

Fast Bounds Checking Using Debug Register

Tzi-cker Chiueh
Computer Science Department
Stony Brook University
chiueh@cs.sunysb.edu

Abstract. The ability to check memory references against their associated array/buffer bounds helps programmers to detect programming errors involving address overruns early on and thus avoid many difficult bugs down the line. This paper proposes a novel approach called *Boud* to the array bounds checking problem that exploits the debug register hardware in modern CPUs. *Boud* allocates a debug register to monitor accesses to an array or buffer within a loop so that accesses stepping outside the array's or buffer's bound will trigger a breakpoint exception. Because the number of debug registers is typically small, in cases when hardware bounds checking is not possible, *Boud* falls back to software bounds checking. Although *Boud* can effectively eliminate per-array-reference software checking overhead in most cases, it still incurs a fixed set-up overhead for each *use* of an array within a loop. This paper presents the detailed design and implementation of the *Boud* compiler, and a comprehensive evaluation of various performance tradeoffs associated with the proposed array bounds checking technique. For the set of real-world network applications we tested, including Apache, Sendmail, Bind, etc., the latency penalty of *Boud's* bounds checking mechanism is between 2.2% to 8.8%, respectively, when compared with the vanilla GCC compiler, which does not perform any bounds checking.

1 Introduction

Checking memory references against the bounds of the data structures they belong to at run time provides a valuable tool for early detection of programming errors that could have otherwise resulted in subtle bugs or total application failures. In some cases, these software errors might lead to security holes that attackers exploit to break into computer systems and cause substantial financial losses. For example, the buffer overflow attack, which accounts for more than 50% of the vulnerabilities reported in the CERT advisory over the last decade [3, 15, 20], exploits the lack of array bounds checking in the compiler and in the applications themselves, and subverts the victim programs to transfer control to a dynamically injected code segment. Although various solutions have been proposed to subjugate the buffer overflow attack, inoculating application programs with strict array bounds checking is considered the best defense against this attack. Despite these benefits, in practice most applications developers still choose to shy away from array bounds checking because its performance

overhead is considered too high to be acceptable [13]. This paper describes a novel approach to the array bounds checking problem that can reduce the array bounds checking overhead to a fraction of the input program’s original execution time, and thus make it practical to apply array bounds checking to real-world programs.

The general problem of bounds checking requires comparing the target address of each memory reference against the bound of its associated data structure, which could be a statically allocated array, or a dynamically allocated array or heap region. Accordingly, bounds checking involves two subproblems: (1) identifying a given memory reference’s associated data structure and thus its bound, and (2) comparing the reference’s address with the bound and raising an exception if the bound is violated. The first subproblem is complicated by the existence of pointer variables. As pointers are used in generating target memory addresses, it is necessary to carry with pointers the ID of the objects they point to, so that the associated bounds could be used to perform bounds checking. There are two general approaches to this subproblem. The first approach, used in BCC [4], tags each pointer with additional fields to store information about its associated object or data structure. These fields could be a physical extension of a pointer, or a shadow variable. The second approach [12] maintains an index structure that keeps track of the mapping between high-level objects and their address ranges, and dynamically searches this index structure with a memory reference’s target address to identify the reference’s associated object. The first approach performs much faster than the second, but at the expense of compatibility of legacy binary code that does not support bounds checking. The second subproblem accounts for most of the bounds checking overhead, and indeed most of the research efforts in the literature were focused on how to cut down the performance cost of address-bound comparison, through techniques such as redundancy elimination or parallel execution. At the highest compiler optimization level, the minimum number of instructions required in BCC [4], a GCC-derived array bounds checking compiler, to check a reference in a C-like program against its lower and upper bounds is 6, two to load the bounds, two comparisons, and two conditional branches. For programs that involve many array/buffer references, software-based bounds checking still incurs a substantial performance penalty despite many proposed optimizations. In this paper, we propose a new approach, called *Boud*¹, which exploits the debug register hardware support available in mainstream CPUs to perform array bounds checking *for free*. The basic idea is to use debug registers to watch the end of each array being accessed, and raise an alarm when its bound is exceeded. Because debug registers perform address monitoring transparently in hardware, *Boud*’s approach to checking array bounds violation incurs no *per-array-reference overhead*. In some cases, hardware bounds checking is not possible, for example, when all debug registers are used up, and *Boud* falls back to traditional software bounds checking. Therefore, the overhead of *Boud* mainly comes from debug register set-up

¹ BOUNDS checking Using Debug register

required for hardware bounds checking, and occasional software-based bounds checking.

The general bounds checking problem requires checking for each memory reference, including references to a field within a C-like structure. Because *Boud* incurs a per-array-use set-up overhead, the debug register approach only makes sense for array-like references inside a loop, i.e., those of the form `A[i]`, `A++`, `++A`, `A--`, or `--A`, where `A` could be a pointer to a static array or a dynamically allocated buffer. For example, if a dynamic buffer is allocated through a `malloc()` call of the following form

```
X = (* TYPE) malloc(N * sizeof(TYPE))
```

where N is larger than 1, then *Boud* takes `X` as a pointer into an array of N elements, and *Boud* will check the references based on `X` if these references are used inside a loop. For array-like references outside loops, *Boud* applies conventional software-based bounds checking.

The rest of this paper is organized as follows. Section 2 reviews previous work on array bound checking and contrasts *Boud* with these efforts. Section 3 describes the detailed design decisions of the *Boud* compiler and their rationale. Section 4 presents a performance evaluation of the *Boud* compiler based on a set of array-intensive programs, and a discussion of various performance overheads associated with the *Boud* approach. Section 5 concludes this paper with a summary of the main research ideas and a brief outline of the on-going improvements to the *Boud* prototype.

2 Related Work

Most previous array bounds checking research focused on the minimization of run-time performance overhead. One notable exception is the work from the Imperial College group [12], which chose to attack the reference/object association problem in the presence of legacy library routines. The general approach towards optimizing array bounds checking overhead is to eliminate unnecessary checks, so that the number of checks is reduced. Gupta [17, 18] proposed a flow analysis technique that avoids redundant bounds checks in such a way that it still guaranteed to identify any array bound violation in the input programs, although it does not necessarily detect these violations immediately after they occur at run time. By trading detection immediacy for reduced overhead, this approach is able to hoist some of the bounds checking code outside the loop and thus reduce the performance cost of array bounds checking significantly. Asuru [11] and Kolte and Wolfe [13] extended this work with more detailed analysis to further reduce the range check overhead.

Concurrent array bounds checking [6] first derives from a given program a reduce version that contains all the array references and their associated bounds checking code, and then runs the derived version and the original version on separate processors in parallel. With the aid of a separate processor, this approach

is able to achieve the lowest array bounds checking overhead reported until the arrival of *Boud*.

Unlike most other array bounds checking compiler projects, the Bounds Checking GCC compiler (BCC) checks the bounds for both array references and general pointers. Among the systems that perform both types of bounds checks, BCC shows the best software-only bounds checking performance. However, BCC only checks the upper bound of an array when the array is accessed directly through the array variable (not pointer variable) while *Boud* automatically checks both the upper and lower bounds. Since the *Boud* compiler is based on BCC, it can also check the bounds for general pointers.

The array bounds checking problem for Java presents new design constraints. Because bounds checking code cannot be directly expressed at bytecode level, elimination of bounds checks can only be performed at run time, after the bytecode program is loaded. Directly applying existing array bounds checking optimizers at run time is not feasible, however, because they are too expensive for dynamic compilation. ABCD [19] is a light-weight algorithm for elimination of Array Bounds Checks on Demand, which adds a few edges to the SSA data flow graph and performs a simple traversal of the resulting graph. Despite its simplicity, ABCD has been proven quite effective.

Intel X86 architecture includes a `bound` instruction [9] for array bounds checking. However, the `bound` instruction is not widely used because on 80486 and Pentium processors, the `bound` instruction is slower than the six normal equivalent instructions. The `bound` instruction requires 7 cycles on a 1.1 GHz P3 machine while the 6 equivalent instructions require 6 cycles.

Previously, we developed an array bounds checking compiler called CASH [14] that exploits the segment limit checking hardware in Intel's X86 architecture [10] and successfully reduces the performance penalty of array bounds checking of large network applications such as Apache, Bind and Sendmail under 9%. The basic idea of CASH is to allocate a segment for each statically allocated array or dynamically allocated buffer, and then generate array/buffer reference instructions in such a way that the segment limit check hardware performs array bounds checking for free. Because CASH does not require software checks for individual buffer/array references, it is the world's fastest array bounds checking compiler for C programs on Intel X86 machines. Unfortunately, the segment limit check hardware exists only on the Intel IA32 architecture. It is not supported even on AMD64, EMT64 or Itanium, let alone in other RISC processors such as Sparc, MIPS, ARM, Xscale or PowerPC.

Qin et al. [16] proposed to exploit the fine-grained memory address monitoring capability of physical memory error correction hardware to detect array bound violations and memory leaks. Although the conceptual idea of this ECC-based scheme is similar to *Boud*, there are several important differences. First, the minimal granularity of the ECC-based scheme is a cache line rather than an individual word as in the case of *Boud*. Second, the ECC-based scheme did not seem to be able to handle array references with arbitrary strides. Third, setting up a honeypot-like cache line in the ECC-based scheme requires not only making

a system call, but also enabling/disabling multiple hardware components, and is thus considered very expensive. Finally, implementation of the ECC-based scheme may be device-specific and is thus not generally portable across different hardware platforms.

3 The Boud Approach

3.1 Association between References and Referents

To check whether a memory reference violates its referent’s bound, one needs to identify its referent first. To solve this reference-object association problem, *Boud* allocates a metadata structure for each high-level object, for example, an array or a buffer, that maintains such information as its lower and upper bounds. Then *Boud* augments each *stand-alone* pointer variable P with a shadow pointer P_A that points to the metadata structure of P ’s referent. P and P_A then form a new fat pointer structure that is still pointed to by P . Both P and its P_A are copied in all pointer assignment/arithmetic operations, including binding of formal and actual pointer arguments in function calls. Because P and P_A are guaranteed to be adjacent to each other, it is relatively straightforward to identify the bounds of a pointer’s referent by following its associated shadow pointer. For array bounds checking purpose, each array or buffer’s metadata structure contains its 4-byte lower and upper bounds. For example, when a 100-byte array is statically allocated, *Boud* allocates 108 bytes, with the first two words dedicated to this array’s information structure. The same thing happens when an array is allocated through `malloc()`.

For pointer variables that are embedded into a C structure or an array of pointers, the fat pointer scheme is problematic because it may break programs that make assumptions about the memory layout or size of these aggregate structures. For these *embedded* pointer variables, *Boud* uses an index tree [12] scheme to identify their referents. That is, when a memory object is created, *Boud* inserts a new entry into the index tree based on the object’s address range, and the new entry contains a pointer pointing to the memory object’s metadata structure. When an embedded pointer variable is dereferenced or copied to a stand-alone pointer variable, *Boud* inserts code to look up the index tree with the pointer variable’s value to locate its referent’s bound. Although index tree look-up is slower than direct access using shadow pointer, the performance cost is small in practice as most pointer variables are stand-alone rather than embedded.

Fat pointers could raise compatibility problems when a *Boud* function interacts with a legacy function, because legacy functions do not expect the additional shadow pointers. To solve this problem, *Boud* allocates a shadow stack, one per thread, to pass the shadow pointers of pointer arguments. Specifically, before calling a function, the caller places in the shadow stack the entry point of the callee and the shadow pointers of all input arguments that are pointers. If a *Boud* callee is called, it first checks if the first argument in the shadow stack matches its entry point, if so composes its fat pointer variables using the shadow pointers on the shadow stack, and finally removes these shadow stack entries.

When a *Boud* callee returns a pointer, it uses the same shadow stack mechanism to record the associated return address and the shadow pointer of the returned pointer. If the caller is also compiled by *Boud*, it compares the return address on the shadow stack with the call site, and if matched composes a fat pointer representation for the returned pointer based on the shadow pointer on the shadow stack.

When a legacy function calls a *Boud* callee, the callee’s comparison between its entry point and the first argument on the shadow stack fails, and the callee simply assigns the shadow pointers of all its pointer arguments to NULL. When a *Boud* callee returns, the shadow pointer that it puts on the shadow stack is ignored by the legacy caller. When a *Boud* function calls a legacy function, the information that the caller puts on the shadow stack is ignored by the callee. When the legacy callee returns, the caller’s comparison of its return address and the return address on the shadow stack fails, and the caller then sets the shadow pointer of the returned pointer to NULL and continues. *Boud*’s shadow stack mechanism effectively solves the pointer argument passing problem associated with fat pointers when a legacy function calls a *Boud* function and when a *Boud* function calls a legacy function.

3.2 Debug Register in Intel X86 Architecture

A key innovation in *Boud* is its use of debug register in detecting array bounds violation. Debug register hardware is universally supported by most if not all mainstream CPUs such as Intel’s 32-bit and 64-bit X86, Itanium, ARM, SPARC, MIPS, PowerPC, etc. In addition, the interfaces these CPUs expose to the software are largely the same. In this paper, we will focus only on the Intel X86 processor [10]. However, the technique described below is equally applicable to other CPUs without much modification.

Debug register is designed to support instruction and data breakpointing functions required by software debuggers. In the X86 architecture, there are totally eight debug registers (DB0 through DB7) and two model-specific registers (MSRs). Among them, DB0 to DB3 are used to hold memory addresses or I/O locations that the debugger wants to monitor. Whenever a memory or instruction address matches the contents of one of these four registers, the processor raises a debug exception. With this support, the debugger does not need to perform expensive intercept-and-check in software. DB4 and DB5 are reserved. DB6 keeps the debugger status while DB7 is for control/configuration. The detailed layout of these DR registers is shown in Figure 1.

The primary function of the four *debug address registers* (DR0 to DR3) is for holding 32-bit linear breakpoint addresses. The hardware compares every instruction/data memory address with these breakpoint addresses in parallel with the normal virtual to physical address translation, and thus incurs no additional performance overhead. The *debug status register* (DR6) reports the status of the debugging conditions when a debugging exception is generated. For example, B_n bit signifies that the n th breakpoint was reached. BS and BT bits indicate the exception is due to single stepping and task switching, respectively. The

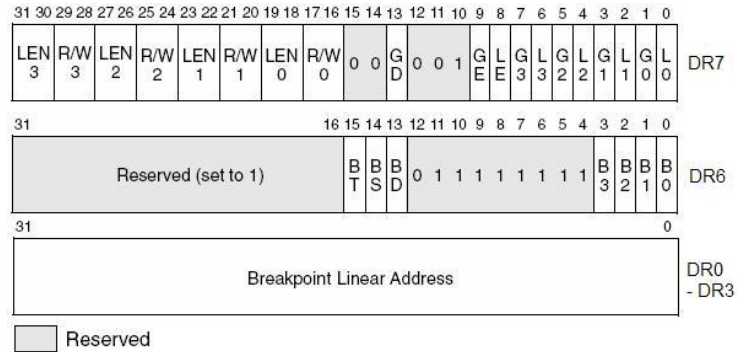


Fig. 1. Debug registers in Intel X86 processor [10]. DR4 and DR5 are reserved and thus not shown. The semantics of individual fields are explained in the text.

debug control register (DR7) allows fine-grained control over each breakpoint condition. The n th breakpoint could be enabled or disabled through setting the corresponding L_n or G_n bit in DR7. L_n bit enables the n th breakpoint for the current task while G_n bit is for all tasks. When both bits are cleared, the corresponding breakpoint is essentially disabled. R/W_n field controls the access mode of the n th breakpoint. For example, value 11 means breaking on either data read or write but not instruction fetch. LEN_n field specifies the size of the memory location associated with the n th breakpoint. Value 00, 01 and 11 indicate length of 1-byte, 2-byte and 4-byte respectively. For other fields that are not directly related to this project, please refer to the IA32 architectural manual [10] for their interpretation.

3.3 Detecting Bounds Violations Using Debug Registers

Fundamentally, debug register hardware provides an efficient way to detect situations when the CPU accesses certain memory addresses. *Boud* exploits this capability to detect array bounds violations by monitoring the boundary words of the arrays accessed within a loop. Almost all known buffer overflow vulnerabilities occur in the context of a loop. The attacker’s input steps through an array/buffer from one end to the other and eventually outside of the upper or lower bound. To apply debug register to array bounds checking, *Boud* allocates an extra memory word above and below each array/buffer as a *honeypot word*, and puts the addresses of an array’s honeypot words in debug registers before the array is accessed. Because honeypot words are introduced by the compiler and thus transparent to the program, they should never be accessed at run time. Therefore, when a honeypot word in a program is read or written, a breakpoint exception is raised and the program is terminated immediately as it signifies an attempt to overflow an array/buffer. As in the case of *Cash*, this approach does not require any software-based bounds check for each array/buffer reference.

For an array/buffer reference statement within a loop, *Boud* allocates a debug register and determines the address of the honeypot word that should be put into the debug register. Because debug registers are used in bounds checking, they become part of a process's context, and therefore need to be saved and restored across context switch and function call/return. Moreover, because some debuggers also use debug registers, the debug register set-up instructions inserted by *Boud* should be disabled when a program is being debugged.

There are three implementation issues associated with *Boud*'s debug register-based array bounds checking mechanism. First, because debug registers are privileged resources, they can only be modified inside the kernel. This means a user application needs to make a system call to modify debug registers, even within a small function that accesses a temporary local array. *Boud* uses two following two techniques to mitigate this performance problem. *Boud* sets up *all* debug registers required in a called function using a single system call, so that the fixed system call invocation overhead (about 200 CPU cycles) is amortized over multiple debug register set-ups. In addition, *Boud* implements a user-level debug register cache that contains the current contents of debug registers. When a user application needs a debug register for bounds checking, *Boud*'s run-time library first checks the user-level cache to see if the corresponding honeypot word address is already in some debug registers, and returns the matched debug register if there it is. If the target honeypot word address is not in any debug register, *Boud*'s run-time library allocates a debug register and makes a system call to put the target honeypot word in the chosen debug register. Empirically this debug register cache saves many unnecessary system calls, and is particularly useful for programs that repeatedly call a function with local arrays within a loop.

Second, most CPUs, including the Intel IA32/X86 architecture, support only 4 debug registers. If the *Boud* compiler requires two debug registers to check each array's upper/lower bound, it can guard at most two arrays or buffers at a time using this mechanism, and has to resort to software bounds checks for other arrays/buffers that are being accessed simultaneously. Fortunately, for most programs, when an array/buffer is accessed within a loop, one only needs to monitor its upper or lower bound but not both; therefore only one debug register is needed. To reduce the number of debug registers per array to one, the *Boud* compiler statically analyzes a program to determine the direction in which each array is accessed (increment or decrement), and sets up a debug register to protect its lower or upper bound accordingly.

Finally, debug register hardware is less powerful than Cash's segment limit check hardware because the former performs *point* check whereas the latter performs *range* check. As a result, it is possible that debug register may fail to detect certain bound violations that segment limit check hardware can catch. We have identified two corner cases in which debug register-based check alone may be ineffective. The first case is exemplified in the following code:

```
int a[10];
int i;
int *p;
```

```

p = &a[10]+5;
for(i = 0; i < 10; i++)
    *p++ = i;

```

Because the first array reference is already outside the loop, no accesses within the loop will touch the associated honeypot word and the debug register hardware cannot catch this overflow. In contrast, a normal buffer overflow attack typically starts from a legitimate element within the buffer and eventually progresses beyond its bound. To solve this problem, the *Boud* compiler checks the first array reference separately in software to ensure that it is within the array's bound, and then checks the remaining array references within the loop using debug register.

The second case in which debug register hardware is ineffective is when the array references within a loop happen to skip the honeypot word. For example, when an array reference progresses in steps of 2 words and the honeypot word is one word beyond the last array element, even if the references step outside the array's bound, the debug register cannot help much because the honeypot word is never accessed. For example,

```

int a[10];
for(i = 0; i < 10; i++)
    a[2*i+1] = i;

```

Boud solves this problem by statically determining the index gap of each within-loop array reference and allocating the honeypot words accordingly. This means that *Boud* may need to allocate multiple honeypot words for an array's lower and upper bound: If the maximum index gap used in a program to traverse through an array *A* is *K*, then the *Boud* compiler allocates *K* honeypot words for both *A*'s lower and upper bounds. However, at run time, only one of these *K* honeypot words is monitored.

For some within-loop array references, their direction of traversal or index gap cannot be determined statically, but they are fixed throughout the loop at run time. For these array references, *Boud* generates code to extract their direction of traversal or index gap and use the extracted values to set up honeypot words and debug registers accordingly at run time. Consequently, even for this type of array references, *Boud* still needs only one debug register per array reference.

3.4 Optimizations

Setting up a debug register requires making a system call. One way to reduce the performance cost of debug register set-up is to cache the contents of recently de-allocated debug registers and reuse them whenever possible. More concretely, *Boud* maintains a `free_dr_entry` list in user space to keep track of the contents of recently de-allocated debug registers. Whenever a debug register is de-allocated, *Boud* does not go into the kernel to modify the debug register; instead it puts the debug register's ID and content to the `free_dr_entry` list. Whenever a debug register is to be allocated, *Boud* checks the memory address to be monitored

against the contents of the debug registers in the `free_dr_entry` list. If there is a match, *Boud* uses the debug register whose content matches the monitored address, and chooses any available debug register otherwise.

When a new monitored memory address matches the previous contents of a currently available debug register, *Boud* does not need to make a system call to modify the matched debug register, because it already contains the desired value. This optimization dramatically reduces the frequency of debug register set-up system calls for functions that contain local arrays and are called many times inside a loop.

There are only four debug registers in the X86 architecture. *Boud* allocates debug registers on a first-come-first-serve basis. The first three arrays the *Boud* compiler encounters during the parsing phase inside a (possibly nested) loop are assigned one of the three debug registers. If more than three arrays are involved within a loop, *Boud* falls back to software array bounds checking for references associated with those arrays beyond the first three.

4 Performance Evaluation

4.1 Methodology

The current *Boud* compiler prototype is derived from the Bounds Checking GCC [4], which is derived from GCC 2.96 version, and runs on Red Hat Linux 7.2. We chose BCC as the base case for the two reasons. BCC is one of the most advanced array bounds checking compilers available to us, boasting a consistent performance penalty of around 100%. It has been heavily optimized. The more recent bounds checking performance study from University of Georgia [2] also reported that the average performance overhead of BCC for a set of numerical kernels is around 117% on Pentium III. Moreover, all previous published research on software-based array bounds checking for C programs *always* did far worse than BCC. Finally, the fact that BCC and *Boud* are based on the same GCC code basis makes the comparison more meaningful. Existing commercial products such as Purify are not very competitive. Purify is a factor of 5-7 slower than the unchecked version because it needs to perform check on every read and write. The VMS compiler and Alpha compiler also supported array bounds checking, but both are at least twice as slow compared with the unchecked case on the average. In all the following measurements, the compiler optimization level of both BCC and *Boud* is set to the highest level. All test programs are statically linked with all required libraries, which are also recompiled with *Boud*.

To understand the quantitatively results of the experiments run on the *Boud* prototype presented in the next subsection, let's first analyze qualitatively the performance savings and overheads associated with the *Boud* approach. Compared with BCC, *Boud*'s bounds checking mechanism does not incur any per-array-reference overhead, because it exploits debug register hardware to detect array bound violations. However, there are other overheads that exist only in *Boud* but not in BCC. First, there is a *per-program overhead*, which results from

the initial set-up of the debug register cache. Then there is a *per-array overhead*, which is related to debug register setting and resetting. On a Pentium-III 1.1-GHz machine running Red Hat Linux 7.2, the measured *per-program overhead* is 98 cycles and the *per-array overhead* is 253 cycles.

Program Name	Lines of Code	Brief Description	Array-Using Loops	> 3 Arrays
Toast	7372	GSM audio compression utility	51	6 (0.6%)
Cjpeg	33717	JPEG compression utility	236	38 (1.5%)
Quat	15093	3D fractal generator	117	19 (3.4%)
RayLab	9275	Raytracer-based 3D renderer	69	4 (0.2%)
Speex	16267	Voice coder/decoder	220	23 (2.8%)
Gif2png	47057	Gif to PNG converter	277	9 (1.3%)

Table 1. Characteristics of a set of batch programs used in the macro-benchmarking study. The source code line count includes all the libraries used in the programs, excluding `libc`.

Program Name	GCC	<i>Boud</i>	BCC
Toast	4,727,612K	4.6%	47.1%
Cjpeg	229,186K	8.5%	84.5%
Quat	9,990,571K	15.8%	238.3%
RayLab	3,304,059K	4.5%	40.6%
Speex	35,885,117K	13.3%	156.4%
Gif2png	706,949K	7.7%	130.4%

Table 2. The performance comparison among GCC, BCC, and *Boud* based on a set of batch programs. GCC numbers are in thousands of CPU cycles, whereas the performance penalty numbers of *Boud* and BCC are in terms of execution time percentage increases with respect to GCC.

4.2 Batch Programs

We first compare the performance of GCC, BCC, and *Boud* using a set of batch programs, whose characteristics are listed in Table 1, and the results are shown in Table 2. In general, the performance difference between *Boud* and BCC is pretty substantial. In call cases, the performance overhead of *Boud* is below 16%, whereas the worst-case performance penalty for BCC is up to 238%.

A major concern early in the *Boud* project is that the number of debug registers is so small that *Boud* may be forced to frequently fall back to the software-based bounds check. Because *Boud* only checks array references within

loops, a small number of debug registers is a problem only when the body of a loop uses more than 3 arrays/buffers. That is, the limit on the number of simultaneous array uses is per loop, not per function, or even per program. To isolate the performance cost associated with this problem, we measure the number of loops that involve array references, and the number of loops that involve more than 3 distinct arrays (called *spilled loops*) during the execution of the test batch programs (assuming one debug register is reserved for the debugger). The results are shown in Table 1, where the percentage numbers within the parenthesis indicate the percentage of loop iterations that are executed in the experiments and that belong to spilled loops. In general, the majority of array-referencing loops in these programs use fewer than 5 arrays. Furthermore, the percentage of spilled loop iterations seems to correlate well with the overall performance penalty. For example, the two programs that exhibit the highest spilled loop iteration percentage, Quat and Speex, also incur the highest performance penalty under *Boud*.

Program Name	Lines of Code	Array-Using Loops	> 3 Arrays
Qpopper-4.0	32104	67	1 (0.9%)
Apache-1.3.20	51974	355	12 (0.5%)
Sendmail-8.11.3	73612	217	24 (1.4%)
Wu-ftpd-2.6.1	28055	138	1 (0.4%)
Pure-ftpd-1.0.16b	22693	45	1 (0.5%)
Bind-8.3.4	46844	734	22 (0.6%)

Table 3. Characteristics of a set of Popular network applications that are known to have buffer overflow vulnerability. The source code line count includes all the libraries used in the programs, excluding `libc`.

4.3 Network Applications

Because a major advantage of array bounds checking is to stop remote attacks that exploit buffer overflow vulnerability, we apply *Boud* to a set of popular network applications that are known to have such a vulnerability. The list of applications and their characteristics are shown in Table 3. At the time of writing this paper, BCC still cannot correctly compile these network applications. because of a BCC bug [4] in the `nss` (name-service switch) library, which is needed by all network applications. Because of this bug, the bounds-checking code BCC generates will cause spurious bounds violations in `nss_parse_service_list`, which is used internally by the GNU C library’s name-service switch. Therefore, for network applications, we only compare the results from *Boud* and GCC.

To evaluate the performance of network applications, we used two client machines (one 700-MHz Pentium-3 with 256MB memory and the other 1.5-GHz

Program Name	Latency Penalty	Throughput Penalty	Space Overhead
Qpopper	5.4%	5.1%	58.1%
Apache	3.1%	2.9%	51.3%
Sendmail	8.8%	7.7%	39.8%
Wu-ftpd	2.2%	2.0%	62.3%
Pure-ftpd	3.2%	2.8%	55.4%
Bind	4.1%	3.9%	48.7%

Table 4. The latency/throughput penalty and space overhead of each network application compiled under *Boud* when compared with the baseline case without bounds checking.

Pentium-4 with 512 MB memory), that continuously send 2000 requests to a server machine (2.1-GHZ Pentium-4 with 2 GB memory) over a 100Mbps Ethernet link. The server machine’s kernel was modified to record the creation and termination time of each forked process. The throughput of a network application running on the server machine is calculated by dividing 2000 with the time interval between creation of the first forked process and termination of the last forked process. The latency is calculated by taking the average of the CPU time used by the 2000 forked processes. The Apache web server program is handled separately in this study. We configured Apache to handle each incoming request with a single child process so that we could accurately measure the latency of each Web request.

We measured the latency of the most common operation for each of these network applications when the bounds checking mechanism in *Boud* is turned on and turned off. The operation measured is sending a mail for Sendmail, retrieving a web page for Apache, getting a file for Wu-ftpd, answering a DNS query for Bind, and retrieving mails for Qpopper. For network applications that can potentially involve disk access, such as Apache, we warmed up the applications with a few runs before taking the 10 measurements used in computing the average. The throughput penalty for these applications ranges from 2.0% (Wu-ftpd) to 7.7% (Sendmail), and the latency penalty ranges from 2.2% (Wu-ftpd) to 8.8% (Sendmail), as shown in Table 4. In general, these numbers are consistent with the results for batch programs, and demonstrate that *Boud* is indeed a highly efficient bounds checking mechanism that is applicable to a wide variety of applications. Table 4 also shows the increase in binary size due to *Boud*, most of which arises from tracking of the reference-object association.

Table 3 also shows the percentage of spilled loop iterations for each tested network applications, which is below 3.5% for all applications except Sendmail, which is at 11%. Not surprisingly, Sendmail also incurs the highest latency and throughput penalty among all tested network applications.

One major concern is the performance cost associated with debug register setting and resetting, which require making system calls. Among all tested applications, Toast makes the most requests (415,659 calls) to allocate debug registers.

223,781 of them (or 53.8% hit ratio) can find a match in the 3-entry debug register cache and 191,878 requests actually need to go into the kernel to set up the allocated debug register. Each call gate invocation takes about 253 cycles, which means that it takes 50,464K cycles for the 191,878 calls, and this is relatively insignificant as compared with Toast's total run time (4,727,612K cycles). Therefore, the overhead of the Toast application compiled under *Boud* is still very small (4.6%) though it makes a large number of debug register requests.

5 Conclusion

Although array bounds checking is an old problem, it has seen revived interest recently out of concerns on security breaches exploiting array bound violation. Despite its robustness advantage, most real-world programs do not incorporate array bounds checking, because its performance penalty is too high to be considered practical. Whereas almost all previous research in this area focused on static analysis techniques to reduce redundant bounds checks and thus minimize the checking overhead, this work took a completely different approach that relies on the debug register hardware feature available in most mainstream CPUs. By leveraging debug registers' address monitoring capability, *Boud* can accurately detect array bound violations almost for free, i.e., without incurring any per-array-reference overhead most of the time. We have successfully built a *Boud* prototype based on the bound-checking GCC compiler under Red Hat Linux 7.2. The current *Boud* prototype can check bound violations for array references both within and outside loops, although it applies debug register-based bounds checking only to within-loop array references. Performance measurements collected from running a set network applications on the *Boud* prototype demonstrate that the throughput and latency penalty of *Boud*'s array bounds checking mechanism is below 7.7% and 8.8%, respectively, when compared with the vanilla GCC compiler, which does not perform any bounds checking. This puts *Boud* as one of the fastest array bounds checking compilers ever reported in the literature for C programs on the X86 architecture.

References

1. Bruce Perens. Electric fence: a malloc() debugger for linux and unix. <http://perens.com/FreeSoftware/>.
2. Chris Bentley, Scott A. Watterson, and David K. Lowenthal. A comparison of array bounds checking on superscalar and vliw architectures. *submitted to the annual IEEE Workshop on Workload Characterization*, September 2002.
3. Crispian Cowan, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.
4. GCC. Bounds-checking gcc. <http://www.gnu.org/software/gcc/projects/bp/main.html>.
5. Glenn Pearson. Array bounds checking with turbo c. *Dr. Dobb's Journal of Software Tools*, 16(5):72, 74, 78–79, 81–82, 104–107, May 1991.

6. Harish Patil and Charles N. Fischer. Efficient run-time monitoring using shadow processing. In *Proceedings of Automated and Algorithmic Debugging Workshop*, pages 119–132, 1995.
7. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, 1998.
8. Hongwei Xi and Songtao Xia. Towards array bound check elimination in java virtual machine language. In *Proceedings of CASCOS '99*, pages 110–125, Mississauga, Ontario, November 1999.
9. Intel. IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference. <http://www.intel.com/design/Pentium4/manuals/>.
10. Intel. Ia-32 intel architecture software developer's manual. volume 3: System programming guide. <http://developer.intel.com/design/pentium4/manuals/245472.htm>.
11. J. M. Asuru. Optimization of array subscript range checks. *ACM letters on Programming Languages and Systems*, 1(2):109–118, June 1992.
12. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of Automated and Algorithmic Debugging Workshop*, pages 13–26, 1997.
13. P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–278, 1995.
14. Lap-chung Lam and Tzi-cker Chiueh. Checking array bound violation using segmentation hardware. *Proceedings of 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, June 2005.
15. Manish Prasad and Tzi-cker Chiueh. A binary rewriting approach to stack-based buffer overflow attacks. In *in Proceedings of 2003 USENIX Conference*, June 2003.
16. F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, February 2005.
17. Rajiv Gupta. A fresh look at optimizing array bound checking. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 272–282, 1990.
18. Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March-December 1993.
19. Rastislav Bodik and Rajiv Gupta and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.
20. Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compiler time solution to buffer overflow attacks. In *in Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, April 2001.
21. Tzi-cker Chiueh and Ganesh Venkitachalam and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *in Proceedings of 17th ACM Symposium on Operating Systems Principles*, Charleston, SC, December 1999.