Segment Protection for Embedded Systems Using Run-time Checks

Matthew Simpson Bhuvan Middha Rajeev Barua

Department of Electrical & Computer Engineering
University of Maryland
College Park, MD 20742, USA

{simpsom, bhuvan, barua}@eng.umd.edu

ABSTRACT

The lack of virtual memory protection is a serious source of unreliability in many embedded systems. Without the segment-level protection it provides, these systems are subject to memory access violations, stemming from programmer error, whose results can be dangerous and catastrophic in safety-critical systems. The traditional method of testing embedded software before its deployment is an insufficient means of detecting and debugging all software errors, and the reliance on this practice is a severe gamble when the reliable performance of the embedded device is critical. Additionally, the use of safe languages and programming semantic restrictions as prevention mechanisms is often infeasible when considering the adoptability and compatibility of these languages since most embedded applications are written in C and C++.

This work improves system reliability by providing a completely automatic software technique for guaranteeing segment protection for embedded systems lacking virtual memory. This is done by inserting optimized run-time checks before memory accesses that detect segmentation violations in cases in which there would otherwise be no error, enabling remedial action before system failure or corruption. This feature is invaluable for safety-critical embedded systems. Other advantages of our method include its low overhead, lack of any programming language or semantic restrictions, and ease of implementation. Our compile-time analysis, known as intended segment analysis, is a uniquely structured analysis that allows for the realization of optimizations used to reduce the number of required run-time checks and foster our technique into a truly viable solution for providing segment protection in embedded systems lacking virtual memory.

Our experimental results show that these optimizations are effective at reducing the performance overheads associated with providing software segment protection to low, and in many cases, negligible levels. For the eight evaluated embedded benchmarks, the average increase in run-time is 0.72%, the average increase in energy consumption is 0.44%, and the average increase in code size is 3.60%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'05, September 24–27, 2005, San Francisco, CA, USA.
Copyright 2005 ACM 1-59593-149-X/05/0009 ...$5.00.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Storage Management; C.4 [**Performance of Systems**]: Fault Tolerance; C.3 [**Special-purpose and Application-based Embedded Systems**]: Real-time and Embedded Systems

General Terms

Languages, Reliability

Keywords

Compilers, Embedded Systems, Memory Safety, MMU, MPU, Reliability, Run-time Checks, Safe Languages, Segment Protection, Segmentation Violations, Virtual Memory

1. INTRODUCTION

Embedded processors are used to control various safety-critical systems, including automotive, aerospace, industrial, and healthcare applications, where the consequences of a system failure can be expensive and catastrophic. Software errors arising from incorrect programs or from inadequate resources are a significant source of unreliability in embedded systems. Although compile-time program analysis can detect or eliminate some of these errors, others are inherently run-time dependent, in that they appear only with certain inputs or control-flow conditions. In desktop systems, virtual memory [19] is typically used to provide protection against out-of-memory errors and memory access violations. However, many embedded systems lack virtual memory, and although the virtual memory functionality of providing swap space is not generally warranted for use in embedded systems, the protection functions of virtual memory are valuable for their reliable operation. This paper is concerned with protection against memory access violations.

Virtual memory of two distinct types can be used to detect memory access violations. First, *paging* provides memory protection at the level of fixed-size pages, and a memory access violation is detected when a process attempts to reference a page belonging to another process. Second, *segmentation* offers memory protection at the level of variable-size segments within a process, and a violation is detected when the offset of an address is beyond the valid range of the specified segment. Some systems make use of a hybrid technique, known as *paged segmentation*, in which a variable-size segment is composed of an integral number of fixed-size pages.

Unfortunately, and in stark contrast to desktop systems, many commercially available embedded processors do not have support

for virtual memory of any kind and are, therefore, subject to memory errors that can be devastating to their reliability. Examples of such processor families include Motorola's M68K series; Intel's i960; ARM's ARM7TDMI, ARM7TDMI-S, and ARM966E-S; TI's MSP430; Atmel's 8051; Analog Devices' Blackfin; Xilinx's Microblaze; Renesas' M32R; and NEC's NEC750; and many others. Unlike desktop systems where a memory error is often no more than an annoyance, in embedded systems, memory errors can lead to a system failure resulting in the loss of functionality of a controlled system, loss of revenue, industrial accidents, and even loss of life, depending on the type of embedded system. Moreover, the lack of hardware virtual memory protection implies that in such embedded systems, memory errors may not even be detected, causing incorrect functionality, rather than protection faults, to be their only observable effects.

If an embedded system had this detection ability, it would be able to take system-specific remedial actions before a system failure occurred such as the following three; others are possible. First, the system can be shut down safely. For example, if a controller for an industrial process encounters a software error, safely shutting down the assembly line avoids industrial accidents and defective product assembly. Second, a human operator can be hailed to take over manual control of the system. For example, in transportation systems such as aircraft, when a software error occurs in the auto-pilot or a controller for a critical sub-system, it can save lives if the failing controller's tasks are transferred to the pilot. Third, the failing process can be halted, which may result in the preservation of the core functionality of the system. In systems without this detection ability, our work is valuable because it provides, at little cost, a desirable functionality that would not otherwise be present.

In this paper we focus on the problem of segmentation violations. A segmentation violation is a memory access error that occurs when memory is accessed with an address whose offset is beyond the valid range of its associated segment. These types of memory violations typically occur because of programmer error and can manifest themselves in production code when certain external inputs that cause the error to arise are missed during testing. Although several software debugging and profiling packages exist to aid in testing, the assumption that software errors have been eliminated through testing is a dangerous gamble, especially considering that modern embedded systems can contain millions of lines of code, and the number of required tests can be exponential in the number of inputs. Albeit rare in production code, segmentation violations can result in errors whose impact can be catastrophic for systems lacking virtual memory.

Currently, there exists broad categories of both hardware and software methods, aside from virtual memory, that aim to provide differing degrees of memory protection. These techniques include hardware-based fault or domain protection, the use of Memory Management Unit (MMU) permissions capabilities without address translation, safe languages, source-to-source translation, and program instrumentation. Hardware methods typically rely on a portion of virtual memory functionality and are, therefore, subject to the cost associated with its use. However, vendors have decided that the cost of MMUs is often too high in resource-constrained embedded environments [12], noting that the contained segment or page tables and associated logic are all expensive in area, performance, and energy consumption [25]. Moreover, this trend will not diminish with time since virtual memory hardware accounts for a significant portion of a system's total energy consumption and Translation Lookaside Buffer (TLB) misses are detrimental to real-time guarantees. Safe languages usually rely on language restrictions, such as strong type enforcement, to prevent memory access violations. The fault prevention mechanisms these languages feature are certainly preferable to fault detection when the choice of programming language in not an issue; unfortunately, most embedded code is written in C and C++ [29,30]. As an alternative, source-to-source translators and program instrumentation techniques aim to detect memory safety violations at run-time. However, their performance overheads are typically large, making them infeasible for use with embedded systems. Although the degree of protection may differ, our method is distinct from these approaches in that we provide a memory protection solution at low cost and overhead.

The contribution of this work is a segmentation violation detection mechanism for embedded systems lacking virtual memory that is significant for the following five reasons. First, our method is able to provide segment protection without any programming language modifications or semantic restrictions of any kind. Second, our method is applicable for use with a multitude of programming languages, especially those that provide no memory safety guarantees. Third, our method is able to guarantee segment protection without sacrificing system performance. Fourth, our method is completely automatic; the programmer need not even be aware of its presence. For this reason, our technique applies to recompiled legacy code as well. Fifth, our compiler analysis is easily implementable because it is rooted in well-known modern compiler theory. Our implementation is built on the Static Single Assignment (SSA) [2] intermediate representation and data-flow analysis, which is widely available in modern compiler infrastructures.

In order to achieve our goal of software-provided segment protection, run-time checks are inserted before memory accesses to ensure the referenced address is not outside the bounds of its intended segment. However, this requires a method of determining to what segment (*i.e.*, code, globals, stack, or heap) a memory access is referring. For this purpose, we have devised a unified data-flow analysis, called *intended segment analysis*. While providing a means for determining the segment a memory access should be referencing, this analysis is also a powerful framework for recognizing optimizations that are able to eliminate many of the required run-time checks. These include the *dominated reference optimization*, which eliminates run-time checks for multiple memory accesses of the same location; the *non-incremental reference optimization*, which eliminates run-time checks for memory accesses that never occur along control paths involving an arithmetic operation to obtain the referenced address; and the *monotonically addressed range optimization*, which hoists run-time checks out of loops.

Experimental results show that our segment protection solution is able to guarantee segment-level memory safety in all eight tested benchmarks without sacrificing performance. The average increase in run-time, energy consumption, and code size were found to be 0.72%, 0.44%, and 3.60% respectively for the optimized solution. However, one of the eight tested benchmarks suffered from a significant increase in code size, and as a result, is responsible for skewing the average. The average code size overhead of the remaining seven benchmarks was found to be 0.58%. It is also clear from the results that our optimizations used to reduce the number of necessary run-time checks are essential in dramatically lowering the performance overheads. In general, our results validate our claim that software segment protection for embedded systems can be achieved at low cost.

The rest of this paper is organized as follows. Section 2 compares our work with hardware and other software techniques of achieving memory protection. Section 3 describes the intuition of our segment protection solution. Section 4 presents its implementation and discusses issues relating to external functions, separate compilation, and alias analysis. Section 5 describes the evaluation

of our technique and presents performance overheads. Section 6 concludes with a summary of our results and suggests possible enhancements of our work.

2. RELATED WORK

The broad impact of this work is the reproduction in software of a portion of the functionality of virtual memory hardware. In our previous work [4], we addressed the issue of memory overflow protection in embedded systems by using run-time checks and novel techniques for enabling the continued execution of a process after an out-of-memory error had been detected. While our previous work dealt with overflow protection, this work has the different goal of providing segment-level memory protection by detecting memory access violations resulting from programmer error.

Several hardware and hardware assisted approaches exist to provide differing degrees of memory protection. The Mondrian Memory Protection [32] scheme is a hardware approach designed to provide fine-grained memory protection for systems requiring data sharing among processes. This is done by dividing a single address space into multiple domains of protection, with each domain owning a portion of the address space and exporting a set of privileges to other domains. Similarly, a common method [6] of providing hardware memory protection for embedded systems involves using only the permissions capability of the MMU and not virtual-to-physical address translation, which makes memory protection more efficient, affordable, and less complex for programmers. This is not to be confused with software-managed TLBs [28] and software address translation [20], which are two techniques used to give the operating system more control over address translation and are, therefore, unrelated to the notion of protection. Hardware techniques such as these are unappealing for use in embedded systems because, as mentioned earlier, many systems lack the support for such hardware, and even if they did have such support, the increased CPU, memory system resources, and energy consumption associated with its functionality would not be as low as they could be with a software-only solution. Energy consumption is a particular concern since protection hardware is activated for each data and instruction memory access. Because of its frequent use, the energy overhead of hardware memory protection can be considerable compared to an optimizing compiler scheme that expends energy only on the small percentage of memory accesses that require run-time checks. Moreover, real-time guarantees are a concern for systems using TLBs because of the possibility of TLB misses.

Some embedded processors, instead of supporting virtual memory hardware, are equipped with a coprocessor, known as a Memory Protection Unit (MPU) [21], dedicated to lightweight memory access control. The MPU is used to partition the address space into, at most, eight regions varying in size from 4KB to 4GB. Each region is associated with individual access permissions, and the MPU allows access to each region based on the mode (user/privileged) and type (read/write) of the access. The incoming address is compared in parallel with all enabled regions to determine the appropriate access permissions. Despite the MPU's simplistic design and narrow functionality, its use results in increased energy consumption as compared to our software-only memory protection solution since, unlike our method, the MPU is activated for each memory access. Additionally, our method handles an arbitrary number of segments, granting access based on pointer usage, whereas the MPU is restricted to eight protection regions with limited set of permissions. Finally, since each memory access is made more complex with the use of additional hardware, the processor's cycle time may be increased.

One method of providing software memory protection is through the use of safe languages. Cyclone [16] and others [5, 15] provide mechanisms for region-based memory management, allowing for a means of guaranteeing heap safety. In addition to disallowing direct deallocation of dynamically allocated objects in memory regions, these safe languages typically prevent memory access violations through techniques such as language and semantic restrictions, type enforcement, and run-time bounds checking of array accesses. However, a significant hindrance arises when considering the adoptability and compatibility of safe languages. Nearly all embedded code is written in C and C++ [29, 30], and although the prevention capability gained from using a safe language is preferable to detection mechanisms, it is often impractical for industry to port legacy code to a new language.

Control-C [22] is a similar safe language, but relies heavily on semantic restrictions to prove memory safety statically through advanced compiler techniques. These semantic restrictions include, among others, requirements that programs be strongly typed, casts to a pointer type never be from a non-pointer type, pointers be initialized, the address of stack variables never be stored in heap-allocated objects or global variables, and array accesses be affine. In cases where certain restrictions cannot be maintained, this work speculates about the possibility of supplemental safety-checking run-time checks. Control-C has been extended with automatic pool allocation [23], a method of region-based memory management that allows explicit deallocation, in order to provide a mechanism [11] for preventing memory safety violations that result from a dereference of a pointer to freed memory. Our software segment protection scheme is distinct from this work for the following four reasons. First, our method generally provides memory protection at a finer granularity. The Control-C work defines a process to be memory safe if it never references a memory location outside its allocated address space nor executes instructions outside its code segment. However, in certain cases (*e.g.*, array accesses), Control-C is able to reason at the granularity of individual memory objects. Our method is concerned with determining the segments *within* the allocated address space of a process a memory access is supposed to reference and is able to reason about individual memory objects of statically known sizes. Second, while having similar goals, our method does not aim to prove memory safety using exclusively static means; instead, it uses optimized run-time checks when the safety of a memory reference depends on run-time conditions. Although their work mentions the possibility of run-time checks, the exact nature of these checks and their associated overheads are not presented. Third, our method does not depend on region-based memory allocation to prevent a dereference of a pointer to freed memory, and instead, does not optimize run-time checks for pointers that alias analysis reveals may reference a freed memory object. Fourth, our method requires no semantic restrictions, whereas Control-C depends on several restrictions that may be a significant hindrance when programming or porting legacy code.

Software-enforced fault isolation [31] is a memory protection mechanism for tightly-coupled software modules sharing the same address space. An application's virtual address space is divided into protection domains consisting of two segments (code and data) aligned so that all virtual addresses within a segment share a unique set of upper bits, known as the segment identifier. One technique, known as *segment-matching* detects errors by inserting run-time checks before memory accesses to compare the segment identifier of the referenced virtual address with the segment identifier associated with a process's valid address range. Alternatively, *sandboxing* inserts code that, instead of performing checks, actually sets the segment identifier of the referenced virtual address to be that of the

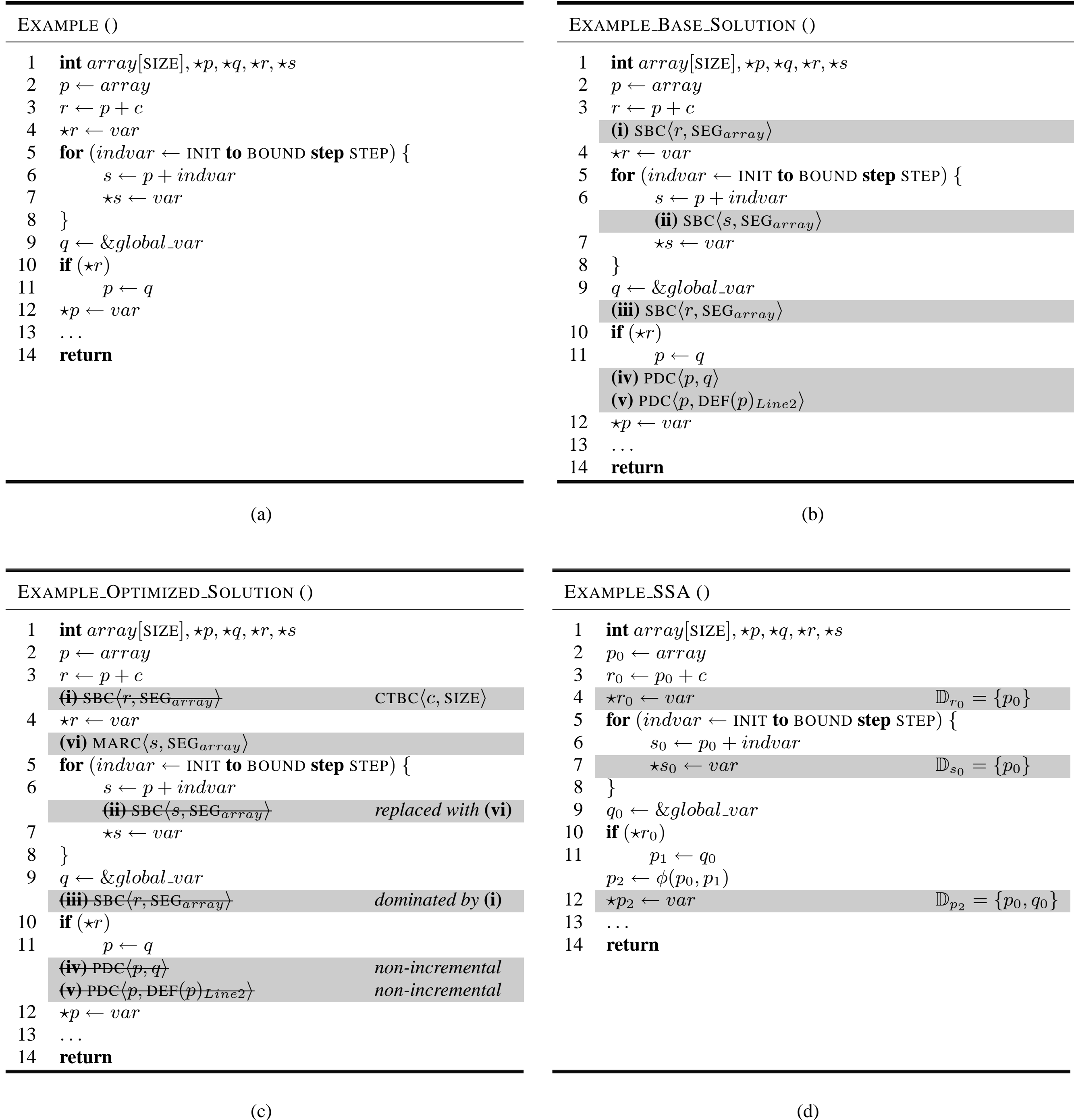


Figure 1: An Example Requiring Run-time Checks. The original code fragment (a) is shown after run-time checks have been inserted using the intuition of the base solution (b) and with optimizations (c). The code is also shown in SSA form (d) along with the set of terminating definitions associated with each pointer dereference. Refer to Figures 2 and 3 for a description of each check and Table 1 for the formulation of each set of terminating definitions.

valid address range. Our method is distinct from these fault isolation techniques for the following three reasons. First, these techniques rely on a virtual address space provided by hardware virtual memory, the need of which our method aims to preclude. While one could envision a system employing these techniques without requiring a virtual address space, it would be inefficient and impractical to partition a physical address space because the use of a fixed number of high-order bits for the segment identifier implies that the physical segments must all be of equal size. (This is not the case for virtual segments since the segment table specifies the size of each segment). The use of equal size physical segments can result in exceedingly wasteful memory usage since the segments can

Segment Bounds Check $\langle p_x, \text{SEG}\rangle$
1 **if** $(p_x < \text{SEG_LOW}$ **or** $p_x > \text{SEG_HIGH})$ {
2 SIGNAL_SEGMENTATION_VIOLATION()
3 }
4 **dereference** p_x

(a)

Pointer Disambiguation Check $\langle p_x, p_y\rangle$
1 **if** $(p_x == p_y)$ {
2 $\text{SBC}\langle p_x, \text{SEG}(p_y)\rangle$
3 }
4 **dereference** p_x

(b)

Figure 2: Segmentation Violation Checks for Base Solution. Segment Bounds Check (SBC) (a): A segmentation violation is signaled if the address refered to by p_x is outside the bounds of its intended segment SEG. Pointer Disambiguation Check (PDC) (b): A segment bounds check is performed if the addresses refered to by p_x and p_y are equal. Refer to Figure 3 for checks required by the optimized solution.

Compile-time Bounds Check $\langle c, \text{SIZE}\rangle$
1 **if** $(c < 0$ **or** $c > \text{SIZE})$ {
2 SIGNAL_COMPILATION_ERROR()
3 }
4 **continue compilation**

(a)

Monotonically Addressed Range Check $\langle p_x, \text{SEG}\rangle$
1 $min_{p_x} \leftarrow p_{x_init} + \text{LOOP-INVARIANT}$
2 $iterations \leftarrow \lceil(\text{BOUND} - \text{INIT})/\text{STEP})\rceil$
3 $max_{p_x} \leftarrow min_{p_x} + iterations \times \text{STEP}$
4 **if** $(min_{p_x} < \text{SEG_LOW}$ **or** $max_{p_x} > \text{SEG_HIGH})$ {
5 SIGNAL_SEGMENTATION_VIOLATION()
6 }
7 **for** $(indvar \leftarrow \text{INIT}$ **to** BOUND **step** STEP$)$
8 **dereference** p_x

(b)

Figure 3: Additional Segmentation Violation Checks for Optimized Solution. Compile-time Bounds Check (CTBC) (a): A compilation error is signaled if constant c is outside the bounds of the memory object whose size is represented by SIZE. Monotonically Addressed Range Check (MARC) (b): A segmentation violation is signaled if the minimum or maximum addresses refered to by p_x within a loop are outside the bounds of its intended segment SEG.

have vastly differing space requirements. Second, as mentioned earlier, our method is concerned with the protection granularity of segments within a process, whereas these techniques treat the entirety of a process's data as a single segment. Third, our method extends the capability of a detection mechanism, such as segment-matching, by discussing the possibility of remedial action.

CCured [9] and Safe-C [3] are well-established source-to-source translators that analyze C programs and insert source-expressible run-time checks to prevent unsafe code from executing. CCured ensures the type safety of C programs by combining a run-time type-inferencing algorithm with checks for pointer dereferences based on their dynamically determined usage type. While being complimentary to our solution, CCured, and other source-to-source translators are distinct from our segment protection scheme for the following three reasons. First, CCured is more restrictive than our method because its goal is to provide memory protection by guaranteeing type safety. Our method is not concerned with types, and provides memory protection by inserting checks to ensure memory accesses are within the bounds of their intended segment. Second, the amount of run-time checks associated with source-to-source translators is typically enormous, requiring far too much overhead for use in an embedded system because they lack many sophisticated compiler optimizations used to significantly reduce the number of checks. In fact, CCured is documented as having a run-time overhead of as much as 60%, with an average of about 30%. Our system exploits compiler analyses to eliminate nearly all run-time checks. Third, CCured cannot perform its translation automatically for all programs. Indeed, CCured requires user assistance when it is unable to infer type information, a common occurrence for medium-sized to large-sized programs.

Valgrind [26], Purify [18], and recently Mudflap [13] are well-known software debugging tools used for preventing pointer use errors. Valgrind is a virtual machine that simulates the execution of unmodified applications, checking the processor operations for validity. Purify is a proprietary software package that instruments object files by replacing compiled pointer operations with calls to a run-time library that ensures the memory safety of these operations. Similarly, Mudflap maintains a database of every memory object in the program that is used in conjunction with calls to a run-time library to verify the validity of potentially unsafe pointer dereferences. However, in concordance with the typical nature of software debugging and testing, these tools incur large application overheads. For example, Mudflap reports an overall application slowdown of a factor of greater than three for two of three evaluated benchmarks. As such, unlike our method, these tools are not intended for continued deployment in embedded systems.

3. SEGMENT PROTECTION SOLUTION

In order to provide a segment protection using software, the intended segment a memory reference is supposed to access must be determined statically or at run-time. It is important to note that while the term *intended segment* is used to denote the supposition that the segment accessed by a memory reference can be deemed incorrect, it is distinct from the notion of *programmer intent*. Indeed, our method does not aim to discover the will of the programmer; its goal is to determine the segment that, without relying on language characteristics or memory organization assumptions, a memory reference is intended to access as it is defined in the program. For example, in ANSI C, no guarantee is provided about the relative ordering of variables or segments. If pointer arithmetic is

Case	Type	$\mathbb{D}_{p_n}$ = **Def**(p_n)	$\mathbb{R}_{p_n}$ = **RT-Check**(p_n)
1	$p_n \leftarrow \&a$	$\{p_n\}$	$\{\}$
2(a)	$p_n \leftarrow \&a + c$	$\{p_n\}$	$\{\}$
(b)	$p_n \leftarrow \&a + var$	$\{p_n\}$	$\{\text{SBC}\langle ptr, \text{SEG}_a\rangle\}$
(c)	$p_n \leftarrow \&a + ind$	$\{p_n\}$	$\{\text{MARC}\langle ptr, \text{SEG}_a\rangle\}$
3(a)	$p_n \leftarrow q$	$\text{DEF}(q)$	$\text{RT-CHECK}(q)$
(b)	$p_n \leftarrow q_{ind}$	$\text{DEF}(q_{ind})$	$\{\text{MARC}\langle ptr, \text{SEG}(q_{ind})\rangle\}$
4(a)	$p_n \leftarrow q + c$	$\text{DEF}(q)$	$\text{RT-CHECK}(q)$
(b)	$p_n \leftarrow q_{ind} + c$	$\text{DEF}(q_{ind})$	$\{\text{MARC}\langle ptr, \text{SEG}(q_{ind})\rangle\}$
(c)	$p_n \leftarrow q + var$	$\text{DEF}(q)$	$\{\text{SBC}\langle ptr, \text{SEG}(q)\rangle\}$
(d)	$p_n \leftarrow q + ind$	$\text{DEF}(q)$	$\begin{cases} \{\text{MARC}\langle ptr, \text{SEG}(q)\rangle\} : \lvert\mathbb{R}_q\rvert = 0 \\ \text{RT-CHECK}(q) : \textbf{otherwise} \end{cases}$
5	$p_n \leftarrow$ *func()*	$\text{DEF}(retvar)$	$\text{RT-CHECK}(retvar)$
6	$p_n \leftarrow$ *malloc()*	$\{p_n\}$	$\{\}$
7	$p_n \leftarrow arg$	$\{p_n\}$	$\{\text{SBC}\langle ptr, \text{SEG}(arg)\rangle\}$
8	$p_n \leftarrow c$	$\{p_n\}$	$\begin{cases} \{\} : c \text{ is a memory-mapped I/O location} \\ \{\text{SBC}\langle ptr, \text{DATA}\rangle\} : \textbf{otherwise} \end{cases}$
9	$p_n \leftarrow \star q$	$\bigcup_{\forall p_i (q \leftarrow \&p_i)} \text{DEF}(p_i)$	$\begin{cases} \{\text{SBC}\langle ptr, \text{S}\rangle\} : \exists \text{S} \forall p \in \mathbb{D}_{p_n} (\lvert\mathbb{R}_p\rvert > 0 \rightarrow \text{SEG}(p) = \text{S}) \\ \{\forall p \in \mathbb{D}_{p_n} (\text{PDC}\langle ptr, p\rangle \mid \lvert\mathbb{R}_p\rvert > 0)\} : \textbf{otherwise} \end{cases}$
10	$p_n \leftarrow \phi(\mathbb{A})$	$\bigcup_{\forall p_i \in \mathbb{A} (p_n \notin \mathbb{D}_{p_i})} \text{DEF}(p_i)$	$\begin{cases} \{\text{SBC}\langle ptr, \text{S}\rangle\} : \exists \text{S} \forall p \in \mathbb{D}_{p_n} (\lvert\mathbb{R}_p\rvert > 0 \rightarrow \text{SEG}(p) = \text{S}) \\ \{\forall p \in \mathbb{D}_{p_n} (\text{PDC}\langle ptr, p\rangle \mid \lvert\mathbb{R}_p\rvert > 0)\} : \textbf{otherwise} \end{cases}$

Table 1: Pointer Definition Types and Run-time Checks. Each pointer definition generates two sets: $\mathbb{D}_{p_n}$ is the set of terminating definitions of p_n and $\mathbb{R}_{p_n}$ is the set of run-time checks required for a safe dereference of p_n. For each definition, p and q are pointers, ptr represents the dereference address, a and var are run-time variables, ind is a non-pointer loop induction variable, q_{ind} is a pointer loop induction variable, and c is a compile time constant. For definitions involving functions, *malloc()* refers to any dynamic memory allocation function and *func()* refers to every other function returning a pointer value. arg refers to the actual argument of a function, and $retvar$ is used to denote the returned pointer of *func()*, which is defined as the ϕ-function of the pointers associated with each return statement. For definitions envolving a ϕ-function, $\mathbb{A}$ is a set containing its arguments. DATA refers to the entire data space allocated to a process, SEG_a refers to the segment in which memory object a resides, and similarly, $\text{SEG}(q)$ refers to the intended segment of pointer q. Refer to Figures 2 and 3 for a description of each check and Table 2 for compile-time checks associated with each definition.

used to reference a memory location, it is clear that the referenced segment is intended to be the same before and after the arithmetic.

Determining the intended segment of non-pointer accesses is a trivial task since the intended segment is simply the segment in which each memory object resides. However, pointer dereferences prove to be more complicated because each pointer has the potential to point to several memory locations, each of which possibly resides in a separate segment. For some of these, it may only be possible to statically determine a *set* of memory locations that are referenced, rather than only one. Although a rare occurrence, the intended segment of these *ambiguous-segment* pointers cannot be determined statically, and code must be inserted to determine the segment at run-time. In this way, one segment can be determined for every memory reference.

For the following discussion of our segment protection solution, the example code fragments shown in Figure 1 will be frequently referenced in order to demonstrate the intuition supporting the required run-time checks and optimizations.

3.1 Base Solution Intuition

In order to guarantee the detection of every possible segmentation violation, a run-time check is required for *each* memory reference, including instruction fetches, scalar variable and array references, and pointer dereferences, to determine if its effective address is within the bounds of the segment it is supposed to access. However, we recognize that checking every memory reference is trivially unnecessary in the following three cases. First, each sequential instruction access does not need to be checked since, as long as instruction fetches do not continue past the end of the code, execution will always remain within the bounds of the segment. Second, a branch instruction with a PC-relative displacement (*i.e.*, a branch whose target is the program counter plus a constant operand of the branch instruction) can be statically determined to be within the code segment bounds since the displacement value is a compile-time constant. Third, a scalar variable reference does not require a run-time check since its address is also a compile-time constant and can be statically determined to be within the bounds of the segment in which the variable resides.

Figure 2 describes the run-time checks necessary to achieve our base solution. Before every memory reference requiring a check, a *segment bounds check* (SBC), shown in Figure 2(a), is inserted to verify that the value of the pointer being dereferenced is within the bounds of its intended segment. For simplicity, when describing checks, no distinction will be made between array references and pointer dereferences since both can be expressed in the same fashion. In case the dereferenced pointer is an ambiguous-segment

Case	Type	$\mathbb{D}_{p_n}$ = **Def(p_n)**	$\mathbb{C}_{p_n}$ = **CT-Check(p_n)**
2(a)	$p_n \leftarrow \&a + c$	$\{p_n\}$	$\{\text{CTBC}\langle c, \text{SIZE}_a\rangle\}$
3(a)	$p_n \leftarrow q$	$\text{DEF}(q)$	$\text{CT-CHECK}(q)$
4(a)	$p_n \leftarrow q + c$	$\text{DEF}(q)$	$\begin{cases} \{\text{CTBC}\langle c+k, \text{S}\rangle\} : (\mathbb{C}_q = \{\text{CTBC}\langle k, \text{S}\rangle\}) \wedge (\lvert\mathbb{R}_q\rvert = 0) \\ \{\text{CTBC}\langle c, \text{SIZE}(q)\rangle\} : (\lvert\mathbb{C}_q\rvert = 0) \wedge (\lvert\mathbb{R}_q\rvert = 0) \\ \{\} : \textbf{otherwise} \end{cases}$
5	$p_n \leftarrow$ *func()*	$\text{DEF}(retvar)$	$\text{CT-CHECK}(retvar)$

Table 2: Pointer Definition Types and Compile-time Checks. In addition to a set of terminating definitions $\mathbb{D}_{p_n}$ and a set of run-time checks $\mathbb{R}_{p_n}$, each pointer definition generates a third set $\mathbb{C}_{p_n}$, which is the set of compile-time checks that may be performed. c and k are both compile-time constants and SIZE_a refers to the size of memory object a in bytes divided by the number of bytes associated with a single element of the type of a. Similarly, $\text{SIZE}(q)$ refers to the SIZE of the intended memory object of q. Refer to Figure 3 for a description of the compile-time check and Table 1 for other terminology and run-time checks associated with each definition. For definition types not listed in this table, the generated set of compile-time checks is the empty set.

pointer, a *pointer disambiguation check* (PDC), shown in Figure 2(b), is inserted for each definition of the pointer that reaches the dereference. This run-time check performs a segment bounds check if the dereferenced pointer value is the same as that of one of its previous definitions. In this way, for each set of pointer disambiguation checks inserted, exactly one will result in the execution of a segment bounds check.

In the running example, the necessary run-time checks for the code fragment in Figure 1(a) are presented in Figure 1(b). A segment bounds check is required before every pointer dereference except the dereference of p on line 12, which requires a pointer disambiguation check since multiple definitions reach its dereference. Note that the lines containing checks are *not* function calls; they represent the actual code for the checks as defined in Figure 2.

3.2 Optimizations

Because the large number of run-time checks required to guarantee segment protection by the base solution can induce prohibitively expensive performance overheads, we have devised the following three compiler optimizations that can dramatically reduce the number of required run-time checks needed to guarantee segment protection. Two of these optimizations rely on additional safety checks, shown in Figure 3, that are used in place of the original ones.

Dominated Reference Optimization Multiple references to the same address require a check only for the reference that dominates the others; subsequent references do not need a check. Well-known in compiler theory, a reference r_1 *dominates* [2] another reference r_2 if every path from the beginning of the program to r_2 includes r_1. Such a case is easily detected at compile-time when the address expression is a single scalar variable. For more complex address expressions, the widely available compiler optimization Common Subexpression Elimination (CSE) [7] reduces the address expression to a repeated single scalar temporary variable, allowing for the trivial detection of its repeated use. Figure 1(c) shows the example code with optimized checks. After this optimization, the dereference of r at line 10 is dominated by its dereference at line 4 so check (iii) is not needed.

Non-incremental Reference Optimization Memory references that are never reached via a data-flow path involving an arithmetic operation to obtain the referenced address do not require a check. For example, if arithmetic is never performed on a pointer, it is guaranteed to continue to point to the location, and therefore the segment, of its initial assignment. Additionally, if a memory reference is reached along a path involving the arithmetic of a compile-time constant, and the memory object referred to is of a statically known size, a compile-time check, instead of a run-time check, can be performed to verify that the referenced address is within the bounds of the memory object. For example, accesses to fields of structures that are of a statically known size can be verified at compile-time to be within the legal bounds of an instance of the structure. This check is known as a *compile-time bounds check* (CTBC) and is shown in Figure 3(a). After applying this optimization to the example code, check (i) can be replaced with a compile-time check since $array$ is of a statically known size, and r is defined with a constant offset. Additionally, since p is never defined with arithmetic, its dereference at line 12 no longer requires checks (iv) and (v).

Monotonically Addressed Range Optimization Memory references whose address is an induction variable plus or minus a loop-invariant quantity (*e.g.*, an affine array access within a loop) that occur within loops whose terminating conditions are also loop-invariant are said to be monotonically addressed. Here, it is sufficient to verify in the loop preheader that the minimum and maximum referenced addresses are within the segment bounds. This run-time check is known as the *monotonically addressed range check* (MARC) and is shown in Figure 3(b). Although array-bounds check elimination [2] has been the topic of much research, our optimization does not require the memory reference to be an affine array access. Indeed, a dereferenced pointer could itself be an induction variable. In certain cases, though, techniques such as affine conversion [14] can be used to convert pointer accesses of array elements into semantically equivalent array representations with explicit index expressions. Thereafter, the array access would be subject to bounds check elimination. However, our method does not rely on such techniques. After applying this optimization to the example code, check (ii) can be replaced by (vi), verifying the minimum and maximum values of s are within the bounds of the segment in which $array$ resides.

4. INTENDED SEGMENT ANALYSIS

Our intended segment analysis algorithm has been implemented using a framework based on the Static Single Assignment (SSA) [2] intermediate representation. SSA is a convenient means of representing our analysis for the following reasons. First, each dereferenced pointer has exactly one definition since SSA creates subscripted versions of a variable for each of its assignments. In this way, use-def chains are explicit and each contains a single element. Second, because the representation of control flow is inherent in SSA using ϕ-functions, other complicated data-flow analyses are not required to determine if a particular memory reference war-

Foo ()		Foo_SSA ()	
1	**int** $stack_var, x, \star p$	1	**int** $stack_var, x, \star p$
2	$x \leftarrow global_var$	2	$x_0 \leftarrow global_var_0$
3	$p \leftarrow \&x$	3	$p_0 \leftarrow \&x_0$
4	$x \leftarrow stack_var$	4	$x_1 \leftarrow stack_var_0$
5	$var \leftarrow \star p$	5	$var_0 \leftarrow \star p_0$
6	...	6	...
7	**return**	7	**return**

(a)

Bar ()		Bar_SSA ()	
1	**int** $stack_var, \star p, \star\star q$	1	**int** $stack_var, \star p, \star\star q$
2	$p \leftarrow \&global_var$	2	$p_0 \leftarrow \&global_var_0$
3	$q \leftarrow \&p$	3	$q_0 \leftarrow \&p_0$
4	$p \leftarrow \&stack_var$	4	$p_1 \leftarrow \&stack_var_0$
5	$var \leftarrow \star\star q$	5	$var_0 \leftarrow \star\star q_0$
6	...	6	...
7	**return**	7	**return**

(b)

Figure 4: Example SSA Forms with Indirection. Function *foo()* (a) is shown in SSA form dereferencing pointer p with a single degree of indirection. Function *bar()* (b) is shown in SSA form dereferencing pointer q with multiple degrees of indirection. Refer to Case 9 in Table 1 for the analysis by which the intended segment of q and the run-time checks for the dereference of q are determined.

rants a check. Finally, situations in which our three optimizations are applicable are easily recognizable, thus making an optimized segment protection solution obtainable without great effort. Moreover, it provides a framework in which more optimizations may be easily realized.

4.1 Determining Required Checks

The intended segment of memory references and their required checks are determined as follows. Each pointer definition is recursively analyzed, generating a set of terminating definitions that determine its intended segment and a set each of run-time and compile-time checks that must be performed to guarantee the safety of its dereference. A *terminating definition* is one that does not use another pointer definition, and the set of these definitions for a pointer p is denoted $\mathbb{D}_p$. Similarly, the set of run-time checks required for a dereference of p is denoted $\mathbb{R}_p$ and the set of compile-time checks is denoted $\mathbb{C}_p$. For example, Figure 1(d) shows the original code from Figure 1(a) in SSA form with the set of terminating definitions for each dereference explicitly indicated. Note that for the dereference of p_2, its definition is recursively analyzed backward through ϕ-functions to determine the dereferenced value could be either the value of p_0 or q_0. This recursive chaining property eliminates p_1 from the set since it uses the definition of another pointer. In this way, the cardinality of $\mathbb{D}_{p_2}$ is two, precipitating the need for the pointer disambiguation checks since the intended segment of each element of the set is different.

Tables 1 and 2 provide a detailed description of how the terminating definitions and necessary run-time and compile-time checks, respectively, are determined for a dereference of every pointer definition type; each one will be discussed in turn. The case numbers are the same in both tables, and each table should be referenced frequently to better understand each case.

Cases 1 & 2 If a pointer is assigned the address of a scalar variable, as in Case 1, its dereference does not require a check because of the non-incremental reference optimization. For definitions involving arithmetic, as in Case 2, if the operand is (a) a compile-time constant, the compile-time bounds check shown in Table 2 can be performed instead of a run-time check. If the operand is (b) a run-time variable, a segment bounds check is required to verify the dereferenced addressed is within the variable's segment, and if the operand is (c) an induction variable, a monotonically addressed range check can be performed in the preheader of the loop to verify the minimum and maximum referenced addresses are within the variable's segment. Each definition associated with Cases 1 and 2 is terminating.

Case 3 If a pointer is assigned the value of (a) another pointer, the set of run-time and compile-time checks required for its dereference are those of the copied pointer. In the event that the copied pointer is (b) an induction variable, a monotonically addressed range check can be performed in the preheader of the loop; no compile-time checks can be performed. The pointer's set of terminating definitions is that of the copied pointer.

Case 4 For pointer copies involving arithmetic, if the operand is (a) a compile-time constant, the necessary run-time checks are those of the copied pointer. In the event that the definition of the copied pointer induces no run-time checks, a compile-time bounds check can be performed if the size of the referenced memory object is known statically. If the operand is a compile-time constant and the copied pointer is (b) an induction variable, a monotonically addressed range check can be performed for the dereferenced pointer and its intended segment, but no compile-time check can be performed. If the operand is (c) a run-time variable, a segment bounds check must be performed, and if the operand is (d) an induction variable, the monotonically addressed range check can be performed as long as the definition of the copied pointer induces no other run-time checks (*e.g.*, an already required SBC for the copied pointer invalidates the ability to perform the optimization in this case); no compile-time checks are possible for either. For each definition associated with Cases 4, the set of terminating definitions is that of the copied pointer.

Cases 5, 6 & 7 If a pointer is assigned the return value of a function, as in Case 5, its set of terminating definitions and run-time and compile-time checks are those of the returned pointer. If the function is a dynamic memory allocating function, as in Case 6, the pointer definition is terminating, no checks are required because of the non-incremental reference optimization, and the intended segment of the pointer is known statically to be the heap. Finally, if the dereferenced pointer is an argument of the function containing the dereference, as in Case 7, its definition is also terminating, but a segment bounds check is required to verify that the referenced address is within the argument's intended segment. Due to the interprocedural nature of our analysis, the bounds of the argument's intended segment are actually maintained as additional arguments of the function and, in the case of an ambiguous-segment pointer, defined at each call site of the function.

Case 8 If a pointer is assigned a statically known constant value, its definition is terminating, and a segment bounds check is required to verify the dereferenced address is within the bounds of the entire data segment (the actual segment is unknown statically) of the process; no compile-time check is possible. As a further optimiza-

Benchmark	Category	Input	Input Size (bytes)	Lines	Description
adpcm	telecomm	small.pcm	1711080	741	Adaptive Differential Pulse Code Modulation
basicmath	automotive	none	none	84	Basic Mathematical Operations
blowfish	security	small.asc	311824	1502	SSL Encryption Algorithm
crc32	telecomm	large.pcm	1368864	281	Cyclic Redundancy Checksum
dijkstra	network	input.dat	29144	174	Shortest Path Algorithm
fft	telecomm	4×2048	none	469	Fast Fourier Transform
stringsearch	office	none	none	3216	String Pattern Matching
susan	automotive	small.pgm	7292	2122	Image Processing and Enhancing

Table 3: Benchmark Programs and Characteristics. For each benchmark listed in the first column, the category to which each benchmark pertains is given in the second column, the test input and size of the input in bytes is given in the third and forth columns, the size of each benchmark in lines of code is given in the fifth column, and a description of each benchmark is given in the sixth column.

tion, if the constant value is known to be a memory-mapped I/O location, no check is required. Case 8 is also used to handle uninitialized pointers and pointers assigned the value of `NULL`. For the dereference of these pointers, it is possible to signal a compilation error when it is statically determined that the dereferenced pointer can *only* be unassigned or `NULL`. However, virtual memory provides no protection beyond the described bounds check for this type of dereference, and since our goal is the reproduction of its functionality in software, our solution does not signal such an error. It would be trivial to add this feature even though other production quality compilers, such as GCC, normally do not report such a warning.

Case 9 If a pointer is assigned the dereferenced value of another pointer, such is the case involving multiple degrees indirection, the set of terminating definitions and required checks are as follows. Consider the example code fragments shown in SSA form in Figure 4. In Figure 4(a), for the dereference of p_0, its intended segment is determined to be the segment in which x_0 resides. Although the dereferenced value is clearly not that of x_0 but of x_1, this is not contentious since x_0 and x_1 are guaranteed to be in the same segment. However, this is not the case for multiple degrees of indirection. In Figure 4(b) it is incorrect to conclude that, by recursively analyzing pointer definitions, the intended segment of q_0 is the segment in which $global_var_0$ resides (it is actually the stack). Therefore, the following rules apply. The set of terminating definitions for a pointer p is the union of the set of terminating definitions of each subscripted pointer associated with the one of which p takes the address. For example, in Figure 4(b), $\mathbb{D}_{q_0}$ contains both p_0 and p_1. For the dereference of p, a segment bounds check is required if the intended segment of all the terminating definitions of p that require a run-time check are the same. If the intended segments are different, p is an ambiguous-segment pointer, and a pointer disambiguation check must be inserted involving p and each terminating definition requiring a run-time check. No compile-time checks can be performed.

Case 10 If a pointer is assigned the result of a ϕ-function, its set of terminating definitions is the union of the set of terminating definitions of each argument of the ϕ-function as long as the dereferenced pointer is not in the set of terminating definitions of the argument (*e.g.*, for certain loops, a pointer can be recursively defined in terms of itself with ϕ-functions). The checks required for a dereference of this pointer type are identical to those of Case 9.

4.2 Maintaining Segment Bounds

Segment bounds for the code, globals, stack, and heap segments are maintained at run-time as global variables inserted into the program, and determining their values at run-time is a trivial task. Linker symbols are used to define the base of the code segment, the base of the globals segment, and the base of the heap segment. The heap pointer is made known to the program by modifying the *malloc()* library function so that its value is available externally. The stack pointer is a register value, and the base of the stack is defined at the beginning of *main()* to be a specified offset from the initial frame pointer, which is also a register value.

4.3 External Functions

When a compiler analysis, such as intended segment analysis, is required to analyze an entire program, external functions become an important consideration if separate compilation is to be maintained. We have devised the following two-part solution for handling external functions while still allowing separate compilation.

First, our analysis is performed at link-time. The advantage of link-time analysis is that the entire program is analyzed as a whole without the loss of any information. The problem with this solution, however, is that, even though separate compilation is maintained, performing a complicated analysis on an entire program at once can be time consuming; it would be convenient if the cost of the analysis was distributed evenly among each compilation. As such, we have adopted link-time analysis for every external function intrinsic to the particular application and not library functions it may call. The increase in link-time is tolerable since it need not be incurred during debugging but only during the final production compile. In future work we will investigate extending our segment protection solution to operate in infrastructures lacking a link-time optimization framework or those where the link-time object code format lacks the information required by our analysis.

Second, the limited assumption is made that library functions can be deemed *semi-safe*. That is, if a pointer is passed as an argument to a library function, it is assumed the library will not cause the intended segment of the pointer at its dereference point inside the function to differ from its intended segment before the call. If library functions do not satisfy this property then they are still allowed, but pointer dereferences inside them are not checked. To prevent a memory violation, it is then necessary to insert a segment bounds check before the call and before the next dereference of the pointer after the call returns, in case the pointer may have been freed. Additionally, if a library function that is not a dynamic memory allocation function returns a pointer, a segment bounds check must be inserted before its first dereference to verify the address is within the bounds of the entire data section of the process since the library function is not analyzed to determine the intended segment of the returned pointer. Using these two techniques, our

segment protection solution is able to safely handle external and library functions that are compiled using our method and still maintain separate compilation.

4.4 Comparison with Alias Analysis

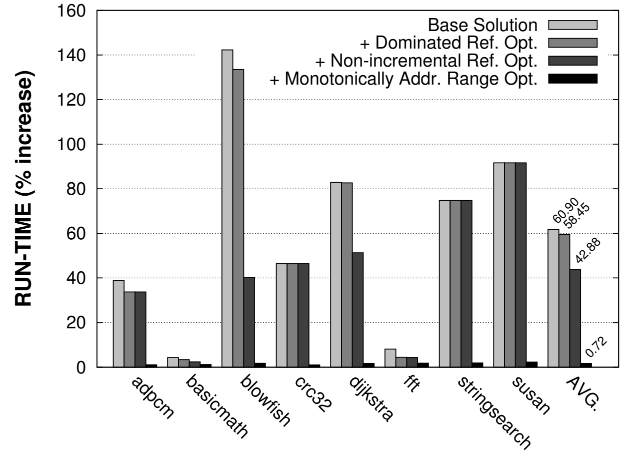
There may be confusion as to how intended segment analysis is distinct from alias analysis [1, 10, 27]. While not requiring alias analysis, our method could be implemented in a such way that it would make use of the resulting information. For example, *may-alias* querying could be used to determine the set of memory objects a pointer could be referencing. The intended segment of the pointer would then simply be the segment in which each object resides. However, this method of determining intended segments makes use of extraneous and unnecessary information. The memory object referenced by the pointer is unimportant, but the segment in which it resides is. In that sense, pointer analysis solves a strictly harder problem than intended segment analysis since the latter reasons at the coarser granularity of segments. Moreover, without the pointer usage information our SSA implementation maintains through the recursive analysis of pointer definitions, such as whether or not a pointer has been incremented, the optimizations essential for reducing the performance overheads of our base solution would not be possible.

Often, the complexity of alias analysis is discussed in terms of flow and context sensitivity. Intended segment analysis is a flow-sensitive data-flow analysis in as much as the control information encapsulated in the ϕ-function of SSA form is flow-sensitive. While being interprocedural, the need for context-sensitivity is avoided by passing intended segment bounds as arguments to functions with pointer arguments. In the case that a pointer argument may be an ambiguous-segment pointer, run-time checks are inserted at each call site of the function to perform the disambiguation and set the appropriate bounds arguments. In this way, context-sensitivity is not needed because the program is instrumented to discover the intended segment at run-time. Because our implementation analyzes a set of definitions per pointer dereference without requiring context sensitivity, its algorithm complexity is linear.

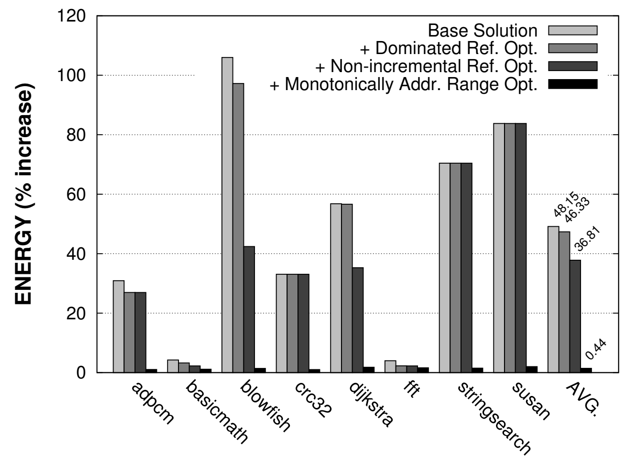
5. RESULTS

Our segment protection solution for embedded systems, as presented, has been implemented in the GCC-based Low Level Virtual Machine (LLVM) [24] research compiler infrastructure and link-time optimization framework. As mentioned in Section 4.3, link-time optimization has the benefit of allowing a mechanism for whole-program interprocedural analysis even when procedures may be defined externally. Since LLVM currently does not yet fully support an embedded target, its C code backend was used to generate optimized source code including our run-time checks, and the instrumented program was then compiled with no optimizations using the production-level, public domain GCC cross-compiler [8] targeting the ARM Version 5 instruction set architecture. The programs were then simulated with the public domain, cycle accurate GDB [8] simulator and debugger. For consistency, each of the eight evaluated benchmarks was compiled using this infrastructure to obtain a base performance before measuring the overheads of our method. These benchmarks are from the MIBench [17] suite of embedded benchmarks and presented in Table 3.

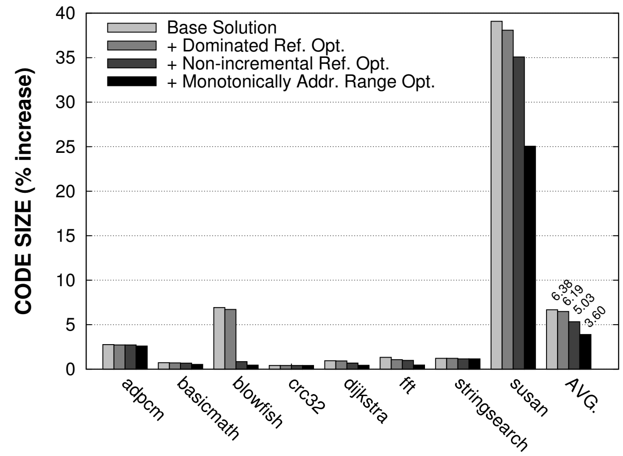
The performance overheads of our segment protection solution, given as a percent increase of each original unmodified benchmark, are presented in Figure 5. The run-time overheads are shown in Figure 5(a) and were found to average 60.90% for the unoptimized base solution and 0.72% for the optimized solution. Similarly, the energy consumption overheads are shown in Figure 5(b) and were



(a)



(b)



(c)

Figure 5: Overheads for Run-time Checks. For each benchmark, the performance is given as a percent increase in run-time (a), energy consumption (b), and code size (c) as measured against the original, unmodified benchmark. The last group in each histogram contains the averages over all the benchmarks.

found to average 48.15% for the base solution and 0.44% for the optimized solution. The optimizations provide a dramatic improve-

Benchmark	Static Checks by Check Type						Static Checks by Dereference Type					
	Unoptimized			Optimized			Unoptimized			Optimized		
	SBC	MARC	PDC	SBC	MARC	PDC	POINTER	ARRAY	STRUCT	POINTER	ARRAY	STRUCT
adpcm	11	0	0	1	8	0	3	8	0	1	8	0
basicmath	9	0	0	2	5	0	2	5	2	2	5	0
blowfish	159	0	0	0	7	0	23	48	88	0	7	0
crc32	3	0	0	0	2	0	2	1	0	1	1	0
dijkstra	25	0	0	1	11	0	4	12	9	0	12	0
fft	34	0	0	0	21	0	34	0	0	21	0	0
stringsearch	13	0	0	0	12	0	4	9	0	4	8	0
susan	937	0	0	587	230	0	594	337	6	579	248	0

Table 4: Number of Statically Inserted Checks. For each benchmark listed in the first column, the number of run-time checks inserted is given, separated by check type and program dereference type for the unoptimized base solution and the fully optimized solution. Refer to Figures 2 and 3 for a description of each run-time check.

ment in performance, especially for *blowfish*, in which the run-time overhead was decreased by over 140%. The code size overheads are shown in Figure 5(c) and were found to average 6.38% for the base solution and 3.60% for the optimized solution. Without including *susan*, the remaining seven benchmarks average a code size increase of 0.58% for the optimized solution. *susan* is interesting because while many run-time checks were inserted to guarantee segment protection, significantly increasing code size, its run-time remained low. This is because checks were inserted for memory references that were rarely or never executed.

As determined from the plots in Figure 5, it is clear that the monotonically addressed range optimization is the most essential optimization used in reducing the performance overheads. Embedded benchmarks are largely loop-intensive and spend the majority of their execution time iterating over the same instructions. Inserting a segment bounds check within a loop is enormously costly in these situations, and it is reasonable to conclude that hoisting the check out of the loop would cause such a dramatic decrease in the performance overheads. The non-incremental reference optimization was reasonably effective, and the dominated dereference optimization, only aiming to eliminate checks for repeated memory references, provided a significant performance improvement for only a few benchmarks. Therefore, it can be concluded that the evaluated benchmarks rarely sequentially reference the same memory location, or if they do, existing optimizations, such as allocating variables to registers or CSE, remove the repeated references.

Finally, the number of statically inserted run-time checks required to guarantee segment protection are presented in Table 4. For each benchmark, the number of inserted checks are separated by the type of check and also by the type of referenced program constructs (*i.e.*, array, struct, or pointer dereference) requiring a check. It is important to note that these numbers do not represent the number of checks that were dynamically executed. In fact, because the optimized run-time of *susan* is so low, it is likely that only a few of the many checks remaining were ever executed. For the optimized solution, there is a strong correlation in the number of inserted monotonically addressed range checks and array accesses. It is the case that, for the benchmarks showing this correlation, the arrays were being traversed within a loop, yielding themselves subject to the optimization. While it is likely that there were some dereferenced pointers that could point to more than one memory object, for each evaluated benchmark, none of these pointers could refer to more than one segment. Therefore, and in keeping with our contention that ambiguous-segment pointers are rare occurrences in embedded applications, none of the benchmarks required the insertion of a pointer disambiguation check.

6. CONCLUSION AND FUTURE WORK

This paper presents a comprehensive segment-level memory protection solution for embedded systems whose goal it is to improve system reliability without the addition of virtual memory hardware. This is done completely automatically by inserting run-time checks for memory accesses without the need for any programming language or semantic restrictions. Known as intended segment analysis, the easily implementable solution is rooted in the widely available SSA intermediate representation and allows the realization of three optimizations used to reduce the number of required run-time checks. These optimizations include the dominated reference optimization, the non-incremental reference optimization, and the monotonically addressed range optimization. Results show that these optimizations are effective at reducing the performance overheads associated with providing software segment protection to low, and in many cases, negligible levels. For eight evaluated embedded benchmarks, the average increase in run-time was 0.72%, the average increase in energy consumption was 0.44%, and the average increase in code size was 3.60%. In future work we wish to extend our current implementation to better handle library and variadic functions and investigate further optimizations for deeply nested memory references and recursive functions.

7. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.

[2] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, 1994.

[4] S. Biswas, M. Simpson, and R. Barua. Memory overflow protection for embedded systems using run-time checks, reuse and compression. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 280–291, 2004.

[5] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management

in Real-time Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 324–337, 2003.

[6] J. Carbone. Efficient memory protection for embedded systems. *RTC Magazine*, September 2004. http://www.rtcmagazine.com/home/article.php?id=100120.

[7] J. Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, 1970.

[8] CodeSourcery, LLC. *GNU ARM Toolchain*. http://www.codesourcery.com/.

[9] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 232–244, 2003.

[10] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, 2000.

[11] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *Transactions on Embedded Computing Systems (TECS)*, 4(1):73–111, 2005.

[12] M. Durrant. Running Linux on low cost, low power MMU-less processors, August 2000. http://www.linuxdevices.com/articles/AT6245686197.html.

[13] F. C. Eigler. Mudflap: Pointer use cheking for C/C++. In *Proceedings of the GCC Developers Summit 2003*, pages 57–70, 2003.

[14] B. Franke and M. O'boyle. Array recovery and high-level transformations for DSP applications. *Transactions on Embedded Computing Systems (TECS)*, 2(2):132–162, 2003.

[15] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 313–323, 1998.

[16] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Workshop on Workload Characterization*, 2001.

[18] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Technical Conference*, pages 205–215, 1992.

[19] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[20] B. L. Jacob and T. N. Mudge. Uniprocessor virtual memory without TLBs. *IEEE Transactions on Computers*, 50(5):482–499, May 2001.

[21] D. Jagger and D. Seal. *ARM Architecture Reference Manual*. Addison Wesley, 2000.

[22] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 288–297, 2002.

[23] C. Lattner and V. Adve. Automatic pool allocation for disjoint data structures. In *Proceedings of the Workshop on Memory System Performance (MSP)*, pages 13–24, 2002.

[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (GCO)*, pages 75–87, 2004.

[25] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *Transactions on Design Automation Electronic Systems*, 6(2):149–206, 2001.

[26] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Technical Conference*, pages 17–30, 2005.

[27] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[28] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design tradeoffs for software-managed TLBs. *Transactions on Computer Systems (TOCS)*, 12(3):175–205, 1994.

[29] Venture Development Corporation. *The Embedded Software Strategic Market Intelligence Program 2002/2003 Volume 2*, 2003. http://www.vdc-corp.com/embedded/white/03/03esdtvol2.pdf.

[30] Venture Development Corporation. *The Embedded Software Strategic Market Intelligence Program 2004 Volume 1*, 2004. http://www.vdc-corp.com/embedded/white/04/04esdtvol1.pdf.

[31] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[32] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, 2002.