Bounds Checking with Taint-Based Analysis

Weihaw Chuang† Satish Narayanasamy† Brad Calder†‡
Ranjit Jhala†

†CSE Department, University of California, San Diego
‡Microsoft
{wchuang,satish,calder,jhala}@cs.ucsd.edu

Abstract

We analyze the performance of different bounds checking implementations. Specifically, we examine using the x86 `bound` *instruction to reduce the run-time overhead. We also propose a compiler optimization that prunes the bounds checks that are not necessary to guarantee security. The optimization is based on the observation that buffer overflow attacks are launched through external inputs. Therefore, it is sufficient to bounds check only the accesses to those data structures that can possibly hold the external inputs. Also, it is sufficient to bounds check only the memory writes. The proposed optimizations reduce the number of required bounds checks as well as the amount of meta-data that need to be maintained to perform those checks.*

1 Introduction

Bounds checking is the process of keeping track of the address boundaries for objects, buffers, and arrays, and checking the loads and stores that access those structures to make sure that they do not stray outside of the bounds. The bounds are typically represented by a lower and upper address, in between which the loads and stores can validly access. The bounds check consists of checking a memory access to make sure the address is within these bounds. If a violation occurs, the check may issue an exception [1–3], or circumvent the error [4] in a safe fashion.

Buffer overflow attacks can be prevented using bounds checking. Consider a fixed size buffer allocated on a stack. A buffer overflow can occur when the application copies external data into the buffer but does not check for the size of the input data, especially while copying strings (e.g. using strcpy). An adversary can exploit this by copying malicious data or even a program into the buffer. When the buffer is located on the stack, the adversary can overwrite the return address as illustrated in Figure 1, and the control will jump to execute the malicious code upon return from the function. This form of buffer overflow attack is called "Stack Smashing" [5]. Stack smashing has been used in Code Red, Nimda and Slammer, just to name a few Internet worms.

The main goal of our research is to provide efficient bounds checking for pointer based applications such as C and C++, as programs written in these languages are error prone and are easily exploited by adversaries. If the bounds checking overhead is reasonable, then it can be used in the production software to make it secure.

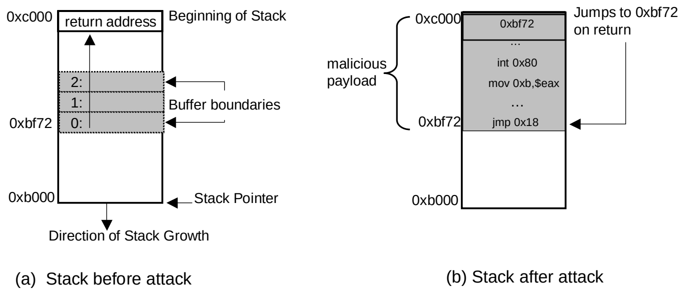

Fig. 1. Stack Smashing example

We first examine the efficiency of various possible bounds checking implementations for C programs. Specifically, we examine the advantages of using the x86 `bound` instruction, which has largely been ignored in the previous studies, and compare its performance with a two-branch [2] and a single-branch [6] bounds check implementations. The bounds instruction reduces the bounds check performance overhead. Also, as it reduces the dynamic instruction count we can obtain significant energy savings.

We focus on static compiler optimizations to provide efficient bounds checking for improving security. The proposed algorithms are based on recent works on preventing buffer overflow attacks. Suh et al. [7], and Crandall and Chong [8] examined using hardware support to tag each memory word with a *taint* bit indicating if the data put into memory was stored there from untrusted sources like the network. Then if any memory address is executed with this taint bit set, a buffer overflow attack is flagged. Assuming this hardware support, they were able to provide this protection with only a few percent slowdown. More recently Newsome and Song [9] implemented this approach using dynamic binary emulation to track data from external sources. They were able to do this without any hardware support, but with a slowdown of 5 times over native execution. Our goal is to achieve the same level of security using bounds checking, but with minimal additional hardware support and performance overhead.

We discuss static compiler algorithms to reduce the number of bounds checks required to guard against buffer overflow attacks. To protect against buffer overflow attacks, not all the memory accesses need to be bounds checked. If we know the memory locations that are prone to buffer overflow attacks, then only the accesses to those locations have to be bounds checked and the rest can be safely pruned away. An adversary can launch a buffer overflow attack only by providing malicious data to the program. Hence, in our analysis, we assume that any memory location that can potentially hold data received from sources external to the program to be vulnerable to buffer overflow attacks. Statically, we can determine such locations by looking for places where the program *interfaces* with the external world through external library calls such as system calls. Any memory location that can receive external data directly from the external sources is considered to be *tainted*. Also, those memory locations that get values from a tainted location is considered to be tainted as well (which can be determined through a simple data-flow analysis). Only the pointer de-references to tainted locations are bounds checked. We call this optimization as the *interface* optimization.

We improve the interface optimization by further limiting the bounds checks to only memory writes, because a worm has to write the malicious data to memory to launch an attack. Also, we can limit the bounds checks to only those memory locations that can hold the data received from the network. All these reductions in the number of bounds checks will also results in reductions in the amount of meta-data that needs to be maintained to perform those checks.

2 Methodology

In this section we summarize our compiler that was derived from McGary's GCC patch [2]. We also describe our experimental methodology and the benchmarks used.

2.1 Compiler

Our compiler is based on Greg McGary's bounds checker [2]. His project contains a patch to the 2.96 GCC sources where the patched compiler generates fat-pointers and compare-branch bounds checks discussed in Section 3. We made several general modifications, as we wanted to experiment with different bounds check implementations, described in Figure 2. To interface with non-bounds checked code, we wrote bounds checking wrappers for many library functions. To eliminate simple bounds check redundancy we modified the GCC value numbering optimization to recognize the bounds instruction and eliminated useless checks. We also implemented simple loop hoisting of bounds checks. Both correspond to an acyclic and cyclic bounds optimization mentioned in Markstein et.al. [10]. Lastly, we modified the compiler to let us do inter-procedural analysis of type information as described in Section 4.

2.2 Benchmarks

Our goal is to reduce the overhead of bounds checking while maintaining the security coverage that bounds checking provides. To measure the performance overhead, we used the SPEC 2000 Integer benchmarks. We provide results for all seven of the C SPEC 2000 Integer programs that compile and run correctly with our baseline McGary `gcc` compiler. These are compiled on GCC with the -O3 option. The remaining four (`gcc`, `perl`, `gap` and `vortex`) failed to compile on the baseline McGary compiler.

2.3 Measurements

Our performance measurements are based on using the hardware counters on commercially available processors. For each result, we executed the program three times to factor out random effects while executing on a real processor. Our results are based on the AMD Athlon 2400+ XP (K7), including all of our hardware performance counter numbers.

We did a micro-architectural analysis of the overhead to get a deeper understanding of bounds checking, using Petterson's hardware performance Linux kernel patch [11]. These results include branch misprediction, L1 data cache miss, memory instruction count, and total instruction count. Pettersson's tool provides application level access to these low level hardware performance counters and handles operating system details such as saving and restoring state during context switch.

3 Efficient Bounds Checking for C

In this section we describe three standard representations for the bounds meta-data and the baseline implementations we consider for bounds checking.

3.1 Representing Bounds Information

A bounds checker needs bounds information to perform its verification. We will now describe how this bounds meta-data information is organized and used when a pointer is dereferenced.

Bounds information for static objects is determined completely by the compiler, while bounds information for dynamic objects is determined at run-time. The bounds for an object are created using its size (which can be deduced from the object's type), and the memory location of the object. The bounds are typically represented using a *low bound* and a *high bound*. Alternatively, they may contain the low bound and the object's size.

There are three ways of storing the bounds meta-data to perform bounds checking. They are (a) fat-pointers, (b) a meta-data table, and (c) adjacent to the object (referenced by pointer). A fat-pointer [2, 12, 3, 6] contains the object's low and high bounds adjacent to the pointer. Thus, it changes the pointer's format. This form of bounds representation requires no additional code to locate the meta-data (bounds information), and so it is fast. A second representation of the bounds information is the table lookup approach [13, 1]. The bounds meta-data table is indexed using the pointer value to retrieve its bounds. This representation does not require changes to the memory layout of the data objects but lookup incurs significant run-time overhead. Another method for tracking bounds meta-data is to store them with the object. The meta-data for a pointer can be located by adding a fixed offset to the base address of the pointer and so the look-up is efficient. However, this representation may not be reliable with C pointers which can potentially get modified. Also, it doesn't work when the base address for the pointer is not available. Hence, in this paper we use fat-pointers as our baseline, and in some experiments use object meta-data to improve array bounds checking code (where we do not have the issues that we mentioned).

3.2 Check Implementation

We now describe the three main implementations for doing a bounds check. The bounds checks are annotated into the program at the pointer dereference operator and at the array subscript operator.

A bounds check can be done by keeping track of lower and upper bounds for the object and comparing them against the effective address used while dereferencing a pointer as shown in Figure 2(a). McGary's compiler, which we build on, uses this method [2]. This implementation requires the execution of at least four CISC instructions including two branches. There is an alternative implementation [6] that requires the execution of just a single branch as shown in Figure 2(b), using low bound and size, resulting in at least three instructions. A third possible implementation for bounds checking is to use a dedicated `bound` check instruction that exists in the x86 instruction set as shown in Figure 2(c). The `bound` instruction is functionally identical to implementation (a) but eliminates the two additional branches, the extra compares and other arithmetic instructions in the binary.

```
start:                       start:                        start:
   flag=(ptr >= low)           tmp=(unsigned)(ptr-low)       bound ptr, b_ptr
   if(flag) then low_pass      flag=(tmp < size)
   trap                        if(flag) then ok
low_pass:                      trap
   flag=(ptr < high)         ok:
   if(flag) then high_pass
   trap
high_pass:
(a) Two Branch               (b) Single branch             (c) x86 bounds
```

Fig. 2. Pseudo-Code for three possible implementations of Bounds Check. (a) Two Branch (b) Single Branch (c) Bound Instruction

Prior work on bounds checking has not performed a direct comparison of the different bounds checking implementations. In addition, the prior techniques have not explored the use of the x86 `bound` instruction for their implementations. In this section we provide performance results comparing the three implementations of bounds checking. Before that, we first present a code generation optimization for using `bound` instruction for checking array references.

3.3 Code-Generation of x86 Bound Instruction for Arrays

The x86 bounds instruction has only two operands as shown in Figure 2(c). The first input (`ptr`) specifies a general purpose register containing the effective address that needs to be bounds checked, and the second input (`b_ptr`) is a memory operand that refers to a location in memory containing the low and high bounds. Limiting this second operand to be a memory location requires that the bounds check needs to be located in memory and cannot be specified as a constant or a register value. On execution, the bounds instruction checks if the address specified by the first operand is between the low and high bounds. If the bounds are exceeded, a bounds-checking exception is issued.

For the approach we focus on, we assume the use of fat pointers. Since we use fat pointers, the bounds information for pointers will be stored in memory adjacent to the actual pointer. We would like to use the bound instruction for all bounds checks, but an issue arises when we try to do bounds checking for global and stack arrays, as they are referenced in C and C++ without pointers. To allow the `bound` instruction to be used for these references, we allocate memory adjacent to statically declared arrays to hold the meta-data bounds information, which will be created and initialized when the arrays are created. Since the bounds information for the global and local arrays are now located adjacent to the object data memory, the `bound` instruction can take in the corresponding memory location as its second operand to do bounds checking for these arrays. This memory location is located at a fixed offset from the base of the array. Out of the techniques we examined, we found this method to provide the best performance for bounds checking array references, and we call this configuration *bnd-array*.

3.4 Analysis

We now examine the performance of the three implementations of bounds checking. The two-branch bounds check (*GM*) implementation is used by the baseline McGary compiler [2]. It has a compare-branch-compare-branch sequence.

The single branch (*1BR*) uses one (unsigned) subtract-compare-branch-trap sequence [6]. We compare these checks to our *bnd-array* results that uses *bounds* instruction bounds check with the array meta-data. Reported numbers are overheads in comparison to the baseline without any bounds checking. All three code combinations are shown in Figure 2. We would also like to understand to what extent one can reduce the bounds checking overhead, if one implements the `bound` instruction as efficiently as possible in a processor. To do this, we generated a binary similar to *bnd-array* but removed all the `bound` instructions, only leaving behind the bounds meta-data in the program and the instructions to maintain them. We call this configuration as *bnd-ideal*.

Figure 3 shows the percent slowdown for running the programs with the different bounds implementations on an AMD Athlon, when compared to an execution without any bounds checking. The first result of our comparison is that bounds checking using McGary's (GM) compiler results in overheads of 71% on average. The second result is that using the single branch compare implementation *1BR* reduces the average performance overhead from 72% down to 48% on the Athlon. The third result is that the x86 bound instruction overhead (*bnd-array*) provides the lowest bounds checking overhead when compared to these two techniques with an average slowdown of 40% on Athlon. Finally, performance overhead for *bnd-ideal* is 30% which is only 10% less than the *bnd-array* configuration. In other words, 30% slowdown is due to just maintaining the meta-data.

One important component of overhead is due to increased dynamic instruction count as seen in Figure 4. As expected there is significant instruction count overhead for *GM* (149%), which is reduced to 121% for *1BR*, and roughly half of that (65% overhead) for *bnd-array*. Of the 65% instruction count overhead in *bnd-array*, only 14% are due to *bounds* instructions and the rest 51% are due to support instructions required to maintain the bounds meta-data (this result is deduced from *bnd-ideal* result which shows just the overhead of maintaining the bounds meta-data).

We also examine the branch misprediction and data cache hardware counters to determine the reasons for the *bounds* performance improvements (graphs not shown due to space limitations). The two-branch *GM* (256%) has much higher branch misprediction rate than the *bnd-array* (11.5%) and the one branch version *1BR* (105%), explaining some of the performance advantages of using *bound* instructions over branched versions. Data cache misses increase in the bounds checking implementations as the bounds information required to perform the checks increases the memory footprint. This overhead is fairly constant across all the implementations as they all keep track of same amount of meta-data.

3.5 Issues to Address for Bounds Checking in C

Because bounds checking requires bounds meta-data to be maintained and associated with pointers, there are a few issues to address. Many of these problems are well known and have solutions, which we cite and use in our implementations.

The type-cast operator in C, might cast a pointer to integer and back to a pointer against, which could strip the bounds meta-data when represented as

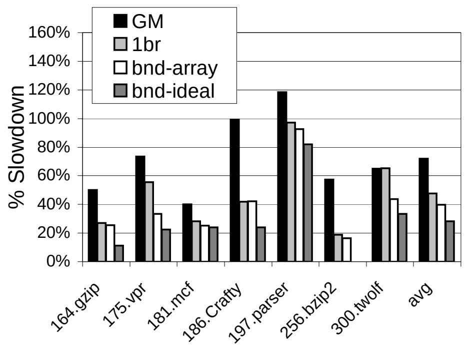

Fig. 3. Run-time overhead of bounds checking (AMD Athlon).

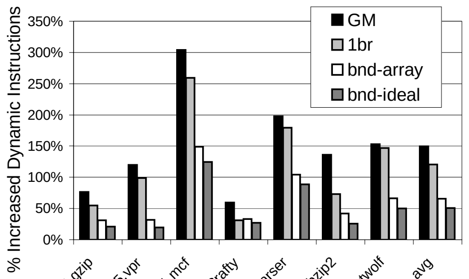

Fig. 4. Increased dynamic instructions due to bounds checking

the integer. One proposed solution, which we use in this paper, is to wrap the integer with bounds meta-data just like how a regular pointer is wrapped [14]. For the programs we compiled for, we found that we had to do this form of type cast infrequently, affecting few variables, and resulting virtually no code bloat. In terms of pointer-to-pointer casts, these are already handled by default because the bounds are transferred to the destination. Even if the pointers have different type representation, the bounds remain valid as the allocated low and high bounds are unchanged.

Linking libraries with application code containing fat-pointers needs to be addressed. The conservative solution is to mandate that library interfaces are fat-pointers in the case where library calls need bounds information. This means that libraries also need to perform bounds checking. McGary's distribution provided a fully bounds checked with fat-pointers patch of libc. Another approach is to use wrappers to interface with non-bounds checked library code, that converts the pointers and performs any safety checks on passed data, as was done by CCured [14]. We use this approach in our simulations, where we wrap each library call by copying the pointer value of a fat-pointer to a normal (skinny) pointer and pass that to the system call. Then on return from the system call, we copy the value of the normal pointer back to the fat-pointer, with the appropriate meta-data, and return. We verified there was no loss in performance or security (for the user code). These interface wrappers can be generated automatically [14].

Buffer overflow can cause denial-of-service when the victim program crashes. Recent work using CRED demonstrates that bounds checkers can continue running even after bounds violations are detected, without any loss of protection [4]. These failure tolerant CRED programs have the same overhead as the original bounds checked versions.

4 Taint-Based Bounds Checking

Bounds checking verifies all pointer and array accesses. While this provides complete protection against all kinds of buffer overflow attacks, it has significant performance overhead. In this section, we present a technique to limit the scope of bounds checking to only those objects that are vulnerable to a buffer overflow attack. The goal of this optimization is to filter away those bounds checks that

```
1. char* gets(char* s);
2. int main(int argc, char** argv)
3. {
4.   char tainted_args[128], tainted_gets[128], indirect[64];
5.   char *tainted_alias = tainted_gets; // Pointer alias to object
6.   strcpy(tainted_args, argv[1]);  // source of malicious input: argv
7.   gets(tainted_alias); // source of malicious input: gets()
8.   for(i=0; (tainted_alias[i] != '\0') ; i++)
     {  // indirect is TAINTED because it 'uses' tainted_alias
9       indirect[i] = tainted_alias[i];
     }
10.  char safe_array[128] = "I am not tainted"; // SAFE and THIN
     // foobar never passes safe_array to external
     // interfaces nor assigns it tainted data
11.  foobar(safe_array);
     ...
}
```

Fig. 5. Accesses through the pointers and arrays named `tainted_args`, `tainted_gets`, `tainted_alias` and `indirect` need to be bounds checked. The array and pointer `tainted_args` and `tainted_alias` get their values directly from the external interfaces - `argv` and the library call `gets` respectively. Hence, they are of type TAINTED. The object `indirect`'s value is defined by a use of `tainted_alias` pointer and hence it is also of type TAINTED. All pointer aliases to TAINTED objects are fat pointers. Also, the fat pointer `tainted_alias` will propagate TAINTED to the array `tainted_gets` on line 5. Finally, `safe_array` is determined to be SAFE because it is passed to a function `foobar`, which does not pass `safe_array` to external interfaces and does not assign `safe_array` data from an external interface.

are not necessary to guarantee security against buffer overflow attacks. The proposed optimization is based on the observation that only the accesses to objects that can hold data received from external interfaces need to be bounds checked.

4.1 Interface Analysis Algorithm

Buffer overflow attacks are launched by an adversary by providing malicious input through the external interfaces to the application. External sources of malicious inputs to the program include, the return values from the library calls such as *gets*, and the command line argument *argv*. In order to protect against buffer overflow attacks, it should be sufficient to bounds check accesses to only those objects that get their values from the external input. We call such objects (and their pointer aliases) as TAINTED and all the other objects as SAFE. A TAINTED object can get assigned to external input either directly from the external interfaces or indirectly from another TAINTED object. Figure 5 shows a simple code example to illustrate what would be labeled as TAINTED when performing our analysis, which we will go through in this section.

Limiting bounds checking to only TAINTED objects can decrease bounds checking overhead but at the same time provide a high level of security against buffer overflow attacks. Reduction in the performance overhead can result from the following two factors. First, we can eliminate the bounds checks for accesses to SAFE objects. Hence, we will be able to reduce the number of instructions executed to perform bounds checking. Second, we do not have to maintain bounds

```
1. Construct the inter-procedural data-flow graph, and initial pointer
   aliasing information for points-to.
2. We forward propagate the points-to alias relationships though the
   data-flow pointer assignment network, generating additional aliases.
3. All objects (including pointers) are initialized to type SAFE. All
   pointers are also initialized to type THIN.
4. Apply the TAINTED type qualifier to the pointers  and objects that
   are either assigned to the return values of the external interface
   functions, are passed as reference to external interface functions, or
   get assigned to the command line parameter ARGV.
5. Using the data-flow graph propagate TAINTED forward along scalar
   dependencies and mark them as TAINTED.
6. Add bounds checking to all pointers and array dereferences that are
   marked as TAINTED.
7. All pointers that are bounds checked are assigned to be type FAT.
8. Backwards propagate FATNESS through the pointer assignment network.
```

Fig. 6. Algorithm for interface optimization

information for all the pointers. This is because the pointers to the SAFE objects need to be only normal pointers instead of being fat pointers. We call the type of normal pointers as THIN and the type of fat pointers as FAT. Reducing the number of FAT pointers also reduces the overhead in initializing them and also copying them while passing those pointers as parameters to functions. More importantly, our optimization can reduce the memory footprint of the application and hence we can improve the cache performance.

Figure 6 shows the steps used for the Interface Analysis algorithm. The goal of the Interface Analysis is to find all the objects and their aliases that should be classified as TAINTED in order to perform bounds checking on their dereferences. As described earlier, a TAINTED object is one that gets assigned with external data either directly from an external interface or from another TAINTED object.

Our algorithm represents information using inter-procedural data-flow, and points-to graphs. The data-flow graph directly represents the explicit assignments of scalar and pointer values. The points-to graph represents the relationship between the pointer and its referenced object. These graphs operate on arrays, scalars and pointers objects with other types reduced to these basic types. The TAINTED and SAFE properties apply to all object, while FAT and THIN apply only to pointers.

The first step of the Interface Analysis is to construct the assignment data-flow graph, and to discover the initial points-to information from the pointer initialization. Address operators and dynamic memory allocator functions perform this initialization, returning the reference of an object to a pointer. Next we propagate the pointer alias relationship, building up our points-to database. We describe properties of the points-to maintained at this step in the following section 4.2. Third, we initialize all the pointer and object types to SAFE. The fourth step in our algorithm is to classify those pointers and objects that are assigned to the command line parameter `argv` and the return value of library calls as TAINTED. If this is a pointer, then the object referenced is TAINTED. Also, those objects whose value can be modified by library calls (pass by reference) are classified as TAINTED. In our example, in Figure 5, the objects `tainted_args` and `tainted_gets` will be classified as TAINTED after this step. In step five, we propagate the TAINTED type information forward along the scalar data-flow graph dependencies, including values from array references. We assume that operations other than copy (e.g. arithmetic) will destroy the taintedness of the scalar assignment. In addition, we use the points-to analysis to mark any pointers that reference a TAINTED object as TAINTED. This step iterates until the TAINTED set no longer changes. In doing this propagation, additional objects may be marked as TAINTED. After this propagation, the array `indirect` will get classified as TAINTED in our example code through forward propagation, and the array `tainted_gets` will be classified as TAINTED through points-to analysis. In step six, we add bounds checks to all dereferences of pointers and arrays that are marked as TAINTED. In seven, all pointers that are bounds checked will be marked as FAT, and the rest will be marked as THIN. In step eight we backwards propagate FAT through the pointer assignment network to initialization, ensuring bounds information can reach the check.

4.2 Aliasing Properties

We use points-to analysis to determine the objects that pointers alias [15–17] for two different purposes. Our first use is to allow pointers to determine if they reference a TAINTED object for which we use Andersen [16] to represent multiple alias relationships. From object aliasing we determine if the pointer references a TAINTED or SAFE object, consequently whether the pointer must be designated FAT or THIN. The second use is to fuse the multiple pointer-to-pointer aliases into a single class, as fusing simplifies how we use the alias information. This version of points-to helps us recognize nested pointers, and prevent conflicting pointer representations. Steensgaard [15] analysis does this fused points-to analysis for us. Consider as an example the type `char **`, which is a pointer to a char pointer `char *`. Variables that assign to or are assigned from the `char *` must have all of the same label, which is either THIN or FAT. TAINTED is similarly made consistent. Both points-to analysis use the pointer propagation data-flow to discover additional aliases.

Consider the following example of data-flow and alias analysis in figure 7. We propagate TAINTED forward through scalars to correctly mark the array `x`, where getchar() is an external interface that may attempt to inject insecure code. Dataflow discovers pointer assignment `z=y;` meaning `z` shares `y` aliases, and points-to analysis would discover that pointers `y` and `z` aliases `x`. The pointer `y` and `z` become FAT, and remain FAT even if they later in the control flow point to a SAFE object.

4.3 Memory Writer Algorithm

In addition to the above *interface* optimization, we also perform another optimization which we call the *memory-writer* optimization. Buffer overflow attacks for remote execution of malicious code are launched by writing beyond the boundaries of a buffer, which implies that we have to do bounds checking only for memory accesses that are writes in order to guard against write attacks.

```
char x[100]; int c, i=0;
char *y=x,*z;
while ((c = getchar()) != EOF){
  x[i++]=c;
}
z=y;
```

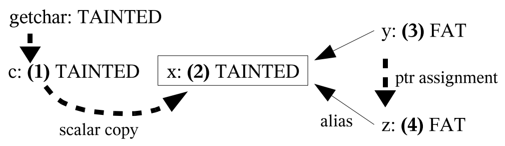

Fig. 7. TAINTED flow via scalar assignment and aliasing

Write attacks are the most common form of buffer overflow attack, and probably the most serious due to the possibility of its use in injecting malicious code. We now describe the memory writer algorithm that can be used by itself, and then explain how it can be used with the interface optimization.

Mem-Write Only - The first step is to mark all writes to array and pointer dereferences as being bounds checked. These are left hand side of assignments through array and pointer dereferences. All of these pointers are marked as FAT. The next step is to find any pointer that will directly or indirectly (through data-flow) define these writes, so that they will also be marked as FAT. They need to be FAT, since they need to also maintain bounds meta-data information for the bounds check. For this we start from the FAT writes and propagate the type FAT backwards, along the edges of the data-flow graph through pointer assignments, to find all pointers that *define* the value of any FAT pointer.

Mem-Write with Interface Optimization - For this approach, we first perform the interface algorithm in Figure 6. The only modification is step 6, where we only add bounds checking for arrays or pointers to buffers that are written as described above and marked as TAINTED.

4.4 Implementation Details

We build a data-flow framework to prune the unnecessary bounds checks. Our variable naming scheme is similar to the one used by Austin et.al. [6] capable of differentiating scalars, references, arrays and fields in structures. Next, we build a graph representing assignments of data-flow information, derived from program assignment statements, address operators, function parameters and return values. After building our data-flow graph on a per function level, we merge the graphs to create a global inter-procedural graph. Our type-based data-flow analysis is path-insensitive; a qualifier that is computed for a variable holds throughout the entire program. Changes to the representation are seen by the entire program, not just the path whose assignment caused it. Similarly, type information passed through procedure calls to other functions must be consistent across all the call sites as we permit only one representation (no function specialization) of that parameter inside of the function. For example, if a pointer parameter becomes FAT at one call site, then that same parameter will be labeled as FAT for all other call sites to that function. In other words, our inter-procedural analysis is context insensitive. Both path and context insensitivity greatly simplify the analysis. In addition, indirect function calls are also treated conservatively, where all possible function definitions that might match an indirect call site will have their parameters assigned with the same FAT or THIN label.

4.5 Network External Interface Results

Our analysis and the pruning of bounds checking can be applied to all external interfaces to an application, which would include disk, keyboard, mouse, cross-process communication, network traffic, etc. Or it could be applied to just a subset of these interfaces.

The current systems based upon dynamic taint analysis only focus on bounds checking accesses to tainted data received from the network [7–9], since this is the channel through which a worm attack can occur. Hence, we can limit bounds check to buffers that are passed to the network system calls, and any of the data in the program that is tainted by it.

To analyze this optimization, we performed our bounds checking analysis on the benchmark `ATPhttpd-0.4b`, which is a small web server, with a buffer overflow vulnerability [9]. In applying our external interface only guarding against write attacks as described above, we experience only a 6% slowdown over no bounds checking. We also verified that the vulnerability was caught using our taint-based bounds checking approach.

4.6 All External Interface Results

Since the SPEC integer benchmark suite does not have any network traffic, the amount of bounds checking is zero, which is not that interesting of a result to analyze. Therefore, we also examined applying our interface optimization for all system call interfaces to the program. For this set of benchmarks this means that we are bounds checking accesses to data that the application receives from the operating system interface, and anything tainted by it.

In this section we will analyze the advantages of our two optimizations, *interface* and *memory-writer*. The binaries that we used for this analysis are generated using the bounds check implementation that uses the x86 `bound` instruction. The code generator that we used is the same as the one used for generating the *bnd-array* binaries that we discussed in Section 3.3. We conducted this experiment on a AMD Athlon processor.

Figure 8 shows the performance advantage of our optimizations. The result labeled as *bnd-interface-only* corresponds to implementing only the interface optimization. The result corresponding to the label *bnd-mem-write* refers to our memory-writer only optimization, and *bnd-interface+mem-write* stands for the implementation where we applied both.

When each of the two optimizations are applied individually, the interface optimization reduces the overhead to 29%, whereas the memory-writer optimization reduces the overhead down to 28%. When both of the optimizations are applied together we find that the average overhead is reduced to 24%, which is a significant reduction when compared to our previous best result of 40% that we achieved using the *bnd_array* implementation.

The *bnd-interface-only* represents performing bounds checks and maintaining fat pointers for all tainted data coming from external interfaces. This provides protection against both write and read buffer overflow attacks. Since write buffer overflow attacks are the most harmful, *bnd-interface+mem-write* provides protection for all writes that write data from external interfaces.

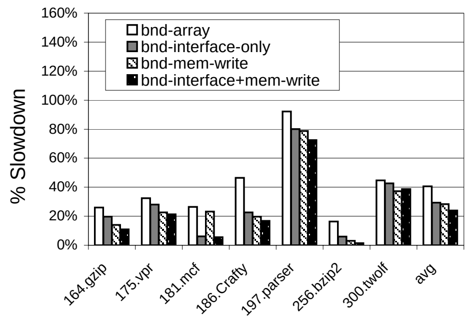

Fig. 8. Performance advantage of interface and memory-writer optimizations.

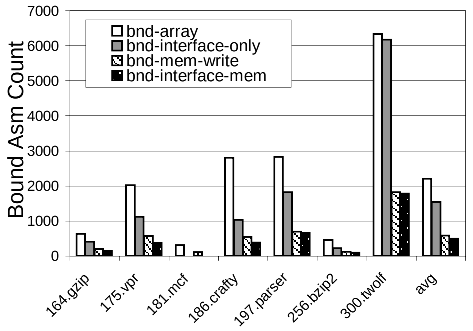

Fig. 9. Reduction in the number of static bounds instruction in the binary.

To analyze the main source of reduction in the performance overhead, in Figure 9 we show the number of *static* bounds check instructions that remain in the binary after applying our optimizations. We can see that the *bnd-array* implementation, where we bounds check all of the memory accesses through pointers, contains 2203 x86 bounds instructions on average. Our interface optimization which eliminates the bounds checks to the SAFE objects is able to remove 660 bounds checks from the binary to 1573 on average. The memory-writer optimization eliminates the bounds check to all the load memory operations. Hence, it significantly reduces the number of checks to 581 on average. When both the optimizations are combined together there are about 495 bounds checks left in the binary on average.

The performance savings shown in Figure 8 are proportional to the number of bounds checks that we managed to eliminate. We would like to highlight the result for our pointer intensive application `mcf`. For `mcf`, we see significant performance reduction for interface only optimization. The reason for this is that the there was a decent size reduction in the heap memory used as a result of our interface optimization as it managed to classify 50% of the pointers as THIN pointers.

4.7 Verification

We used Wilander security benchmarks [18] and the Accmon [19] programs with buffer overflow vulnerabilities to verify that our interface optimization and memory-writer optimization do not compromise security. For this analysis, we applied our taint-based bounds checking to all the system calls. Wilander provides a set of kernel tests that behave as if it is an adversary trying a buffer overflow exploit [18]. It checks for overwriting of the return address, base pointer, function pointers, and long jump buffer. This is done for the stack and heap over twenty tests. All the exploits in the Wilander and Accmon benchmarks were caught by our bounds checking mechanism that is optimized using the *interface* and *memory-writer* optimization techniques.

5 Conclusion

Bounds checking can prevent buffer overflow attacks. The key limitation in using bounds checking in the production software is its performance overhead, reducing this was the focus of our work. We first examined how to efficiently provide complete bounds checking for all pointer dereferences using the x86 `bound` instruction, which resulted in 40% overhead. We then examined performing bounds checks for only tainted data received from the external interfaces. This provides protection against both write and read buffer overflow attacks, and resulted in an average overhead of 29%. If we only care about write buffer overflow attacks, then we need to bounds check only the writes, which incurred 24% overhead. Finally, if we are only interested in bounds checking data from the network, we showed that overhead can be reduced to just 6%. Bounds checking provides greater protection than the hardware and dynamic taint based approaches [7–9] as it protects not only against control but also data buffer overflow attacks. In addition, it does not require any additional hardware [7, 8]. When compared to using dynamic emulation system [9] it incurs less performance overhead.

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by NSF grant CNS-0509546 and grants from Microsoft and Intel.

References

1. Ruwase, O., Lam, M.: A practical dyanmic buffer overflow detector. In: 11th Annual Network and Distributed Security Symposium (NDSS 2004), San Diego, California (2004) 159–169
2. G. McGary: Bounds Checking in C and C++ using Bounded Pointers (2000) http://gnu.open-mirror.com/software/gcc/projects/bp/main.html.
3. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy code. In: Symposium on Principles of Programming Languages. (2002) 128–139
4. Rinard, M., Cadar, C., Dumitran, D., Roy, D., Leu, T., Jr., W.S.B.: Enhancing server availability and security through failure-oblivious computing. In: 6th Symposium on Operating System Design and Implementation(OSDI). (2004)
5. Levy, E.: Smashing the stack for fun and profit. Phrack **7**(49) (1996)
6. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: Symposium on Programming Language Design and Implementation. (1994) 290–301
7. Suh, G., Lee, J., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems. (2004)
8. Crandall, J., Chong, F.: Minos: Control data attack prevention orthogonal to memory model. In: 37th International Symposium on Microarchitecture. (2004)
9. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: 12th Annual Network and Distributed System Security Symposium. (2005)
10. Markstein, V., Cocke, J., Markstein, P.: Optimization of range checking. In: Symposium on Compiler Construction. (1982) 114–119
11. Pettersson, M.: Hardware performance counter (2004) http://user.it.uu.se/ mikpe/linux/perfctr/.
12. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: USENIX Annual Technical Conference. (2002) 275–288
13. Jones, R., Kelly, P.: Backwards-compatible bounds checking for arrays and pointers in c programs. In: Automated and Algorithmic Debugging. (1997) 13–26

14. Harren, M., Necula, G.C.: Lightweight wrappers for interfacing with binary code in ccured. In: Software Security Symposium (ISSS'03). (2003)
15. Steensgaard, B.: Points-to analysis in almost linear time. In: Symposium on Principles of Programming Languages. (1996) 32–41
16. Andersen, L.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994) (DIKU report 94/19).
17. II, M.S., Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: Symposium on Principles of Programming Languages. (1997) 1–14
18. Wilander, J., Kamkar, M.: A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proceedings of the 10th Network and Distributed System Security Symposium. (2003) 149–162
19. Zhou, P., Liu, W., Fei, L., Lu, S., Qin, F., Midkiff, Y.Z.S., Torrellas, J.: Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In: 37st International Symposium on Microarchitecture. (2004)