers to improve analysis precision. Our implementation is similar to the one described in [23] but it is field-insensitive rather than field-based (i.e., it does not distinguish between the different fields in a structure, union, or class). We use Phoenix to compile each source file to MIR and write points-to constraints to a file. The analysis reads the constraints file, computes the points-to sets, and stores them in a file.

The analysis assumes that the relative layout of independent objects in memory is undefined [9]. For example in Figure 2, it assumes that correct programs will not use `t276`, which is a pointer into the `cgiCommand` array, to write to `cgiDir`. Compilers already make this assumption when implementing standard optimizations. Under this assumption, the analysis is conservative: a points-to set includes all objects that the pointer may refer to in executions that do not violate memory safety (but it may include additional objects). Therefore, WIT has no false positives.

The write safety analysis classifies instructions as safe or unsafe: an instruction is marked safe if it cannot violate write integrity. The analysis marks safe all MIR instructions without an explicit destination operand or whose destination operand is a temporary, a local variable, or a global. These instructions are safe because they either modify registers or they modify a constant number of bytes starting at a constant offset from the frame pointer [1] or the data segment. Assuming the constants generated by the compiler and linker are correct, the write safety analysis does not introduce false negatives because control-flow integrity prevents the attacker from bypassing our checks or changing the data segment or the frame pointer. In the example in Figure 2, all instructions are safe except instruction 6.

In addition, the write safety analysis runs a simple intra-procedural pointer-range analysis to compute writes through pointers that are always in bounds. The instructions that perform these writes are marked safe. Our pointer-range analysis is a simplified version of the one described in [47]. It collects sizes of aggregate objects (e.g., structs) and arrays that are known statically. Then it uses symbolic execution to compute the minimum size of the objects each pointer can refer to and the maximum offset of the pointer into these objects. When the analysis cannot compute this information or the offset can be negative, it conservatively assumes a minimum size of zero. Our current implementation can track constant offsets and offsets that can be bound using Phoenix's built-in value range information for numeric variables. Given information about the minimum sizes, the maximum offsets, and the size of the intended write, the analysis checks if writes through the pointer are always in bounds. If they are, the corresponding instruction is marked safe.

While making the global pass over all source files to collect constraints for the points-to analysis, we also run the write safety analysis. We write *unsafe pointers* to a file. A pointer is unsafe if it is dereferenced for writing by an unsafe instruction.

We use the results of the points-to and write safety analysis to assign colors to objects and to unsafe instructions. We use an iterative process to compute *color sets*, which include objects and unsafe pointer dereferences that must be assigned the same color because they may alias each other. Initially, there is a separate color set for each points-to set of an unsafe pointer: the initial color set for a points-to set $p \rightarrow \{o_1, ..., o_n\}$ is $\{[p], o_1, ..., o_n\}$. Then we merge color sets that intersect until we reach a fixed point. We assign a distinct color to each color set: we assign this color to all objects in the color set and all instructions that write pointer dereferences in the set. All the other objects in the original program are assigned color zero. By only considering points-to sets of unsafe pointers when computing colors, we reduce the false negative rate and the overhead to maintain the color table.

WIT uses a similar algorithm to assign colors to functions that may be called indirectly. The differences are that this version of the algorithm iterates over the points-to sets of pointers that are used in indirect call instructions (except indirect calls to functions in dynamically linked libraries), and that it only considers the objects in these sets that are functions. We can exclude indirect calls to library functions because they use a pointer that our write checks protect from

[1] The frame pointer is safe even with recursion and calls to `alloca` because there is a guard page at the end of the stack to prevent stack overflows. The prologues of functions with frames larger than a page and `alloca` check if the pages they need are resident. Therefore, they fault on the guard page to trigger stack growth when necessary. If the operating system cannot grow the stack, it raises an exception.

being corrupted by the attacker. We assign a different color to each color set. These colors are different from 0, 1, and the colors assigned to unsafe objects. The rest of the code is assigned color zero.

## 4. Instrumentation

We implemented WIT for 32-bit x86 machines running Windows. We chose this architecture and operating system because they are the most common today but it should be easy to retarget our implementation to other architectures and operating systems. We used several Phoenix plugins [30] to generate WIT's instrumentation. This section starts by describing the color table. Then it explains in more detail how we instrument the code.

### 4.1. Color table

WIT maintains a color table that maps memory addresses to colors. The color table must cover the whole user virtual address space and it is accessed often by write and indirect call checks. To achieve low space and time overhead, we designed the color table to be compact and to enable efficient lookups and updates.

To keep the color table small, we divide the virtual memory of the instrumented program into aligned eight-byte slots. The color table is implemented as an array with an eight-bit color identifier for each of these slots. Therefore, it introduces a space overhead of only 12.5%.

We are able to record a single color for each eight-byte slot because we generate code such that no two objects with distinct colors ever share the same slot. It is easy to enforce this requirement for heap objects because they are eight-byte aligned and for functions because they are 16-byte aligned. But since the stack and data sections are only four-byte aligned in 32-bit x86 architectures, we cannot currently force eight byte alignment of objects in these sections without introducing runtime overhead.

Instead, we force unsafe objects and guard objects in the stack and data sections to be four-byte aligned and we insert a four-byte aligned pad after unsafe objects. For an unsafe object of size $s$, the pad is eight-bytes long if $\lceil s/4 \rceil$ is even and four-bytes long if $\lceil s/4 \rceil$ is odd. We set $\lceil s/8 \rceil$ color table entries to the color of the unsafe object when the pad is four-bytes long and $\lceil s/8 \rceil + 1$ when the pad is eight-bytes long. We should be able to reduce the space overhead when targeting 64-bit x86 architectures because the stack and data sections are eight-byte aligned in these architectures.

Figure 3 shows how padding works. Depending on the alignment at runtime, the pad gets the color of the unsafe object, the guard, or both. All these configurations are legal because the pads and guards should not be accessed by correct programs and the storage locations occupied by unsafe objects are always colored correctly. Conceptually, the pads allow the guards to "move" to ensure that they do not share a slot with the unsafe objects.

Since our points-to analysis does not distinguish between different fields in objects and between different elements in arrays, we always assign the same color to all the elements of an array and to all the fields of an object. Therefore, it is not necessary to change the layout of arrays and objects, which is important for backwards compatibility.

We only require eight bits to represent colors because the write safety analysis is very effective at reducing the number of objects that we must assign colors to. However, it is possible that more bits will be required to represent colors in very large programs. If this ever happens, there are several things we can do. For example, we can increase the size of color table entries to 16-bits and increase memory slot sizes to 16-bytes, or use 8-bit color identifiers at the expense of worse coverage.

The color table can be accessed efficiently. Since there are 2 GB of virtual address space available for the user in Windows XP and Windows Vista, we allocate 256 MB of virtual address space for the color table [2]. We rely on the operating system to allocate physical pages for the color table on demand when they are first accessed. The base of the color table is currently at address 40000000h. So to compute the address of the color table entry for a storage location, we take the address of the storage location, shift it right by three, and add 40000000h.

The base of the color table can be at a different address. For example, we changed the linker to place the base of the table at address 0h. This reduces the number of bytes needed to encode the instructions that access the table because we can omit the large constant. However, it had little impact on the runtime overhead in our experiments.

To protect the color table from being overwritten by an attacker, we read-protect the pages in the table that contain the entries for the virtual address range occupied by the table itself. With the base of the table at 40000000h, we protect the pages in the address range 48000000h to 4A000000h to prevent reads and writes. Since we add checks before unsafe writes and control-flow integrity ensures that the attacker cannot bypass these checks, the attacker cannot overwrite the color table because the write check would trigger a read fault on the protected address range. This technique was first described in [44].

### 4.2. Inserting guards

We insert small guards before and after unsafe objects in the vulnerable program. This improves WIT's coverage

[2] It is possible to use a boot option to increase the user virtual address space to 3 GB. In this case, we need to allocate more virtual address space for the color table.

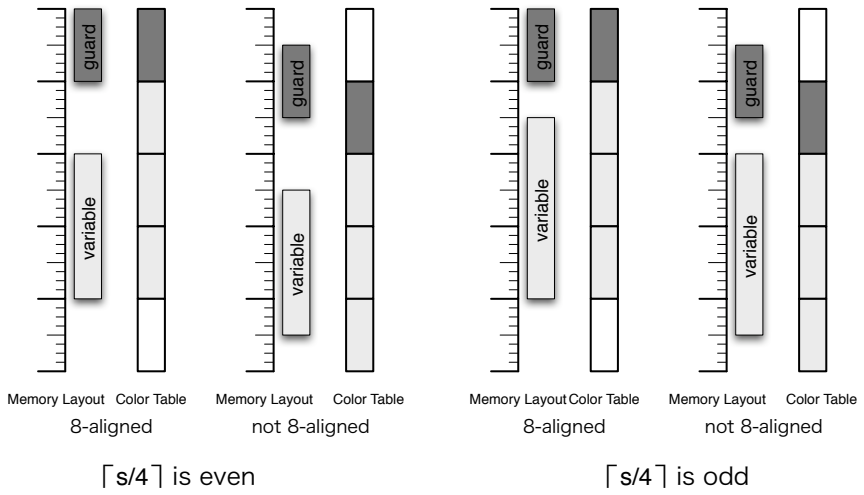


**Figure 3. Ensuring that two objects with distinct colors never share the same eight-byte slot. The pad after unsafe objects takes the color of the guard, the unsafe object, or both depending on the alignment. The lowest addresses are at the bottom of the figure.**

when the points-to analysis is imprecise while adding little runtime overhead. WIT is guaranteed to detect overflows and underflows that write array elements sequentially until an object boundary is crossed, which are the most common.

The guards are eight-bytes long to match the size of the slots that we record colors for in the color table. The instrumentation to insert these guards is different for the stack, heap, and global data sections.

To insert guards in the stack, we replace the compiler phase that lays out local variables in a stack frame by our implementation. We segregate safe local variables from unsafe ones to reduce the space overhead. First, we allocate contiguous storage for the safe local variables. Then we allocate storage for the guards, pads, and unsafe local variables. This allows us to insert only $n+1$ guards and pads for $n$ unsafe local variables: the guard that prevents overflows of a variable prevents underflows of the next variable.

In the rare case where a function argument is written by an unsafe instruction, we cannot easily insert guards and pads around it. Therefore, we copy the argument to a local variable and rewrite the instructions to refer to the copy. This local variable is marked unsafe and we insert guards and pads around it.

We mark all heap-allocated objects as unsafe but we do not insert pads or guards around them. The standard heap allocator in Windows Vista, Windows XP SP2, and Windows 2003 inserts an eight-byte header before each allocated object. We use this header as a guard by simply setting its color to 1 in the color table. Since heap objects and headers are eight-byte aligned, we do not need pads either. This optimization avoids space overhead, which could be significant for programs with many small allocations. In systems with different heap allocators, we can achieve similarly low overhead by modifying the allocator.

We add guards and pads between all variables in the `.data` section and `.bss` sections but not in the read-only data section (`.rdata`). We could use the same optimizations that we used for the stack but our results show that they would have little impact on overall performance when applied to globals.

We plan to implement an optimization that avoids the need for most guards by laying out stack and global objects such that adjacent objects have different colors.

### 4.3. Maintaining the color table

We rely on the operating system to initialize color table pages to zero when they are first accessed. This ensures security by default: unsafe writes and indirect calls to an address are not allowed unless we explicitly set the corresponding color table entry.

We initialize the color table entries for global variables and their guards at program start up. We use color zero for guards of global variables. We also initialize the color table entries corresponding to the first instructions of allowed indirect call targets at program start up. But we update the color table dynamically when objects are allocated on the stack or the heap.

We use an optimization to reduce the cost of updating the color table when we allocate a new stack frame. Instead of updating the color table entries for all objects in the stack frame, we only update entries corresponding to unsafe local variables and their guards on function entry. On function exit, we reset the entries that we updated on function entry to zero. This works because all safe objects have color zero, which is the initial color of all color table entries. We use

color zero for guards of unsafe local variables.

We instrument function prologues and epilogues to set and reset color table entries. This instrumentation is added after the phase that lays out the stack frames. Therefore, it is dependent on the target architecture. For example, we add the following code sequence to the prologue of a function with a single unsafe local variable with 12 bytes:

```
push ecx                                    # 1 byte
lea ecx,[esp+3Ch]                           # 4 bytes
shr ecx,3                                   # 3 bytes
mov dword ptr [ecx+40000000h],00020200h #10 bytes
pop ecx                                     # 1 byte
```

This sequence saves `ecx` on the stack to use it as a temporary. Then it loads the address of the first guard into `ecx` and shifts it by three to obtain the index of the guard's color table entry. It uses this index to set the color table entries to the appropriate colors. We set one color table entry for each guard. For an unsafe object of size $s$, we set $\lceil s/8 \rceil$ color table entries when $\lceil s/4 \rceil$ is odd and $\lceil s/8 \rceil + 1$ when $\lceil s/4 \rceil$ is even (see section 4.1). We use 2-byte and 4-byte moves to reduce the space and time overhead of the instrumentation whenever possible. In our example, the `mov` updates the four color table entries: the entries corresponding to guard objects are set to 0 and those corresponding to the unsafe local variable are set to two. The base of the color table is at address 40000000h. If the base was at address 0h the `mov` would be 6 bytes long. The final instruction restores the original value of `ecx`. The instrumentation for epilogues is identical but it sets the color table entries to zero.

An alternative would be to update color table entries only on function entry for all objects in the stack frame. This alternative adds significantly higher overhead because on average only a small fraction of local variables are unsafe. Additionally, WIT incurs no overhead to update the color table when functions have no unsafe locals or arguments, which is common for functions that are invoked often.

We also update the color table when heap objects are allocated or freed. We instrument the code to call wrappers of the allocation functions, for example, `malloc` and `calloc`. These wrappers receive the color of the object being allocated as an additional argument. They call the corresponding allocator and then set the color table entries for the allocated object to the argument color. They set $\lceil s/8 \rceil$ color table entries for an object of size $s$. They also set the color table entries for the eight-byte slots immediately before and after the object to color one. These two slots contain a chunk header maintained by the standard allocator in Windows. We use these headers as guards. We also replace calls to `free` by calls to a wrapper. This wrapper sets the color table entries of the object being freed to zero and then invokes free. We use a different color for guards in the heap to detect some invalid uses of free (as explained in the next section).

## 4.4. Instrumenting writes

We only check writes performed by unsafe instructions. These checks lookup the color of the destination operand in the color table. Then they compare this color with the color of the instruction. If the colors are the same, they allow the write to proceed. Otherwise, they generate an exception. We insert write checks in the MIR, which makes this instrumentation phase independent of the target architecture. For example, we add the following instrumentation before the unsafe write in Figure 2:

```
      t300 = SHIFTRIGHT &[t276], 3
      t301 = COMPARE(EQ) [t300+40000000h], 3
      CONDITIONALBRANCH(True) t301, $L11, $L10
$L10: BREAK
$L11: [t276] = ASSIGN [t278] # unsafe write
```

where `t300` and `t301` are fresh temporaries, and the unsafe write has color 3. Phoenix lowers this into the following sequence of x86 assembly code:

```
      lea edx,[ecx]                    # 2 bytes
      shr edx,3                        # 3 bytes
      cmp byte ptr [edx+40000000h],3 # 7 bytes
      je  $L11                         # 2 bytes
      int 3                            # 1 byte
$L11: mov byte ptr [ecx],ebx      #unsafe write
```

where register ecx holds the target address of the unsafe write and the color table starts at 40000000h. This code sequence loads the address of the destination operand into a register, and shifts the register right by three to obtain the operand's index in the color table. Then it compares the color in the table with the color of the unsafe instruction. If they are different, it executes `int 3`. This raises an exception that invokes the debugger in debugging runs, or terminates execution in production runs. We could easily raise a different exception but this one is convenient for debugging. The instructions in the write check are encoded in 15 bytes when the table is at address 40000000h but they require only 11 bytes when the table is at address 0h.

We treat `free` as an unsafe instruction that writes to the object pointed to by its argument. The wrapper for `free` receives the color computed by the static analysis for the object being freed. Then it checks if the pointer argument points to an object with this color, if it is eight-byte aligned, if it points to user address space, and if the slot before this object has color one. If this check fails, we raise an exception. The first check prevents double frees because we reset the color of heap objects to zero when we free them. The last two checks prevent frees whose argument is a pointer to a non-heap object or a pointer into the middle of an allocated object. Recall that color one is reserved for heap guards and is never assigned to other memory locations.

### 4.5. Instrumenting indirect calls

We also add checks before each indirect call. These checks lookup the color of the target function in the color table and compare this color with the color of the indirect call instruction. If they do not match, we raise an exception. This instrumentation phase is also independent of the target architecture because it works with MIR. For example, we replace the indirect call `call t280` by the following MIR instruction sequence:

```
      t300 = SHIFTRIGHT t280, 3
      t301 = COMPARE(EQ) [t200+40000000h], 20
      CONDITIONALBRANCH(True) t301, $L10, $L11
$L11: BREAK
$L10: t302 = SHIFTLEFT t300, 3
      CALL t302                # indirect call
```

where `t300`, `t301`, and `t302` are fresh temporaries and 20 is the color of the indirect call instruction. Phoenix lowers this MIR instructions into the following sequence of x86 assembly code:

```
      shr  edx,3                          # 3 bytes
      cmp  byte ptr [edx+40000000h], 20 # 7 bytes
      je   $L10                           # 2 bytes
      int  3                              # 1 byte
$L10: shl  edx,3                          # 3 bytes
      call edx                     # indirect call
```

where register edx holds the function pointer and the color table starts at address 40000000h. The first instruction shifts the function pointer right by three to compute the color table index of the first instruction in the target function. The `cmp` instruction checks if the color in the table is the color of allowed targets for this indirect call instruction. If they are different, WIT raises an exception. If they are equal, the index is shifted left by three to restore the original function pointer value and the function is called.

This instruction sequence zeroes the three least significant bits of the function pointer value. Since the first instruction in a function is always 16-byte aligned, this has no effect if the function pointer value is correct. But it prevents attacks that cause a control flow transfer into the middle of the first eight-byte slot of an allowed target function. Therefore, this instruction sequence ensures that the indirect call transfers control to the first instruction of a call target that is allowed by the static analysis. The checks on indirect calls are sufficient to enforce control-flow integrity because all other control data is protected by the write checks.

## 5. Runtime

WIT has a small runtime that includes an initialization function and some wrappers for C runtime functions, for example, for `malloc` and `free`. The initialization function allocates the color table using `VirtualAlloc`, which reserves virtual memory for the table without adding space overhead for pages that are not accessed. The operating system zeroes the pages in the table when they are first accessed. The initialization function sets the color table entries for globals an their guard objects, and for the entry points of indirect call targets. We instrument the C runtime (`libc`) start-up function to invoke our initialization.

Since there are many memory errors due to incorrect use of `libc` functions, we use a version of `libc` instrumented with WIT. If we used the variant of WIT described in the previous sections, we would require a different `libc` binary for each program. Instead we developed a variant of WIT for libraries. This variant assigns the same well-known color (different from zero or one) to all unsafe objects allocated by the library and inserts guards around these objects. All safe objects used by the library functions have color zero. Before writes, this variant of WIT checks that the color of the location being written is greater than one, that is, that the location is not a safe object or a guard object. These checks prevent `libc` functions from violating control-flow integrity. They also prevent all commmon buffer overflows due to incorrect use of `libc` functions. However, they cannot prevent attacks that overwrite an unsafe object by exploiting format string vulnerabilities with the `%n` specifier, but these can be prevented with static analysis [38, 9] and are disallowed by some implementations.

We still need to write wrappers for `libc` functions that are written in assembly (for example, `memcpy` and `strcpy`) and for system calls (for example, `recv`). These wrappers receive the colors of destination buffers as extra arguments and scan the color table entries corresponding to the slots written by the wrapped function to ensure that they have the right color. Since the color table is very compact, these wrappers introduce little overhead. Other techniques require similar wrappers, e.g., [37, 21].

## 6. Effectiveness at preventing attacks

WIT can prevent all attacks that violate write integrity but the number of attacks that violate this property depends on the precision of the points-to analysis. For example if two objects have the same color, we may fail to detect attacks that use a pointer to one object to write to the other.

We ran experiments to evaluate the precision of the points-to analysis and its impact on security. We used WIT to compile nine programs from the SPEC CPU 2000 benchmark suite [40] (gzip, vpr, mcf, crafty, parser, gap, vortex, bzip2 and twolf). During compilation, we measured the number of colors used in each benchmark and the number of memory write instructions with each color. Then we ran the benchmarks and measured the maximum number of *objects* with each color at runtime, where an object is a local vari-

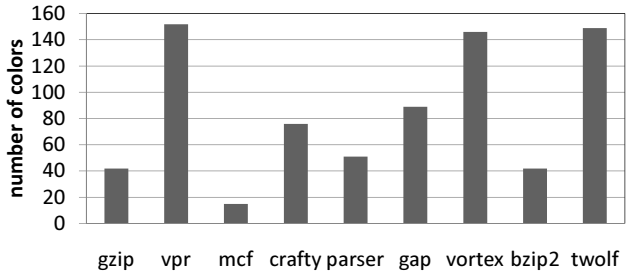


**Figure 4. Number of colors for SPEC benchmarks.**

able, a global variable, or an object allocated dynamically using `malloc`, `calloc`, `realloc`, or `alloca`. We combined these measurements to obtain an upper bound on the number of objects writable by each instruction at runtime. To compute this upper bound, we assumed a vulnerability that allows an unsafe instruction to write to any object with the same color as the instruction. We ignored constraints imposed by the program code and our guards.

Figure 4 shows the number of colors used by objects and functions in these benchmarks, and Figure 5 shows a cumulative distribution of the fraction of memory write instructions versus the upper bound on the number of objects writable by each instruction. For example, the first graph in Figure 5 shows that 88% of the memory write instructions in bzip can write at most one object at runtime, 99.5% can write at most two objects, and all instructions can write at most three objects. Therefore, even in this worst case, the attacker can only use a pointer to one object to write to another in 12% of the write instructions and in 96% of these instructions it can write to at most one other object. In practice, the program code and our guards will further reduce the sets of objects writable by each instruction.

The results in Figure 5 show that the precision of the points two analysis can vary significantly from one application to the other. For all applications except mcf and parser, the attacker cannot make the majority of instructions write to incorrect objects. For bzip, gap, crafty, and gzip, 93% of the instructions can write to at most one incorrect object in the worst case. The precision is worse for twolf, vpr and vortex because they allocate many objects dynamically. However, the fraction of instructions that can write a large number of objects is relatively small.

It is important to note that WIT can prevent many attacks regardless of the precision of the points-to analysis. Even when the analysis assigns the same color to all unsafe objects, our write checks can prevent: attacks that exploit sequential overflows and underflows, attacks that overwrite safe objects or code, and attacks that corrupt heap management data structures.

WIT prevents attacks that exploit buffer overflows and underflows by writing elements sequentially until an object boundary is crossed. These attacks are always prevented

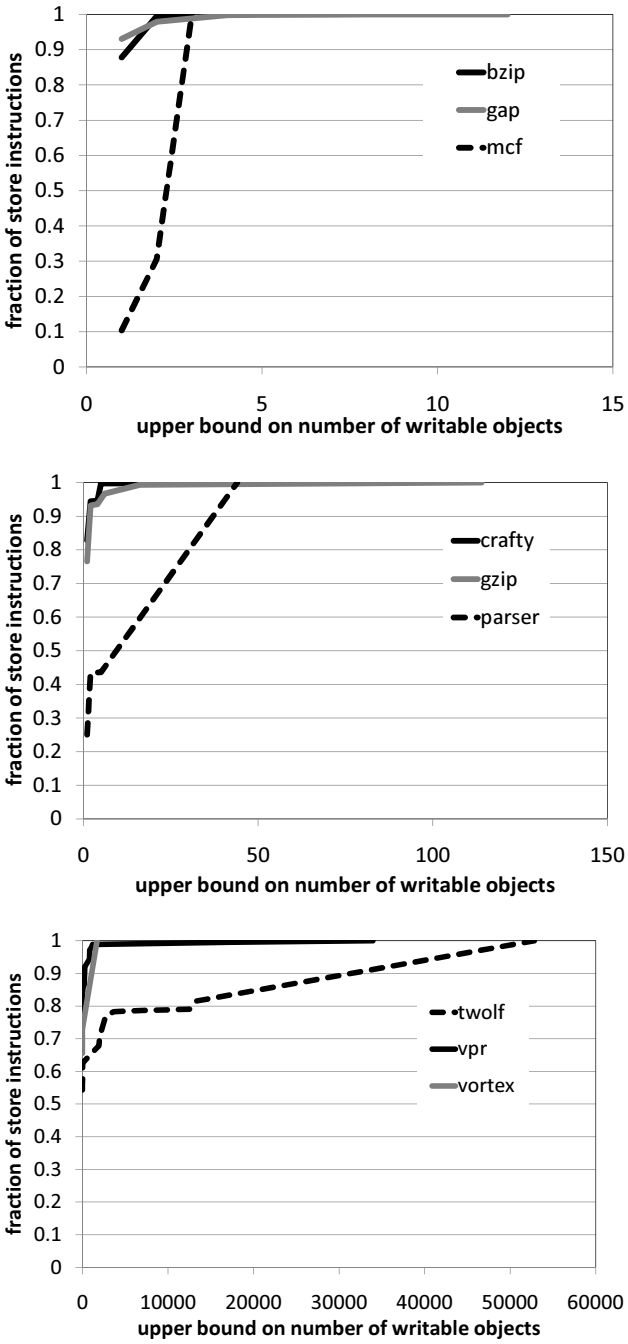


**Figure 5. Cumulative distribution of the fraction of store instructions versus the upper bound on the number of objects writable by each instruction.**

because the write checks fail when a guard is about to be overwritten. This type of attack is very common.

The write checks do not detect buffer overflows and underflows inside an object. For example, they will not detect an overflow of an array inside a C structure that overwrites a function pointer, a data pointer, or some security-critical

data in the same structure. In the first two cases, WIT can prevent the attacker from successfully exploiting this type of overflow because the indirect call checks severely restrict the targets of indirect calls and because the write checks may prevent writes using the corrupt data pointer. Overflows inside objects are not detected by any of the backwards-compatible C bounds checkers [25, 37, 21] and unlike WIT they have no additional checks to prevent successful exploits.

The write checks prevent all attacks that attempt to overwrite objects with color zero or code. Since objects have color zero by default, this includes many common types of attacks. For example, return addresses, saved base pointers, and exception handler pointers in the stack all have color zero. Other common attack targets like the import address table (IAT), which is used for dynamic linking, also have color zero. The write checks prevent the attacker from modifying code because the colors assigned to indirect call targets are different from the colors assigned to unsafe objects and the rest of the code has color zero.

WIT can prevent corruption of the heap management data structures used by the standard allocator in Windows without any changes to the allocator code. The checks on `free` prevent corruption due to incorrect use of `free`, and the write checks prevent corruption by unsafe aligned writes because the data structures have color one or zero. However, writes that are not aligned may overwrite the first few bytes of the heap metadata after an object. Misaligned writes generate exceptions in many architectures but they are allowed in the x86. We can prevent corruption in all cases by adding eight bytes of padding at the end of each heap object. In most applications, this adds little space and time overhead but it can add significant overhead in applications with many small allocations. This overhead may not be justified because most programs avoid misaligned writes for portability and performance, and the Windows allocator can detect corrupt heap meta-data when it tries to use it.

Control-flow integrity provides an effective second line of defense when the write checks fail to detect an attack. The number of attacks that violate control-flow integrity also depends on the precision of the points-to analysis. In the experiments described above, the maximum number of indirect call targets with the same color is 212 for gap, 38 for vortex and below 7 for all the other applications.

Even if the analysis assigned the same color to all indirect call targets, an attacker that corrupted a function pointer could only invoke one of these targets. Furthermore, these targets do not include functions in dynamically linked libraries that are invoked indirectly though the IAT. These library functions have color zero and we do not check these indirect calls because the IAT is protected by our write checks. Therefore, the attacker cannot use a corrupt function pointer to transfer control to library code, to injected code, or to other addresses in executable memory regions. This makes it hard to launch attacks that subvert the intended control flow, which are the most common.

WIT does not prevent out-of-bounds reads. These can lead to disclosure of confidential data but it is hard to exploit them to execute arbitrary code without violating write integrity or control-flow integrity in the process. We ran experiments using the same checks that we used for writes to prevent most out-of-bounds reads. Since the extra checks could increase overhead by more than a factor of three, we decided that the extra overhead did not justify the security improvement for most applications.

## 7. Experimental evaluation

We ran experiments to evaluate the overhead of our WIT implementation and its effectiveness at preventing a broad range of real and synthetic attacks. This section presents our results. WIT detects all the attacks in our tests and its CPU and memory overhead are low.

### 7.1. Overhead on CPU benchmarks

In our first experiment, we measured the overhead added by WIT to 9 programs from the SPEC CPU 2000 benchmark suite [40] (gzip, vpr, mcf, crafty, parser, gap, vortex, bzip2 and twolf) [3], and to 9 programs from the Olden [12] benchmark suite (bh, bisort, em3d, health, mst, perimeter, power, treeadd, and tsp). We chose these programs to facilitate comparison with other techniques that have been evaluated using the same benchmark suites.

We compared the running time and peak physical memory usage of the programs compiled using Phoenix [30] with and without WIT's instrumentation. We compiled the programs with options -O2 (maximize speed), -fp:fast (fast floating point model), and -GS- (no stack guards). When building WIT binaries, we linked with our runtime and with a WIT-instrumented version of libc (see Section 5). We ran the experiments on Windows Vista Enterprise, on an idle Dell Optilex 745 Workstation with a 2.46GHz Intel Core 2 processor and 3GB of memory. For each experiment, we present the average of 3 runs; the variance was negligible.

Figures 6 and 7 show the CPU overhead on SPEC and Olden benchmarks with WIT. For SPEC, the average overhead is 10% and the maximum is 25%. For Olden, the average overhead is 4% and the maximum is 13%. It is hard to do definitive comparisons with previous techniques because they use different compilers, operating systems and hardware, and they prevent different types of attacks. However, we can compare WIT's overhead with published overheads of other techniques on SPEC and Olden benchmarks.

[3] We did not run gcc, eon, and perlbmk because our prototype compiler cannot compile these benchmarks yet.

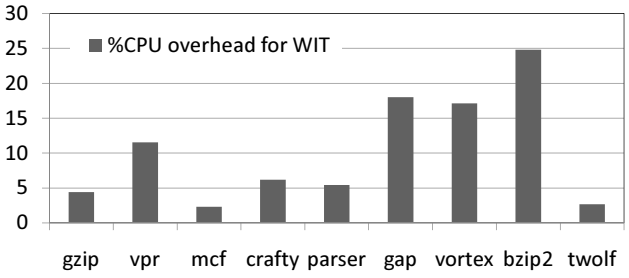


**Figure 6. CPU overhead on SPEC benchmarks.**

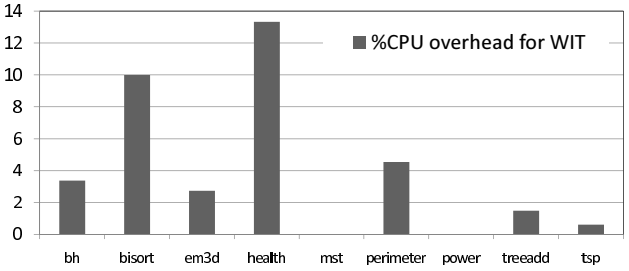


**Figure 7. CPU overhead on Olden benchmarks.**

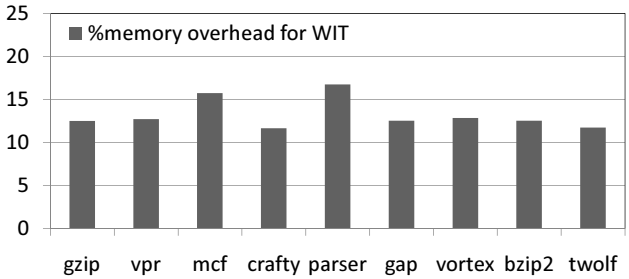


**Figure 8. Memory overhead on SPEC benchmarks.**

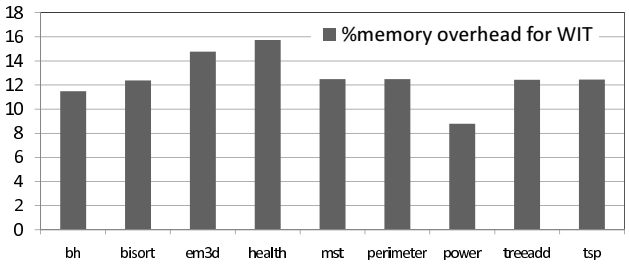


**Figure 9. Memory overhead on Olden benchmarks.**

For example, CCured [33] reports a maximum overhead of 87% and an average of 28% for Olden benchmarks, but it slows down some applications by more than a factor of 9. The bounds checking technique in [21] has an average overhead of 12% and a maximum overhead of 69% in the Olden benchmarks. WIT has three times lower overhead on average and the maximum is five times lower.

Figures 8 and 9 show WIT's memory overhead on SPEC and Olden benchmarks. The overhead is low for all benchmarks. For SPEC, the average memory overhead is 13% and the maximum is 17%. For Olden, the average is 13% and the maximum is 16%. This overhead is in line with our expectations: since WIT uses one byte in the color table for each 8 bytes of application data, the memory overhead is close to 12.5%. The overhead can decrease below 12.5% because we do not set color table entries for safe objects. On the other hand, the overhead can grow above 12.5% because we insert guard objects and pads between unsafe objects, but the results show that this overhead is small. It is interesting to compare this overhead with previous techniques even though they have different coverage. For example, CCured [33] reports an average memory overhead of 85% for Olden and a maximum of 161%. Xu et al. [45] report an average increase of memory usage by a factor of 4.31 for Olden benchmarks and 1.59 for SPEC benchmarks.

We also compiled the SPEC and Olden benchmarks with a version of WIT that adds 8 bytes of padding at the end of each heap object to protect heap metadata from hypothetical corruption by misaligned writes. The average time to complete the SPEC benchmarks increases from 10 to 11% and the average memory overhead increases from 13 to 15%. The average time to complete the Olden benchmarks increases from 4 to 7% and the average memory overhead increases from 13 to 63%. The overhead increases significantly in the Olden benchmarks because there are many small allocations. As we discussed earlier, we do not believe the increased security justifies the extra overhead.

## 7.2. Overhead on a Web server

The benchmarks used in the previous sections are CPU-intensive. They spend most time executing instrumented code at user level. The overhead of our instrumentation is likely to be higher in these benchmarks than in other programs where it would be masked by other overheads. Therefore, we also measured the overhead added by our instrumentation to the `NullHttpd` Web server running the SPEC WEB 1999 [40] benchmark.

The server ran on a Dell Optilex 745 Workstation with an Intel Core 2 CPU at 2.4GHz and 2GB of RAM, running Windows Vista Enterprise. We simulated clients using a Dell Workstation running Windows XP SP2. The machines were connected by a 100Mbps D-Link Ethernet switch. We configured the clients to request only a static 100-byte file from the SPEC Web benchmark. We could easily drive our overhead to zero by requesting large files, reading them from disk, or creating processes to generate dynamic content. But we chose this setting to measure worst case overhead for Web server performance with WIT instrumentation. We measured the average response time and throughput with and without instrumentation and we increased the number of clients until the server reached peak throughput.

The results are the average of three runs.

When load is low, WIT's overhead is masked by the time to send requests and replies across the network. The average operation response time in an unloaded server (1 client) is only 0.2% longer with instrumentation than without. When load is high and the server is saturated, WIT's overhead increases because the server is CPU-bound in this benchmark. The overhead increases up to a maximum of 4.8%, which shows that WIT can be used in production Web servers.

### 7.3. Synthetic exploits

We ran the benchmark described in [43] that has 18 control-data attacks that exploit buffer overflow vulnerabilities. The attacks are classified according to the technique they use to overwrite control-data, the location of the buffer they overflow, and the control-data they target. There are two techniques to overwrite control-data. The first overflows a buffer until the control-data is overwritten. The second overflows a buffer until a pointer is overwritten, and uses an assignment through the pointer to overwrite the control-data. The attacks can overflow buffers located in the stack or in the data segment, and they can target four types of control-data: the return address on the stack, the old base pointer on the stack, and function pointers and longjmp buffers in the stack or in the data segment.

WIT can prevent all the attacks in the benchmark. All the attacks except one are detected when a guard object is about to be overwritten. The remaining attack is not prevented by the guard objects because it overflows a buffer inside a structure to overwrite a pointer in the same structure. This attack is detected when the corrupted pointer is used to overwrite a return address because the return address has color zero.

These attacks can be prevented by other techniques, for example, [37, 28, 6, 13], but these techniques are not widely used because they have high overhead. StackGuard [20] is widely deployed because it has low overhead but it does not prevent attacks that overflow non-stack buffers.

### 7.4. Real vulnerabilities

In our final experiment, we tested WIT's ability to prevent attacks that exploit real vulnerabilities in `SQL server`, `Ghttpd`, `Nullhttpd`, `Stunnel`, and `libpng`.

`SQL server` is a relational database from Microsoft that was infected by the infamous Slammer [32] worm. The vulnerability exploited by Slammer causes `sprintf` to overflow a stack buffer. We used WIT to compile the SQL server library with the vulnerability. WIT detects Slammer when the `sprintf` function tries to write over the guard object inserted after the vulnerable buffer.

`Ghttpd` is an HTTP server with several vulnerabilities [1]. The vulnerability that we chose is a stack buffer overflow when logging `GET` requests inside a call to `vsprintf`. WIT detects attacks that exploit this vulnerability when `vsprintf` tries to write over the guard object at the end of the buffer.

`Nullhttpd` is another HTTP server. This server has a heap overflow vulnerability that can be exploited by sending HTTP `POST` requests with a negative content length field [2]. These requests cause the server to allocate a heap buffer that is too small to hold the data in the request. While calling `recv` to read the `POST` data into the buffer, the server overwrites the heap management data structures maintained by the C library. This vulnerability can be exploited to overwrite arbitrary words in memory. We attacked `NullHttpd` using the technique described in [15]. The attack works by corrupting the CGI-BIN configuration string. This string identifies a directory holding programs that may be executed while processing HTTP requests. Therefore, by corrupting it, the attacker can force `NullHttpd` to run arbitrary programs. This is a non-control-data attack because the attacker does not subvert the intended control-flow in the server. WIT detects the attack when the wrapper for the `recv` call is about to write to the guard object at the end of the buffer. The example in Section 2 is inspired by this.

`Stunnel` is a generic tunnelling service that encrypts TCP connections using SSL. We studied a format string vulnerability in the code that establishes a tunnel for SMTP [4]. An attacker can overflow a stack buffer by sending a message that is passed as a format string to the `vsprintf` function. WIT detects the attack when `vsprintf` attempts to write the guard object at the end of the buffer.

`Libpng` is a library for processing images in the PNG file format [3]. Many applications use `Libpng` to display images. We built a test application distributed with `Libpng` and attacked it using the vulnerability described in [31]. The attacker can supply a malformed image file that causes the application to overflow a stack buffer. WIT detects the attack when a guard object is about to be written.

## 8. Related work

Type safe languages like Java and C# eliminate memory errors. However, there is a large amount of software written in unsafe languages like C and C++, and these languages are still widely used to develop new software. So memory errors will remain a problem in the foreseeable future. There is a large body of work on techniques to protect C and C++ programs from attacks that exploit memory errors.

Some techniques use static analysis to identify vulnerabilities, for example, [42, 29, 14]. They have the advantage of removing vulnerabilities from software before it ships and they do not introduce any runtime overhead. How-

ever, static analysis techniques are not sufficient because they are imprecise: they can miss vulnerabilities and raise false alarms. Too many false alarms are an issue because they may cause developers to stop using the tools.

Many techniques to prevent attacks that exploit memory errors are not widely used. We believe there are two reasons for this: requiring non-trivial changes to the source code or the language runtime, or incurring high overhead. For example, CCured [33] and Cyclone [24] proposed memory safe dialects of C. They can prevent all memory errors but require a significant effort to port C applications to the safe dialects, and require major changes to the runtime. For example, they replace `free` by a garbage collector. Their runtime overhead is also significantly higher than WIT's.

Other techniques can be applied to C and C++ programs without modifications. Several systems detect attacks using dynamic taint analysis, e.g., [34, 17], which can prevent many attacks that exploit memory errors and other types of attacks. They work with binaries and do not require source code. However, their overhead is several orders of magnitude higher than WIT's. Xu *et al* [44] describe a dynamic taint analysis technique that is implemented as a source-to-source transformation on C programs. Their overheads are an order of magnitude lower than previous techniques but they are still above 100% when preventing memory error exploits on CPU-intensive benchmarks.

There are several bounds checkers for C. For example, the Jones and Kelly [25] bounds checker does not require changes to the pointer format. It instruments pointer arithmetic to ensure that the result and original pointers point to the same object. To find the target object of a pointer, it uses a splay tree that keeps track of the base address and size of heap, stack, and global objects. CRED [37] is similar but provides support for some common uses of out-of-bounds pointers in existing C programs. These techniques have high overhead, for example, CRED can slow down applications by up to a factor of 12. Xu *et al* [45] describe a technique that improves the coverage of the previous bounds checkers and reduces their overhead. Their average overhead when preventing only spatial errors in the Olden benchmarks is 12 times larger than WIT's.

Some techniques to prevent memory error exploits defend from attacks that overwrite specific targets, such as return addresses, pointers, or other control data (e.g., [20, 19, 39]), or that exploit specific vulnerabilities, such as format string vulnerabilities (e.g., [18]). These techniques have low overhead but they cannot defend from other attacks. Techniques inspired by StackGuard [20] are widely used. However, there are memory error exploits that they cannot catch [43]. For example, they provide no protection from overflows of heap and static variables [35]. WIT has similar performance and broader coverage than these techniques.

The concept of control-flow integrity generalizes the work of Wagner and Dean [41] and was introduced in [28, 6]. However, attackers can exploit memory errors to execute arbitrary code without violating control-flow integrity. There are examples of several attacks of this type in [15]. CFI [6] and Program Shepherding [28] cannot detect this type of attack. WIT can and it also detects all attacks that violate control-flow integrity. Additionally, WIT has lower overhead because it avoids control-flow integrity checks on returns. For example, CFI has an average overhead of 15% and a maximum overhead of 45% on the SPEC benchmarks where we overlap. The average overhead grows to 24% with the version of CFI that uses a shadow call stack to ensure that functions return to their caller. WIT has an average overhead of 10% and a maximum of 25% and it also ensures functions return to their caller.

DFI [13] combines static points-to analysis with runtime instrumentation like WIT. For each instruction that reads a value, it uses static analysis to compute the instructions that are allowed to write the value. Then it instruments writes and reads to ensure that the values read at runtime were written by allowed instructions. Its coverage is similar to WIT's. It can detect some out-of-bounds reads and reads after free but it does not have guards to improve coverage when the analysis is imprecise. DFI's average overhead on the SPEC benchmarks where we ovelap is 104%.

The technique described by Yong *et al.* [46] has some similarities with WIT. It assigns colors to objects and checks a color table on writes. However, it has worse coverage than WIT because it uses only two colors, it does not insert guards, and it does not enforce control-flow integrity. The two colors distinguish between objects that can be written by an unsafe pointer and those that cannot. Yong *et al.* incur an average overhead ten times larger than WIT on the SPEC benchmarks where we overlap.

In concurrent work, Clause *et al* [16] describe a technique that assigns colors to objects and pointers dynamically. It assigns a random color to memory objects when they are allocated and, when a pointer to an object is created, it assigns the color of the object to the pointer. Then it propagates pointer colors on assignment and arithmetic. On reads and writes to memory, it checks if the color of the pointer and the memory match. Their software-only version slows down SPEC INT by a factor of 100 or more. With special hardware and 256 colors, their average overhead on SPEC INT is 7%. WIT has similar overhead without special hardware support.

There are several techniques that insert guard zones or pages around objects, for example, [36, 22]. WIT achieves better coverage by combining guard objects with runtime enforcement of write and control-flow integrity.

Some techniques randomize the layout of objects in memory to make it harder for attackers to exploit memory errors. The most comprehensive randomization technique

that we know [11] has an overhead of 17% on gzip and WIT has an overhead of 7%. DieHard [10] randomizes the location of objects in the heap and the order in which objects are reused after being freed. The geometric mean of their overhead on SPEC INT is 12% but their maximum overhead is 109% (on twolf). WIT's overhead on twolf is 3%. Other techniques do instruction set randomization [27] but have high overhead without hardware support.

The fastest software-only technique with coverage similar to WIT was presented by Dhurjati *et al* [21]. WIT can prevent attacks that cannot be prevented by this technique. For example, Dhurjati *et al* cannot prevent attacks that overflow a buffer inside a structure to overwrite a pointer field in the same structure, or attacks that exploit dereferences of pointers in objects that are freed and reused [7]. These attacks can use corrupt pointers to write to arbitrary memory locations or to invoke arbitrary code. WIT's write and indirect call checks severely restrict the use of these corrupt pointers. On the other hand, Dhurjati *et al* can prevent most out-of-bounds reads and writes. WIT does not prevent out-of-bounds reads and it may fail to prevent some out-of-bounds writes as discussed earlier.

The technique of Dhurjati *et al* is similar to CRED but introduces several optimizations that reduce runtime overhead dramatically. For example, it uses points-to analysis to partition objects into pools and uses a splay tree for each pool. These splay trees can be looked up more efficiently than the large splay tree used by previous approaches and each pool has a cache for even faster lookups. This technique has an average overhead of 12% and a maximum overhead of 69% in the Olden benchmarks. WIT's average overhead on the same benchmarks is 4% and the maximum overhead is 13%. WIT is three times faster than the fastest technique with similar coverage.

## 9. Conclusion

We presented WIT, a new technique to prevent memory error exploits. WIT uses a combination of static analysis and runtime instrumentation to enforce two safety properties: write integrity ensures that instructions do not write to unintended storage locations, and control-flow integrity ensures that control is not transferred to unintended targets. WIT can prevent all memory error exploits that enable arbitrary code execution that we know about. We have an efficient implementation of WIT: the average space overhead in our benchmarks is 13% and the average runtime overhead is 7%. We believe this overhead is low enough for WIT to be deployed widely.

## Acknowledgments

We would like to thank the anonymous reviewers and the following people for discussions and comments on the work described in this paper: Martin Abadi, Andy Ayers, Paul Barham, Richard Black, Tim Burrel, Chris Hawblitzel, Nitin Kumar Goel, Joshua Goodman, Steve Kruy, Ronald Laeremans, Scott Lambert, Butler Lampson, John Manferdelli, Jean-Philippe Martin, Chris McKinsey, Matt Miller, Matt Moore, Marcus Peinado, Herb Sutter, Damien Watkins, Chris Walker, James Whittaker, and Ben Zorn.

## References

[1] GHttpd Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960.

[2] Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774.

[3] Portable Network Graphics (PNG) Specification and Extensions. http://www.libpng.org/pub/png/spec/.

[4] STunnel Client Negotiation Protocol Format String Vulnerability. http://www.securityfocus.com/bid/3748.

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

[6] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM Conference on Computer and Communications Security* (Nov. 2005).

[7] AFEK, J., AND SHARABANI, A. Dangling Pointer: Smashing the pointer for fun and profit. Watchfire white paper, Aug. 2007.

[8] ANDERSEN, L. Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen, 1994.

[9] AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. Improving software security with a C pointer analysis. In *ACM/IEEE International Conference on Software Engineering* (May 2005).

[10] BERGER, E., AND ZORN, B. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM Conference on Programming Language Design and Implementation* (June 2006).

[11] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium* (July 2005).

[12] CARLISLE, M. C. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, 1996.

[13] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation* (Nov. 2006).

[14] CHEN, K., AND WAGNER, D. Large-scale analysis of format string vulnerabilities in Debian Linux. In *Workshop on Programming Languages and Analysis for Security* (June 2007).

[15] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *USENIX Security Symposium* (July 2005).

[16] CLAUSE, J., DOUDALIS, I., ORSO, A., AND PRVULOVIC, M. Effective Memory Protection Using Dynamic Tainting. In *International Conference on Automated Software Engineering* (Nov. 2007).

[17] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *Symposium on Operating System Principles* (Oct. 2005).

[18] COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. FormatGuard: automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium* (Aug. 2001).

[19] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium* (Aug. 2003).

[20] COWAN, C., PU, C., MAIER, D., HINTON, H., WADPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *USENIX Security Symposium* (Jan. 1998).

[21] DHURJATI, D., AND ADVE, V. Backwards-compatible array bounds checking for C with very low overhead. In *ACM/IEEE International Conference on Software Engineering* (May 2006).

[22] DHURJATI, D., AND ADVE, V. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *International Conference on Dependable Systems and Networks* (June 2006).

[23] HEINTZE, N., AND TARDIEU, O. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *ACM Conference on Programming Language Design and Implementation* (June 2001).

[24] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference* (June 2002).

[25] JONES, R., AND KELLY, P. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Workshop on Automated Debugging* (May 1997).

[26] JP. Advanced Doug Lea's malloc exploits. *Phrack*, 61 (Sep. 2003).

[27] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *ACM CCS* (Oct. 2003).

[28] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *USENIX Security Symposium* (Aug. 2002).

[29] LAROCHELLE, D., AND EVANS, D. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium* (Aug. 2001).

[30] MICROSOFT. Phoenix compiler framework. `http://research.microsoft.com/phoenix/phoenixrdk.aspx`.

[31] MITRE CORPORATION. Multiple buffer overflows in libpng 1.2.5. CVE-2004-0597, June 2004.

[32] MOORE, D., PAXSON, V., SAVAGE, S., SHANNON, C., STANIFORD, S., AND WEAVER, N. Inside the Slammer worm. *IEEE Security and Privacy 1*, 4 (July 2003).

[33] NECULA, G., CONDIT, J., HARREN, M., MCPEAK, S., AND WEIMER, W. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems 27*, 3 (May 2005).

[34] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *NDSS* (Feb. 2005).

[35] PINCUS, J., AND BAKER, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2*, 4 (2004), 20–27.

[36] ROBERTSON, W., KRUEGEL, C., MUTZ, D., AND VALEUR, F. Run-time Detection of Heap-based Overflows. In *USENIX conference on System administration* (2003).

[37] RUWASE, O., AND LAM, M. A practical dynamic buffer overflow detector. In *NDSS* (Feb. 2004).

[38] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium* (Aug. 2001).

[39] SMIRNOV, A., AND CHIUEH, T. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *NDSS* (Feb. 2005).

[40] SPEC. SPEC Benchmarks. `http://www.spec.org`.

[41] WAGNER, D., AND DEAN, D. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy* (May 2001).

[42] WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS* (2000).

[43] WILANDER, J., AND KAMKAR, M. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS* (Feb. 2003).

[44] XU, W., BHATKAR, S., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security* (2006).

[45] XU, W., DUVARNEY, D. C., AND SEKAR, R. An efficient and backwards-compatible transformation to ensure memory safety of c programs. *SIGSOFT Softw. Eng. Notes 29*, 6 (2004), 117–126.

[46] YONG, S. H., AND HORWITZ, S. Protecting C programs from attacks via invalid pointer dereferences. In *European Software Engineering Conference* (2003).

[47] YONG, S. H., AND HORWITZ, S. Pointer-Range Analysis. In *Static Analysis Symposium* (2004).