Architectural Support for Low Overhead Detection of Memory Violations

Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, Corey Waxman
State University of New York, Binghamton, NY 13902
aneesh@binghamton.edu

Abstract

Violations in memory references cause tremendous loss of productivity, catastrophic mission failures, loss of privacy and security, and much more. Software mechanisms to detect memory violations have high false positive and negative rates or huge performance overhead. This paper proposes architectural support to detect memory reference violations in inherently unsafe languages such as C and C++. In this approach, the ISA is extended to include "safety" instructions that provide compile-time information on pointers and objects. The microarchitecture is extended to efficiently execute the safety instructions. We explore optimizations, such as delayed violation detection and stack-based handling of local pointers, to reduce the performance overhead. Our experiments show that the synergy between hardware and software results in this approach having less than 5% average performance overhead, while an exclusively software mechanism incurs 480% impact for the same benchmarks.

1 Introduction

Software bugs often result in high rates of computer system failures [24, 27] and make computer systems vulnerable to security attacks [2, 20, 25]. Development of effective means preventing bugs from entering the software design process is highly unlikely [19]. Hence, it is of utmost importance to develop mechanisms to facilitate automatic detection of software bugs and vulnerabilities.

Static detection techniques are computationally infeasible, and have limited usability in the detection of software errors [23]. Dynamic detection mechanisms have relatively higher detection accuracies, but they still miss input-dependent software errors. Dynamic mechanisms with extremely low performance overhead are required so that checks can be performed for a large set of inputs. Low performance overhead of the mechanisms can even result in their usability in deployed production code for post-deployment maintenance of software.

Violations in memory references are common in inherently unsafe languages like C and C++ because pointers are subjected to very few restrictions. Memory violations are the most difficult to track and among the easiest to exploit [36]. This paper proposes *SafeProc* – a processor that provides architectural support for detecting memory bounds violations, dangling pointers, and multiple deletions in array and pointer references. Bounds violations occur when accesses to objects fall outside the boundaries, and are a major source for system failures and exploits. Dangling pointers are pointers to invalid/deallocated memory objects, and primarily result in system failures. Multiple deletions occur when a deallocated memory object is deallocated again, and mostly cause program failures, but can also allow unauthorized execution of arbitrary code [26].

Previous dynamic detection approaches, discussed in Section 5, to detect these violations are exclusively software mechanisms that have a high performance overhead. For instance, the mechanism in [30] makes applications as much as 12X slower. The large performance overhead of these mechanisms is due to the execution of a large number of additional instructions that access huge data structures storing metadata for the applications. This results in long stalls in the application's execution and extensive pollution of the caches. The coprocessor approach [3] to avoid the limitations of software has a considerable increase in the off-chip traffic.

SafeProc, on the other hand, uses a small fraction of the processor transistors (our studies show less than 0.1% for a billion transistor processor) to perform the operations required for memory violations detection. With the large number of transistors available on modern processor chips, using a small fraction of that transistor budget to support memory safety is a viable and an attractive option. The performance impact of SafeProc is low because the original application is not stalled while the checks are performed, checks are performed with considerably fewer specialized instructions executing on specialized hardware, uppermost level of cache is not polluted, and off-chip traffic is restricted.

We propose ISA extensions to include "safety" instructions, extensions to the microarchitecture to efficiently execute these instructions, compilation methodology for safety instructions, and optimizations to reduce the performance impact of our approach. The hardware-software synergy keeps the overhead of the proposed mechanism low – less than 5% with optimizations and about 93% with the base implementation. Our experiments with a previous software mechanism [22] resulted in about 480% performance impact for the same benchmarks.

2 The Design of SafeProc

SafeProc design is based on the property of ANSI C that any memory access that originates from within the bounds of an object must remain within those bounds after any arithmetic on the pointer. The premise of this property is that programmers can manage the internal memory space of an object in any manner, if the object's memory organization is known to them.

This property has been used by most of the proposed software mechanisms, which maintain information on the pointers and the size of the associated objects. SafeProc, on the other hand, provides architectural support to maintain such pointer-object associa-

978-3-9810801-5-5/DATE09 © 2009 EDAA

tion records (henceforth called *POA records*). A pointer is identified by its memory address and an object by the starting and the ending addresses of its memory space. The pointer address for an array is the address of the first element in the array. For instance, if pointer *ptr* (at memory address X) points to an object with start address (SA) and end address (EA), then POA record *X : {SA, EA}* is maintained by SafeProc. Storing the end address of an object, instead of its size, to mark the object boundaries has a higher memory overhead but simplifies the memory management of the POA records, as discussed in Section 2.4. To also simplify the microarchitectural extensions, discussed in Section 2.3, a POA record is assigned for each distinct pointer. This may result in duplicate object bounds in POA records of different pointers pointing to the same object.

2.1 SafeProc Architecture – ISA Extensions

We extend a Reduced Instruction Set Architecture to include "safety" instructions with specialized opcodes, which are used to detect violations in memory references.

Safety Instructions	Operations	Violation Detected
crc R_s, immediate	PA = *R_s + immediate*; Create record for PA	None
drc R_s, immediate	PA = *R_s + immediate*; Delete PA's record	None
dob R_s, immediate	SA = *R_s + immediate*; Invalidate objects with SA	Multiple deletions
mtr R_s, R_t, immediate	PA = *R_s + immediate*; Copy *R_t* and *$(R_t + 1)$* into object bounds of PA's record	None
mfr R_s, R_t, immediate	PA = *R_s + immediate*; Copy object bounds of PA's record into *R_t* and *$(R_t + 1)$*	None
check R_s, R_t, immediate	PA = *R_s + immediate*; Check *R_t* lies within object bounds of PA's record	Dangling pointer, bounds violation

Table 1: Summary of safety instructions; PA = Pointer Address; SA = Start Address

The basic operations required are creating, updating, checking, and deleting the POA records, defined in Table 1. Displacement addressing mode is used for safety instructions because a pointer address is generated by adding an offset to the frame pointer, the global pointer, or a register storing another pointer value. The c*rc* instruction invalidates the object in the created POA record, i.e. a new pointer is not associated with any object. This is done because uninitialized pointers may be used along some control paths in the program. Additional instructions may be added to the ISA to optimize the compiled code. We refrain from doing that to limit the amount of ISA extension.

The *mtr* and *mfr* are defined to initialize and update the POA records. The *mtr* instruction places the contents of registers, which should contain the starting and ending addresses for an object, into a POA record. Similarly, *mfr* loads the object bounds in a POA record into registers. The *mfr* instruction facilitates the pointer copy operations such as *A = B*, where A and B are pointers. B is used in an *mfr* instruction, and A in an ensuing *mtr* instruction to update A's POA record. The *dob* instruction invalidates all objects whose start address matches that generated by *R_s + immediate*. The *check* instruction validates that the memory address stored in *R_t* lies within the object bounds in the associated POA record.

The safety instructions detect the violations when they actually take effect. For instance, a violation is not reported when a pointer points outside the associated object's bounds, but rather when that pointer is used. Studies [30] have shown this relaxed violation detection approach removes the false positives.

2.2 Compilation for SafeProc

The safety instructions are inserted by the compiler during compilation. Static compiler analyses may be performed to optimize the number of pointers that are dynamically tracked for violations [5]. However, in this paper, we experiment with a basic implementation that checks all the pointers. When a pointer is declared (*e.g.* int * ptr), its record is created using *crc*, and when it is deleted, its record is deleted using *drc*. For instance, upon exit from a procedure, all local pointers' records are deleted. *Mtr* and *mfr* instructions are inserted when pointers are assigned values, and *check* instructions are inserted when the pointers are dereferenced. D*ob* instructions are inserted for all deallocated heap objects and for local objects that may be referenced from outside the procedures containing them, e.g. a local object being accessed by a global pointer. Memory not accessed through pointers is not checked. For instance, access to an element of a struct using <*struct.element*> is not checked.

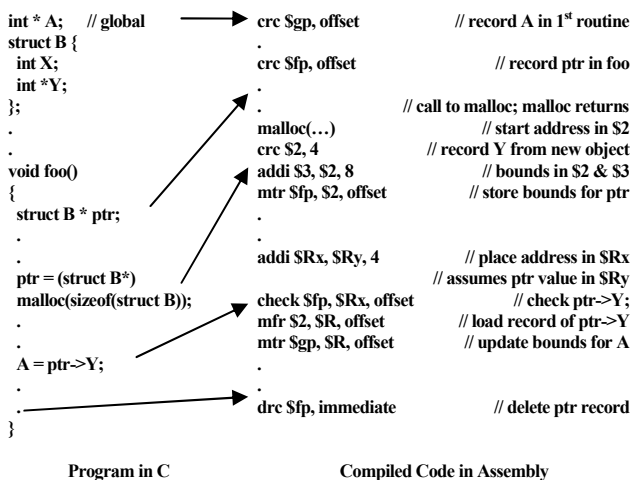

Figure 1: Code fragment with inserted instructions

Figure 1 shows a program written in C on the left and the associated assembly code, including the safety instructions, on the right. Global pointer records are created in the first routine in program execution, such as the *main(…)* routine in C. All local pointer records are created within the routines they are declared, and heap pointer records when they are created. For instance, pointer *Y*'s record in Figure 1 is created after the call to *malloc(…)*. On return from a procedure, the records for its local pointers are deleted. The record for *Y* is not deleted in Figure 1 because it is a heap pointer. The record for *ptr* is updated after *malloc(…)*, and A's record after it is assigned *ptr->Y*. When *ptr* is used to access *Y*, the access is checked against object bounds associated with *ptr*.

If functions are compiled separately and linked together, object bounds for pointers received in a procedure may not be known.

For instance, bounds for a pointer passed from one procedure to another also apply for the receiving pointer, but the bounds are not known to the receiving procedure if the two are compiled separately. We utilize activation records for passing bounds between procedures. The bounds for pointers passed as parameters are placed just before the parameters and in the same order as the parameters. The bounds of return pointers are placed at the start of activation records. Parameter and return pointer bounds are accessed using offsets from frame and stack pointers, respectively.

If parts of an executable are compiled without inserting safety instructions, SafeProc may incur false positives or negatives for pointers and objects that are accessed or modified in those parts.

2.3 SafeProc Architecture – Microarchitecture Extensions

The safety instructions are executed in-order at commit to avoid false errors and continue the application while the safety instructions are executed in the background. Figure 2 shows the pipeline. The safety instructions are only dispatched to the reorder buffer. When they become the oldest instructions, they are committed and removed from the re-order buffer.

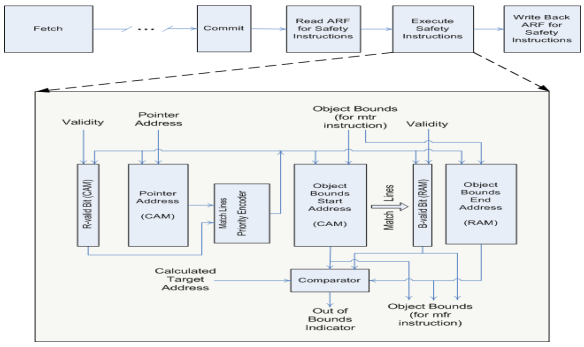

Figure 2: SafeProc pipeline with violation detection hardware

The committed safety instructions read their operands from the architectural register file (ARF), compute the addresses in an additional ALU, and then access the violation detection hardware shown in Figure 2. They write the results back into the ARF. The execution of safety instructions is entirely off the critical path, and can be further pipelined to suit the target clock. Nevertheless, we commit/execute only one safety instruction per cycle to keep the microarchitectural extensions simple.

The violation detection hardware includes a Pointer Address Buffer (*PAB)*, a Start Address Buffer (*SAB)*, and an End Address Buffer (*EAB)*, which are used to store, update, and check the POA records. Two valid bits are used per record; *R-valid* for the validity of a record and *B-valid* for the validity of the object bounds.

A *crc* instruction places the new pointer address in the PAB, sets its R-valid and resets its B-valid bit. A *drc* instruction resets the R-valid bit of the matching entry in the PAB. *Mfr* reads the SAB, EAB, and B-valid fields of the matching valid entry. If the B-valid is set, it places the SAB and EAB values in the registers; otherwise, it resets the registers. *Mtr* places the register values into the SAB and EAB fields of the matching entry. If the register values are not zeros, it sets the corresponding B-valid bit; otherwise, it resets the bit. The *check* instruction reads the SAB, EAB, and B-valid fields of the entry matching the pointer address, and forwards them to the comparator. The other register operand value of the *check* instruction is compared against the bounds for violation detection. The *dob* instruction searches the SAB, and also the backup storage discussed later. The start address of a *dob* instruction may match multiple entries. The B-valid bits of all the matched entries are gang-invalidated. If no matching valid entries are found, a multiple deletion violation is detected.

A detailed analysis [9] suggests that the complete violation detection hardware in Figure 2 requires only about 220K transistors, not including interconnect wires, for 256 entries per buffer. This is a small fraction of the billion-transistor budget of modern chips.

The limited capacity of the buffers restricts the number of pointers that can be handled simultaneously. To handle large number of pointers, we provide microarchitectural support for a *backup-records-storage (BRS)* to store the records evicted from the violation detection hardware. A new record in the violation detection hardware evicts the LRU entry. If a record is fetched from the BRS, its entry in the BRS is invalidated; all records evicted from the violation detection hardware are written back into the BRS. Such handling of records avoids the search for the evicted records.

2.4 BRS Organization

BRS is allocated in the application's virtual address space. BRS is searched when the search fails in the buffers. Searching BRS for read operations stalls the application. To expedite these searches, BRS is organized in multiple buckets. Each bucket consists of multiple bins linked in a list. Separate buckets are provided for pointer addresses and object bounds. An *object address,* which points to the associated object, is also stored along with each pointer address. POA record in the BRS is stored in two parts, one in a pointer address bin and the other in an object bounds bin. To read the record for a particular pointer address, its object address is read, and the object bounds are read using that object address. This BRS structure facilitates faster hardware-based searches in each bin, as discussed later in the section.

BRS will rarely create memory constraints, as was also observed in our experiments, because applications have huge virtual address spaces in current systems. For instance, one million concurrent pointers require only about 16MB. We also discuss an optimization in Section 3 that reduces the memory overhead. In the rare event that an application runs out of virtual memory, error detection may be stopped and the BRS memory may be entirely reclaimed for the application.

To expedite BRS searches, we also provide the *Expedited Search Microarchitecture Extension (ExSME)*, shown in Figure 3. The hash table stores the starting addresses of the buckets. SafeProc loads the valid bits and the pointer or start addresses from a bin into the CAM shown in Figure 3. The required address is then associatively searched with all the valid addresses. The next bin address is also loaded and is used to fetch the contents in the next bin if the search continues across bins. SafeProc uses the processor's L2 cache to store the bin contents. The L1 data cache is not polluted by the POA records, thus limiting the performance impact

of violation detection. To further expedite the search, the bin size is chosen so the valid bits, the pointer or start addresses, and the next bin address are accommodated in one L2 cache line.

To expedite the writeback of evicted POA records into the BRS, the memory locations for the start or pointer addresses of all invalid entries in a bucket are connected in a linked list. A POA record is allocated the entry at the head of this linked list. When the head becomes NULL, i.e. all the entries in the bucket are occupied, a new bin is allocated for that bucket. If an entry is invalidated in the BRS, it is simply made the head of the list, and the current head becomes the next available entry. To avoid BRS write latencies, SafeProc also uses a FIFO *memory update buffer (MUB)*, shown in Figure 3. When instructions that write to POA records commit but miss in the violation detection hardware, they are removed from the ROB and placed in the MUB. The commit of the following instructions continues. Writebacks of evicted POA records are also placed in the MUB. All writes in the MUB are completed in an in-order serialized fashion.

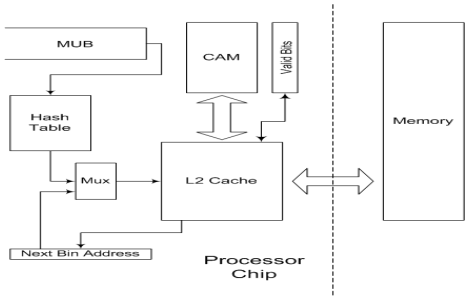

Figure 3: L2 cache interface to speedup searches in the BRS

Values are also bypassed across entries in the MUB. Our analysis shows that ExSME with a 256-entry MUB requires about 200K transistors for 32 buckets each for pointer addresses and object bounds.

3 Enhancements

SafeProc with Delayed Violation Detection: In base SafeProc, *mfr* and *check* instructions may stall the pipeline for a long time because of high BRS search latency. In this technique, these instructions are also placed in the MUB. The performance improves at the expense of delayed detection of violations, as checks may be performed later than the actual access. In production-phase checks, slight delay in violation detection with significantly lower performance impact may be an attractive option. The *mfr*, *check*, and *dob* instructions in the MUB are completed in-order relative to one another, but out-of-order relative to the writes. Furthermore, a waiting *mfr* instruction whose destination architectural register is written by following instructions is not allowed to write its results to the register file. An *mfr* instruction is immediately followed by a dependent *mtr*; it is ensured during compilation. If an *mfr* instruction is placed in the MUB, the following *mtr* instruction is also placed in the MUB, and its entry in the detection hardware is invalidated. When an *mfr* instruction completes, it forwards the object bounds to the dependent *mtr* instruction, which writes the updated record in the violation detection hardware.

Stack-based Approach for Local Pointers: The high BRS search latency results from the large number of POA records maintained in the BRS. This technique uses the observation that the only local pointer records accessed are those in the current scope. In this technique, object bounds associated with local pointers are stored in the corresponding procedure stacks. Local pointers' records are neither stored in the BRS nor in the violation detection hardware. The space to store object bounds in a procedure stack is created and deleted along with the procedure stack. All writes and reads to the object bounds in a procedure stack are performed using load and store instructions, inserted in the code during compilation. The POA records for global and heap pointers are maintained and accessed as in the base SafeProc. Transfers of object bounds from global/heap pointers to local pointers are performed by mfr instructions followed by dependent store instructions. This optimization reduces the access latencies for local pointers (because no BRS search is required), reduces the number of records, and hence the search latencies for all pointers in the BRS, and reduces the memory overhead for each local pointer.

4 Experimental Results

4.1 Experimental Setup

We use a subset of the Olden benchmarks [7] to evaluate SafeProc. These benchmarks are highly pointer intensive and have been extensively used to evaluate software mechanisms for detecting bounds violations [28, 34, 35]. We could not use more benchmarks because the benchmarks are manually modified to insert safety instructions. The benchmarks are then compiled with a gcc-based cross-compiler and executed on a modified SimpleScalar simulator [6] using a customized version of Portable ISA (PISA). Table 2 shows the total number of pointers, and hence the total number of POA records, created for the input sizes used for the benchmarks. Table 2 also shows the maximum number of concurrent pointers for which POA records are stored in the execution of each benchmark. The processor parameters used in our experiments are shown in Table 3.

	Power	Tree-add	Perimeter	Health	Bisort	Mst
Total	232.9	262.1	18.9	77.7	43.3	12.8
MC	0.9	98.3	3.0	2.7	0.6	2.8

Table 2: Total/max concurrent (MC) pointers (in thousands)

4.2 Performance Results

We focus on the performance impact of memory violation detection in SafeProc, compared to a base without detection. The Olden benchmarks do not have any violations. Hence, we inserted several array and pointer reference violations in these benchmarks, similar to those observed in string libraries and SSH/HTTP servers. Our experiments showed that SafeProc detected all inserted errors, without any false positives. We also compare SafeProc with a software mechanism [22] to detect array and pointer reference violations. For this, we installed their compiler patch and executed the compiled benchmarks on SimpleScalar.

Figure 4 presents the increase in execution time for detecting the violations, with respect to the baseline. Larger values are writ-

ten beside the bars. The average performance impact of the software approach is about 479%, while base SafeProc is about 93%, and SafeProc with the two optimizations is less than 5%. Delayed detection optimization performs better than the stack-based optimization for all benchmarks, except *power* and *bisort*. All benchmarks, except *power* and *bisort*, create a considerable number of heap objects, and hiding the latency of BRS searches by delaying the violation detection is more effective than reducing the BRS search latency through stack-based optimization of local pointers.

Parameter	Value
Fetch/Commit	6 instructions
Register File	128 Int/128 FP; 1 cycle inter-subsys. Lat.
Issue Width	4 Int/2 FP instructions
Issue Queue Size	32 Int/32 FP instructions
Branch Predictor	4K bimodal
BTB Size	4K entries, 4-way assoc.
L1 Dcache	64-byte block, 32K, 2-way assoc.; 2 cycle lat.
L1 Icache	32K direct mapped; 2 cycle lat.
L2 cache	128-byte block, unified 512K, 8-way assoc.; 10 cycle lat.
Memory Latency	200 cycles first word, 2 cycle inter-word
ROB Size	192 instructions
Load/store buffer	64 entries

Table 3: Configuration for the base processor

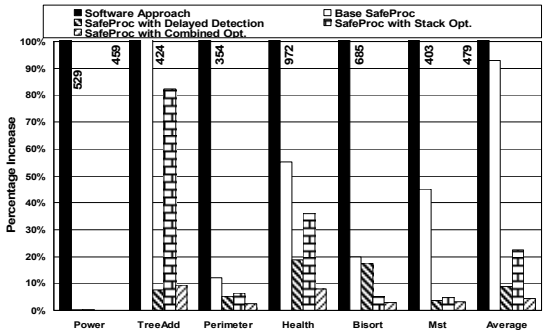

Figure 4: Percentage Increase in Execution Time

Treeadd has the highest performance impact among all the benchmarks because it utilizes a very large number of concurrently active pointers (refer to Table 2). We observed that the average number of bins traversed, and hence the average read miss latency, is low for all benchmarks, except *treeadd*. *Treeadd* also incurs a large number of read misses.

Figure 5 shows the increase in the code size and in the number of instructions executed for base SafeProc and for the software approach. The average code size for SafeProc increases by only about 1%. However, the number of additional instructions executed increases by an average of about 28%. The software approach, on the other hand, more than doubles the code size (133% increase) and executes almost 8x the number of instructions as compared to the base processor. Furthermore, the additional instructions are also executed only at commit, thus having minimal impact on the execution bandwidth. SafeProc also does not significantly interfere with the application data in the L1 data cache, especially for non-inclusive caches used in our experiments. The application and safety instructions access the L1 and L2 cache, concurrently and independently. Figure 5b shows the percentage increase in L1 D-cache and L2 cache misses for the base SafeProc and the software approach. Very similar cache miss rates were observed for SafeProc with the two optimizations. Figure 5b shows that SafeProc has significantly lower cache misses, even for the L2 cache, than the software approach.

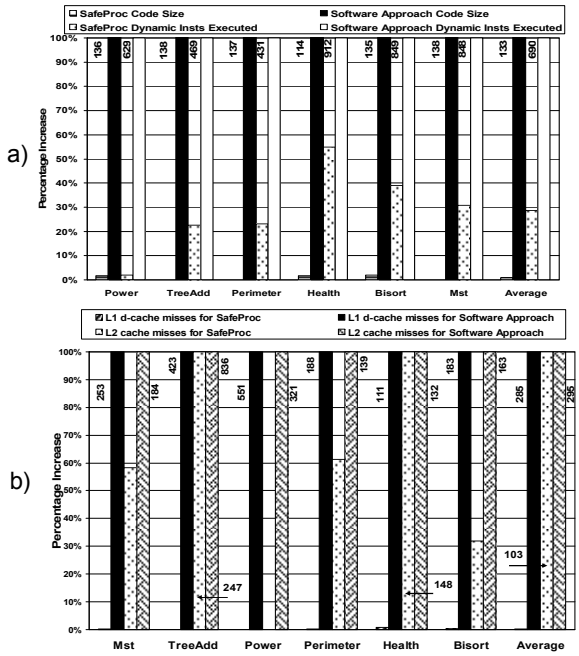

Figure 5: Percentage Increase in Execution Time

We also increased the number of pointers, to an average of about two million, in the benchmarks by increasing the input sizes. As the pointer count increases, the number of pointers stored in the BRS increases, resulting in higher read and write miss penalties. We observed that SafeProc with the two optimizations combined performed an average of about 9% worse than the base processor for these pointers. These experiments use more (512) buckets each for pointer and object addresses to limit the search time in the BRS. Increasing buckets only expands the hash table in ExSME.

5 Related Work

Static tools for reference violation detection review the program code statically and may not be sufficient because they prove correctness of only a fraction of array and pointer references [1, 5, 14, 16, 17, 33]. Dynamic tools detect errors at runtime. The fat-pointer approach [4, 21] stores the object bounds along with the pointers. However, this approach does not handle object deallocations efficiently because all pointers must be checked to determine the ones pointing to a deallocated object, requiring traversing through all structures in the application. Hardbound [37] proposes architectural support to expedite the fat-pointer based approach. The table-based approach [8, 18, 28, 32, 34] maintains a map of pointers to objects in a separate table, thus handling object deallocations efficiently by searching through the centralized database. However, this approach incurs significant overhead due to misses in the huge table. The table-based approach is closer in spirit to our approach.

The third approach [15, 22, 30] stores only address ranges of live objects in a global table. This approach finds the intended object before every pointer arithmetic operation, and ensures that the arithmetic does not cross object bounds. This approach also incurs significant performance loss because of the software-based lookup. This method cannot detect illegal references if deallocated space is allocated to another object, *i.e.* the new object is illegally accessed by a pointer legally pointing to the deallocated object.

Electric Fence [29] places inaccessible pages before and after dynamically allocated objects that result in segmentation faults on out-of-bound accesses. Other approaches, *e.g.* StackGuard [11], specifically target detection of buffer overflows in stacks to prevent overwrite of return addresses. There are techniques (e.g., [10, 12, 13, 31]) that use hardware support to specifically detect external inputs being used as jump addresses or fetched as instructions. These techniques taint external inputs, which propagate during execution. However, these techniques only detect a small fraction of the errors discussed in this paper. The authors in [3] developed a tool that replaces memory violation detection operations with custom instructions, executed on co-processors, for programs compiled with CCured [28] for embedded processors.

6 Conclusions

Software mechanisms to automatically detect array and pointer reference violations have high false positive or negative rates or significant performance overhead, severely restricting their usability. In this paper, we propose architectural support to dynamically detect bounds violations, dangling pointers, and multiple deletions. In this approach, the ISA is extended to include safety instructions to convey compile-time information to the hardware, the processor microarchitecture is extended to efficiently execute these instructions, and the compiler is extended to appropriately insert these instructions during compilation. Overall, our approach handles a large number of pointers with less than 5% performance impact. In comparison, the software mechanism that we evaluated resulted in about 480% performance impact.

7 References

[1] A. Aggarwal and K. Randall, "Related Field Analysis," Proc. of Programming Language Design and Implementation, 2001.

[2] Aleph One, "Smashing The Stack For Fun and Profit," Phrack Volume Seven, Issue Forty-Nine, July 2003.

[3] D.Arora, et al., "Architectural support for safe software execution on embedded processors," In Proc. If Int'l Conf. on Hardware/software codesign and system synthesis, 2006.

[4] T. Austin, S. Breach, and G. Sohi, "Efficient Detection of All Pointer and Array Access Errors," Proc. Programming Language Design and Implementation (PLDI), 1994.

[5] R. Bodik, R. Gupta, and V. Sarkar, "ABCD: Eliminating Array Bounds Checks on Demand," Proc. of PLDI, 2000.

[6] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Computer Arch. News*. 1997.

[7] M. Carlisle, "Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines," PhD Thesis, Princeton University Department of Computer Science, June 1996.

[8] Checker. http://www.gnu.org/software/checker/checker.html

[9] W. Chen, "The VLSI Handbook," 2nd Edition, CRC Press, 2007.

[10] J. Chow, et al., "Understanding data lifetime via whole system simulation," In Proc. of the USENIX Security Symp, 2004.

[11] C. Cowan, et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," Proc. USENIX Security Conf., 1998.

[12] J. R. Crandall and F. T. Chong, "Minos: Control data attack prevention orthogonal to memory model," In Proc. Micro, 2004.

[13] M. Dalton, et al., "Raksha: A flexible informatin flow architecture for software security," Proc. ISCA, 2007.

[14] D. Dhurjati, et al., "Memory safety without garbage collection for embedded applications," ACM Trans. on Embedded Computing Sys, Feb. 2005.

[15] D. Dhurjati and V. Adve, "Backwards-Compatible Array Bounds Checking for C with Very Low Overhead," Proc. Int'l Conf. on Software Engineering (ICSE), 2006.

[16] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," Proc. PLDI, 2003.

[17] V. Ganapathy, et al., "Buffer overrun detection using linear programming and static analysis," Proc. ACM Conf. on Comp. and Comm. Sec., 2003.

[18] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," Proc. of the 1992 Winter Usenix Conference, 1992.

[19] G.J. Holzmann, "The Logic of Bugs," In *Proc. 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 2002.

[20] iSec Security Research: Vulnerabilities 2004. http://www.isec.pl/vulnerabilities04.html

[21] T. Jim, et al., "Cyclone: A Safe Dialect of C," Proceedings of the USENIX Annual Technical Conference, June 2002.

[22] R. Jones and P. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," Proc. of Int'l Workshop on Automated Debugging, May 1997.

[23] E. Larson, "Efficient Dynamic Detection of Input Related Software Errors," PhD Dissertation, University of Michigan, 2004.

[24] E. Marcus and H. Stern, "Blueprints for high availability," John Willey and Sons, 2000.

[25] MS TechNet Security, http://www.microsoft.com/technet/Security/default.mspx

[26] MITRE Corporation. CAN-2004-0416. Common Vulnerabilities and Exposures (CVE) (cve.mitre.org), 2004.

[27] National Institute of Standards and Technology (NIST), Department of Commerce, "Software errors cost U.S. economy $59.5 billion annually," NIST News Release 2002-10, June 2002.

[28] G. Necula, et al., "CCured: Type-Safe Retrofitting of Legacy Code," Proc. of the Symposium on Principles of Programming Languages, 2002.

[29] B. Perens. Electric Fence. http://sunsite.unc.edu/pub/Linux/devel/lang/c/ElectricFence.2.0.5.tar.gz

[30] O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," Proc. of Network and Distributed System Security Symp., 2004.

[31] N. Vachharajani, et al., "RIFLE: An architectural framework for usercentric information-flow security," In Proc. Micro, 2004.

[32] Valgrind. http://valgrind.kde.org.

[33] Y. Xie, A. Chou, D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," Proc. of 11th International Symposium on the Foundations of Software Engineering, Sep. 2003.

[34] W. Xu, et al., "An efficient and backwards-compatible transformation to ensure memory safety of c programs," Proc. Symp. on Foundations of Software Engineering, 2004.

[35] S. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," In Foundations of Software Engineering, 2003.

[36] http://www.mcafee.com/us/local_content/white_papers/wp_ricochetbriefbuffer.pdf

[37] J. Devietti, et al. "Hardbound: Architectural Support for Spatial Safety of the C Programming Language," Proc. ASPLOS 2008.