Technical Report

UCAM-CL-TR-798
ISSN 1476-2986

Number 798

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Practical memory safety for C

Periklis Akritidis

June 2011

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted May 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Practical memory safety for C

Periklis Akritidis

Copious amounts of high-performance and low-level systems code are written in memory-unsafe languages such as C and C++. Unfortunately, the lack of memory safety undermines security and reliability; for example, memory-corruption bugs in programs can breach security, and faults in kernel extensions can bring down the entire operating system. Memory-safe languages, however, are unlikely to displace C and C++ in the near future; thus, solutions for future and existing C and C++ code are needed.

Despite considerable prior research, memory-safety problems in C and C++ programs persist because the existing proposals that are practical enough for production use cannot offer adequate protection, while comprehensive proposals are either too slow for practical use, or break backwards compatibility by requiring significant porting or generating binary-incompatible code.

To enable practical protection against memory-corruption attacks and operating system crashes, I designed new integrity properties preventing dangerous memory corruption at low cost instead of enforcing strict memory safety to catch every memory error at high cost. Then, at the implementation level, I aggressively optimised for the common case, and streamlined execution by modifying memory layouts as far as allowed without breaking binary compatibility.

I developed three compiler-based tools for analysing and instrumenting unmodified source code to automatically generate binaries hardened against memory errors: BBC and WIT to harden user-space C programs, and BGI to harden and to isolate Microsoft Windows kernel extensions. The generated code incurs low performance overhead and is binary-compatible with uninstrumented code. BBC offers strong protection with lower overhead than previously possible for its level of protection; WIT further lowers overhead while offering stronger protection than previous solutions of similar performance; and BGI improves backwards compatibility and performance over previous proposals, making kernel extension isolation practical for commodity systems.

Acknowledgements

I would like to thank my supervisor Steve Hand for his guidance, as well as my hosts at Microsoft Research Cambridge, Manuel Costa and Miguel Castro; I greatly enjoyed working with them. I am grateful to Brad Karp and Prof. Alan Mycroft for their valuable feedback, and to Prof. Jon Crowcroft for his encouragement. I am also indebted to my previous supervisors and colleagues, especially Prof. Evangelos Markatos for introducing me to research.

During the last years, I have enjoyed the company of my labmates Amitabha Roy, Bharat Venkatakrishnan, David Miller, Myoung Jin Nam, Carl Forsell, Tassos Noulas, Haris Rotsos, and Eva Kalyvianaki. I am also grateful to my friends Mina Brimpari, Daniel Holland, and Lefteris Garyfallidis for their support and company.

Lastly, I am immensely grateful to my parents, Themistoklis and Maria for their crucial early support. Thank you!

This work was supported by Microsoft Research with a postgraduate scholarship.

Contents

List of figures

List of tables

Chapter 1

Introduction

C and C++ are holding their ground [52, 157] against new, memory-safe languages thanks to their capacity for high performance execution and low-level systems programming, and, of course, the abundance of legacy code. The lack of memory safety, however, causes widespread security and reliability problems. Despite considerable prior research, existing solutions are weak, expensive, or incompatible with legacy code.

My work demonstrates that a spectrum of efficient backwards-compatible solutions are possible through careful engineering and judicious tradeoffs between performance and error detection. The key idea is to enforce the minimum integrity guarantees necessary to protect against memory-safety attacks and operating system crashes beyond the protection of current practical solutions, while maintaining a low cost instead of aiming to detect all memory-safety violations at a prohibitive cost.

Finally, while the rest of this dissertation focuses on programs written in C, the observations and solutions are intended to apply equally well to programs written in C++.

1.1 The problem

C facilitates high performance execution and systems programming through lightweight abstractions close to the hardware. Low-level control, however, undermines security and reliability. Support for arbitrary pointer arithmetic, manual memory management, unchecked memory access, and the lack of a built-in string type burden programmers with safeguarding the integrity of the program, introducing ample opportunities for programmer errors. Without runtime checking, such errors may be aggravated to catastrophic failures. For example, attackers can exploit bugs like that of Figure 1.1 anywhere in a program to breach security, and similar faults in kernel extensions can bring down the entire operating system.

This is in direct contrast with *safe* languages that use strict type checking and garbage collection to make programmer errors less likely and mitigate the remaining memory-safety bugs through runtime checks. In particular, non-null pointers in safe languages always refer to their intended target objects, and bugs like that of Figure 1.1 cause runtime exceptions instead of memory corruption.

Safe languages, however, cannot displace C in the near future, because performance, direct hardware control, and its near-universal support across platforms continue to

```
1 char buf[N];
2 char *q = buf;
3 while (*p)
4     *q++ = *p++;
```

Figure 1.1: The archetypal memory corruption error: writes through pointer `q` may overwrite memory beyond the intended buffer `buf` if `strlen(p)` can become `>= N`.

attract developers. Furthermore, existing critical software, that is more exposed to hostile input or more likely to run in the kernel address space, is written predominantly in C. This includes network servers, multimedia decoders, file systems, and device drivers. Finally, the sheer amount of legacy code in C discourages porting to safe languages. Without a complete switch to safe languages on the horizon, solutions readily applicable to legacy code are called for.

1.2 State of the art

The research community has long recognised this problem and has developed many solutions. Static analysis can be used to find some security and reliability defects during development, and checks can be inserted in the code automatically to detect some of the remaining errors at runtime. In addition, return-address protection (*e.g.* the `/GS` option in Microsoft compilers), address-space-layout randomisation (ASLR) [151], and data-execution prevention (DEP) [98] can all be used to thwart some but not all attacks. Finally, paging hardware [40, 100] and software-based fault-isolation (SFI) [163, 97, 159] can be used to contain faults in kernel extensions and user programs.

Despite previous research, however, problems persist in the real world. Buffer overflows and similar vulnerabilities are routinely reported online [160, 153, 132], and are exploited by attackers to break into networked systems, spread malware, and bypass the security protections of closed devices such as gaming consoles and mobile phones. Furthermore, despite support for user-space kernel extensions, most device drivers still run in the kernel address space.

These problems persist because adopted proposals provide insufficient protection, while solutions that could provide adequate protection if used, are either too slow for practical use or break backwards compatibility by requiring significant porting or preventing linking with legacy libraries.

For example, automatically added bounds checks [107, 105, 43, 130, 170, 104] add significant overhead and may modify the pointer representation, while static analysis is incomplete, and has limited effectiveness without pervasive source-code annotations or modifications. Programmers, however, dislike having to annotate source code, or to weed out static analysis false positives, and reject solutions which suffer high runtime overheads or break binary compatibility.

As for solutions employing separate hardware address spaces to isolate kernel extensions, they too incur prohibitive performance overheads because drivers perform

little processing and lots of I/O, resulting in frequent expensive hardware address-space switches. In addition, existing solutions break backwards compatibility, as existing device drivers must be ported to new, coarse-grained APIs. While software-based sandboxing solutions such as software fault isolation (SFI) can address the cost of frequent hardware address-space switching, they still fail to preserve backwards compatibility with legacy APIs at a reasonable cost.

Thus, the state of the art is neatly summarised in the adage: solutions are safe, fast, and backwards-compatible—pick *two*. This hinders the adoption of comprehensive solutions, since performance and legacy code are the main drivers of C and C++ use. Hence, only fast, backwards-compatible solutions are likely to see wide adoption; but to date, these do not provide sufficient protection to curb memory-safety problems in practice.

1.3 Requirements and challenges

I have identified a number of requirements for successful solutions, based on shortcomings that have limited the effectiveness or hindered the adoption of previous proposals.

1.3.1 Adequate protection

First, solutions must have the potential to significantly and reliably reduce the rate of exploitable bugs in programs. Therefore, solutions that prevent specific attacks, but allow exploiting the same bugs using different techniques, are inadequate. For example, stack-based buffer-overflow defences [48] designed to thwart attacks that overwrite saved return addresses may still allow the exploitation of the same bugs to overwrite local variables instead. Conversely, adequate solutions are not required to eliminate all exploitable bugs. In fact, this goal is arguably unattainable for unmodified C programs. Partial solutions can keep memory-safety problems in check by preventing a significant fraction of attacks and being resilient against workarounds, thus improving the status quo.

Moreover, accurately detecting every bug is *not* required for protection against attacks or operating system crashes. While accuracy is essential for debugging, where incidentally performance requirements are less pronounced, there is great value in efficiently preventing the exploitation of dangerous bugs in production systems, even if "harmless" bugs—those that cannot be exploited by attackers or propagate across kernel extensions—are silently ignored. Consequently, adequate security and safety solutions may not necessarily double as debugging tools.

Second, solutions must be practical, or they will not be adopted. This boils down to good performance, backwards compatibility, and lack of false alarms.

1.3.2 Good performance

Compared to other widespread security threats, including SQL injection and cross-site scripting, mitigating memory errors is particularly challenging because runtime

checking may result in severe performance degradation—an order of magnitude degradation is common for comprehensive solutions. Unfortunately, the majority of users are unwilling to accept such a noticeable performance degradation. Ideally, protection should come without any cost because it is not clear what constitutes acceptable performance degradation. My goal in this work was a runtime overhead around 10–15%, which I believe should be acceptable in practice. Moreover, space overhead is often neglected by many proposals. While secondary to runtime overhead, it should not be excessive, because the memory overheads of individual applications add up resulting in degradation of overall system performance. I aimed for a memory overhead around 10%; I believe that memory overheads beyond 50% incurred by many existing solutions are excessive.

1.3.3 Pristine sources

Backwards compatibility is critical for wide adoption, because the human effort associated with porting is likely the most significant cost in deploying any solution. The ideal for backwards compatibility is a solution which works with pre-existing binary code. However, I believe it is reasonable to aim for working with unmodified source code instead, and simply require recompilation by software vendors (or even software distributors in the case of open-source software).

Unfortunately, even developing solutions that can take advantage of source-code availability is challenging, and some manual porting may still be necessary, for example, in response to evolving operating system APIs, or for programs using custom memory-allocators. Porting the solution itself to new operating system versions is acceptable, because end-programmers are shielded from this effort and operating system APIs are relatively stable. As for programs with custom memory allocators, they should not break after applying a solution, but it may be necessary to tweak the custom memory allocation routines to take full advantage of a solution's protection.

Finally, the need for code annotations should also be avoided. This limits the scope of solutions to security properties that can be inferred from source code alone, but is important for saving human effort.

1.3.4 Binary compatibility

Backwards compatibility is also important at the binary level. Solutions should preserve binary compatibility to allow linking against legacy binary code such as third-party libraries. In these cases, no protection (or limited protection) is offered against bugs in the unprotected binaries, but incremental deployment is made possible, and protection is still offered against bugs in the code under the programmer's control. This can also help with performance by enabling trusted libraries (for example, heavily audited performance critical code) to run without protection by choice.

1.3.5 Low false positive rate

Finally, while the false negatives rate merely influences effectiveness, a significant false positives rate may render solutions entirely unusable. A few false positives

conclusively addressable with minimal source-code modifications may be acceptable when security is paramount, but widespread adoption may be significantly hindered; thus, a low false positives rate is critical.

In practice, the tradeoffs between the requirements for performance, protection, backwards compatibility, and low false-positive rate, make the design of a comprehensive solution challenging. A testimony to the challenges in addressing memory errors is that despite intense scrutiny, the problem has persisted for decades (Section 2.1).

1.4 Hypothesis and contributions

My work aims for effective, fast, and backwards-compatible solutions to memory safety problems in C programs. I have developed compiler-based tools for generating executables hardened against memory-safety errors through runtime checks. In contrast to most of the prior work addressing performance mainly by reducing the number of runtime checks through static analysis, my approach aims to lower the cost of individual runtime checks remaining after static analysis. To this end, I introduce novel integrity rules offering adequate protection that are cheap to enforce at runtime (Section 1.4.2). Advances over previous work are made possible using a combination of judicious tradeoffs between performance and safety by enforcing only the minimum guarantees necessary to prevent memory corruption, and careful design of the data structures and operations to support the common case efficiently. The work in this dissertation has been published in [5, 6, 33].

1.4.1 Hypothesis

My thesis is that protecting the execution integrity of code written in memory-unsafe languages against memory-safety errors can be made practical. A practical solution requires minimal porting of existing source-code, incurs acceptable performance degradation, allows incremental deployment, and avoids false alarms for almost all programs. My work shows how to achieve these by striking a better balance between protection and performance.

1.4.2 New integrity properties

When I started this work, existing solutions for mitigating the lack of memory safety, such as bounds checking and taint tracking, while still prohibitively expensive, had been heavily optimised, thus performance gains were unlikely through better implementation of an existing integrity property. Efficient solutions, on the other hand, offered relatively little protection. That led me into refining the protection guarantees, looking for sweet spots in the design space to maximise protection and minimise cost. I came up with two new integrity properties.

Baggy bounds

Traditional bounds checking prevents spatial memory-safety violations by detecting memory accesses outside the intended object's bounds. I observed, however, that padding objects and permitting memory accesses to their padding does not compromise security: some spatial memory safety violations are silently tolerated (they just access the padding), while those that would access another object are still detected.

This observation enables tuning the bounds to increase performance. By padding every object to a power-of-two size and aligning its base address to a multiple of its padded size, bounds can be represented with the binary logarithm of the power-of-two size, which can fit in a single byte for address spaces up to 2^{256} bytes. That is an eight-fold improvement over traditional bounds representations on 32-bit systems that require eight bytes (four for the base address plus four for the object size). The compact bounds representation can help replace expensive data structures used in previous work with efficient ones, reducing the time to access the data structures, and, despite trading space for time by padding objects, allowing for competitive memory overhead due to less memory being used for the data structures storing the bounds. Furthermore, with object bounds constrained this way, bounds checks can be streamlined into bit-pattern checks. Finally, on 64-bit architectures, it is possible to use spare bits in the pointers to store the bounds without having to change the pointer size or use an auxiliary data structure.

Write integrity

A fundamental cost of many comprehensive solutions for memory safety, including baggy bounds checking, is tracking the intended target object of a pointer. Intuitively, write integrity approximates this relation without tracking pointers.

Write integrity uses interprocedural points-to analysis [7] at compile time to conservatively approximate the set of objects writable by an instruction, and enforces this set at runtime using lightweight checks to prevent memory corruption. Additionally, it introduces small guards between the original objects in the program. As these guards are not in any writable set, they prevent sequential overflows (Section 2.2) even when the static analysis is imprecise. WIT maintains call stack and heap metadata integrity (Section 2.4) because return addresses and heap metadata are excluded from any permitted set by default, and can prevent double frees by checking free operations similarly to writes in combination with excluding released memory from any writable set. It prevents more memory errors on top of these, subject to the precision of the analysis. In fact, subsequent write integrity checks can often nullify bugs due to corruption of sub-objects, dangling pointers, and out-of-bounds reads.

Write integrity is coupled with control-flow integrity (CFI, Section 2.4) [86, 1, 2] to reinforce each other. CFI prevents bypassing write checks and provides a second line of defence. In turn, the CFI implementation does not have to check function returns, as call-stack integrity is guaranteed by the write checks.

1.4.3 New implementations

I developed three solutions, BBC (baggy bounds checking) based on the baggy bounds integrity property, and WIT (write integrity testing) and BGI (byte-granularity isolation) based on write integrity. BBC and WIT are designed to harden user-space programs, while BGI is designed to harden kernel extensions, and is implemented for the Microsoft Windows operating system. All three solutions were implemented as compilation phases generating instrumented code that is linked with a library containing runtime support and wrappers for standard library functions. I used the Phoenix compiler framework [99] from Microsoft to implement the compilation phases in the form of Phoenix compiler plugins.

BBC, a bounds checking system based on baggy bounds, uses a binary-buddy-system memory allocator for the heap, changes the layout of static and automatic objects, and instruments code during compilation to enforce baggy bounds. I implemented a 32-bit and a 64-bit version to experiment with different implementation options.

WIT uses a points-to analysis due to Andersen [7] at compile time (using the implementation from [32] that in turn is based on that described in [76]). It then instruments code to enforce write integrity and control-flow integrity, and changes the memory layout to insert guards between objects and ensure the proper alignment needed for efficient runtime checks.

BGI, in addition to detecting errors in kernel extensions similarly to WIT, uses efficient byte-granularity memory protection to contain memory faults within protection domains according to the documented or implied memory ownership semantics of the interface between drivers and the kernel. BGI isolates kernel extensions from each other and the kernel and ensures type safety for kernel objects using API interposition to assign writable sets and types dynamically, which are enforced using API interposition and checks inserted in the driver code during compilation.

Finally, as an offshoot of this work, the same underlying permission-checking mechanism was used for enforcing a sharing policy for concurrent programs [96]. An advantage over prior work is that sharing policy annotations are enforced in the presence of memory errors without additional overhead. This work, however, is not included in this dissertation.

1.4.4 Experimental results

My experimental results show that all three solutions improve the state of the art, and confirm the hypothesis that adequate protection can be practical.

BBC has low runtime overhead in practice—only an 8% throughput decrease for Apache—and is more than two times faster than the fastest previous technique [57] on the same CPU-bound benchmarks and about five times faster—using less memory—than recording object bounds using a splay tree. Its average runtime overhead for CPU-intensive benchmarks is 30% with an average peak memory overhead of 7.5%. BBC has better performance than previous solutions with comparable protection, but its running time overhead exceeds the ideal target of 10% prescribed in Section 1.3, leading me to explore further performance improvements with WIT.

WIT has better coverage than solutions with comparable performance, and has consistently low enough overhead to be used in practice for protecting user-space applications. Its average runtime overhead is only 7% across a set of CPU-intensive benchmarks and it is negligible when I/O is the bottleneck. Its memory overhead is 13% and can be halved on 64-bit architectures.

BGI extends WIT to isolate device drivers from each other and the kernel, offering high protection with CPU overhead between 0% and 16%.

All three solutions satisfy the requirement of backwards compatibility, both at the source and binary level. BBC and WIT can compile user-space C programs without modifications, and these programs can be linked against unmodified binary libraries. BGI can compile Windows drivers [114] without requiring changes to the source code, and these drivers can coexist with unmodified binary drivers.

1.5 Organisation

The rest of this dissertation is organised as follows. Chapter 2 provides background information related to this work. It summarises the long history of the problem, classifies the weaknesses this work aims to address, and introduces various integrity guarantees to give some background context for the design process.

The next three chapters present my work on providing effective and practical protection for user-space and kernel-space C programs. Chapter 3 addresses spatial safety using the baggy bounds integrity property. It shows how baggy bounds checking (BBC) can be used to enforce spatial safety in production systems more efficiently than traditional backwards-compatible bounds checking [82]. Special attention is given to how 64-bit architectures can be used for faster protection. To address temporal safety, BBC can be combined with existing techniques, as discussed in Section 3.10.3, to provide a complete solution.

Chapters 4 and 5 present further work that was motivated by two reasons. First, I tried to lower overheads further by making runtime checks even more lightweight. This led to the formulation of write integrity testing (WIT) in Chapter 4, with a design aiming to maximise the safety that can be provided using the cheapest-possible runtime checks. WIT can also protect against some temporal memory-safety violations due to uses of pointers to deallocated objects.

Next, I tried to address memory-safety issues in the context of legacy Windows device drivers, highlighting temporal-safety risks beyond memory deallocation, which are addressed in Chapter 5. In short, I observed that lack of temporal access-control allows memory corruption faults to propagate across kernel components. For example, a memory error in a kernel extension corrupting an object allocated by the extension but referenced by kernel data structures can cause kernel code to corrupt memory when using the object. These errors can be prevented by enforcing dynamic access rights according to the kernel API rules to prevent extensions from corrupting objects they no longer own.

Finally, Chapter 6 critically reviews related work, and Chapter 7 concludes.

Chapter 2

Background

2.1 History

A look at the long history of memory-safety problems can highlight some challenges faced by proposed solutions. Attackers have shown significant motivation, quickly adapting to defences, and a spirit of full disclosure has emerged in condemnation of "security through obscurity".

After Dennis Ritchie developed C in 1972 as a general-purpose computer programming language for use with the UNIX operating system, it quickly became popular for developing portable application software. Bjarne Stroustrup started developing C++ in 1979 based on C, inheriting its security and reliability problems. According to several crude estimates [52, 157], most software today is written in one of these languages.

Buffer overflows were identified as a security threat as early as 1972 [8], and the earliest documented exploitation of a buffer overflow was in 1988 as one of several exploits used by the Morris worm [116] to propagate over the Internet. Since then, several high-profile Internet worms have exploited buffer overflows for their propagation, including Code Red [178] in July 2001, Code Red II in August 2001, and Nimda in September 2001, then SQL Slammer [102] in January 2003 and Blaster [13] in August 2003, until attacker attention shifted to stealthier attacks such as botnets and drive-by attacks that generate illegal revenue rather than mayhem.

Stack-overflow vulnerabilities and their exploitation became widely known in 1996 through an influential step-by-step article [113] by Elias Levy (known as Aleph One). Early stack-based buffer overflows targeting the saved function return address were later extended to non-control-data attacks [37] and heap-based overflows [119, 44]. In August 1997 Alexander Peslyak (known as Solar Designer) showed how to bypass the then promising non-executable stack defences [53]. Exploitation mechanisms for memory errors beyond buffer overflows, such as integer overflows [23, 4], format-string vulnerabilities, and dangling pointers followed soon. Format-string vulnerabilities were discovered in June 2000, when a vulnerability in WU-FTP was exploited. The first exploitation of a heap overflow was demonstrated in July 2000 by Solar Designer [54]. Most browser-based heap-memory exploits now use a technique first described in 2004 by SkyLined [137], called heap spraying, that overcomes the unpredictability of heap memory-layout by filling up the heap with many attacker-controlled objects allocated through a malicious web-page. The first exploit for a dangling pointer vulnerability, latent in the Microsoft IIS web server since December

2005, was presented [3] in August 2007, complete with an instruction manual on how to find and exploit dangling pointer vulnerabilities.

Twenty years after the first attack, we are witnessing frequent exploitation of buffer overflows and other memory errors, as well as new attack targets. In 2003, buffer overflows in licensed Xbox games were exploited to enable running unlicensed software. Buffer overflows were also used for the same purpose targeting the PlayStation 2 and the Wii. Some trends have changed over the years. Instead of servers exposed to malicious clients, Web browsers and their plugins are becoming targeted through malicious websites. Moreover, vulnerabilities in thousands of end-user systems may be exploited for large-scale attacks affecting the security of third parties or the stability of the Internet as a whole.

2.2 Common vulnerabilities

Low-level control makes C very error-prone. Here, I list several common programmer errors and their safety consequences. The enumeration helps frame the scope of this work. An exhaustive dictionary of software weakness types is available from CWE [154].

Insufficient bounds checks The predominant memory corruption vulnerability is the notorious buffer overflow due to insufficient bounds checks, for instance when copying a zero-terminated string into a fixed-size buffer, thereby causing data to be stored outside the memory set aside for the buffer.

Figure 2.1 illustrates an important practical distinction between *sequential* buffer overflows (or underflows), accessing memory addresses consecutively, inevitably accessing memory at the object bounds before straying into memory adjacent to the buffer; and *random access* bound errors, giving access to arbitrary memory locations without accessing memory in-between.

The rest of this section illustrates that lack of bounds checks is not the only source of memory errors.

Integer errors Silent integer overflow (wraparound) errors, and integer coercion (width and sign conversion) errors, while not memory errors themselves, may trigger bound errors [23, 4]. For instance, a negative value may pass a signed integer check guarding against values greater than a buffer's size, but subsequent use as an unsigned integer, *e.g.* by a function like `memcpy`, can overflow the buffer. Another example is when a wrapped-around negative integer size is passed to `malloc` causing a zero-sized buffer allocation that leads to an overflow when the buffer is later used.

Uncontrolled format-strings Another vulnerability involves using program input as a format string for `printf` or similar functions. Memory can be corrupted if the input includes bogus format specifiers, which use unintended, potentially attacker-controlled, arguments from the stack. For example, arbitrary data may be written to arbitrary locations using the `%n` format specifier, which overwrites the memory location specified by its argument with the number of characters written so far.

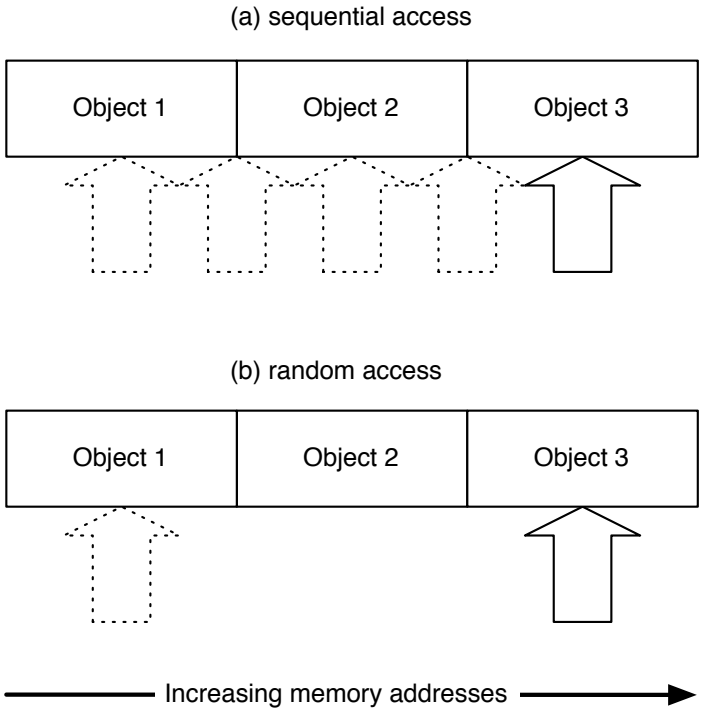


Figure 2.1: A sequential buffer overflow (a) cannot access Object 3 using a pointer intended for Object 1 without accessing Object 2, but a random access bound error (b) can access Object 3 without accessing Object 2.

Use-after-free Manual memory management can be another source of errors. Releasing referenced memory may result in dangling pointers. When this memory is reused, dangling pointers will refer to an object other than the intended one.

Use of uninitialised variables A similar vulnerability results from the use of uninitialised data, especially pointers. Uninitialised data may be controlled by the attacker, but use of uninitialised data can be exploited even if the data is not controlled by the attacker.

Double frees Another error related to manual memory management are double-free bugs. If a program calls `free` twice on the same memory address, vulnerable heap-memory allocators re-enter double-freed memory into a free list, subsequently returning the same memory chunk twice, with the allocator erroneously interpreting the data stored in the first allocation as heap-metadata pointers when the second allocation request for the doubly entered chunk is processed.

2.3 Possible attack targets

Attackers can use memory errors described in the previous section to subvert vulnerable programs. This is typically achieved by overwriting a critical value to divert

control flow to attacker code, or otherwise alter program behaviour. We will see, however, that invalid reads and legitimate code can be used too. Here I discuss the various critical program elements targeted by attackers, to help with evaluating the effectiveness of proposals.

Return address Typical buffer overflows caused by insufficient bounds checking while copying from one buffer to another give access to memory adjacent to the overflowed buffer. With the call stack growing towards lower addresses, the currently executing function's return address is conveniently located adjacent to the top of the function's stack frame, allowing runoff contents of an overflowed stack buffer to overwrite the return address with the address of attacker-injected code. This code, known as *shellcode*, can be smuggled into the process as data, often inside the stack buffer being overflowed. However, the address of this buffer, required for overwriting the return address, is known only approximately. This limitation is overcome by prepending the shellcode with a sequence of `nop` instructions dubbed a *sled*, or by first diverting control to an indirect control-transfer instruction at a known address in the program that uses a register known to contain the target buffer's address.

Heap-based and static variables Buffer overflows in heap-based and static buffers can also be exploited to overwrite targets on the heap or in global variables, as discussed next.

Control-flow data In addition to return addresses, other program control-flow data can be targeted as well, including exception handlers and virtual-table pointers which abound in C++ programs. In fact, any function pointer may be targeted, including function pointers in program data, and, in general, targets not adjacent to an overflowed buffer.

Even an invalid control-flow transfer where the attacker has limited or no control over the target can be exploited. Heap spraying (see [125] for a detailed exposition) can be used to make control flow likely to land on shellcode by allocating many objects filled with shellcode.

Non-control-flow data In addition to control-flow data, attackers can target a variety of non-control-flow application data including user identity, configuration, input, and decision-making data [37]. Furthermore, a data pointer can be corrupted as an intermediate step in corrupting some other target. For example, a pointer adjacent to a buffer on the stack can be overwritten with the address of a target to be overwritten indirectly through subsequent use of the pointer by the program. Finally, a buffer overflow occurring in the heap data area can overwrite malloc metadata such as pointers linking together free memory blocks, and then use the resulting pointer when the block is removed from the free list to overwrite another target.

Existing code As defences made data and stack sections non-executable or prevented injecting new code, *return-to-libc* attacks were developed which divert execution to an existing function, using an additional portion of the stack to provide

arguments to this function. Possible targets include `system` to execute a shell or `VirtualProtect` to make data executable again. An elaboration of this technique [134, 28] targets short instruction sequences scattered in the executable that end with a return instruction. Longer operations can be executed using the return instructions to chain individual short sequences through a series of return addresses placed on the stack. This technique greatly increases the number of illegal control-flow transition targets.

Exploiting reads Finally, even memory reads can indirectly corrupt memory. Illegal reads of pointer values are particularly dangerous. For example, if a program reads a pointer from attacker-controlled data and writes to memory through that pointer, an attacker can divert the write to overwrite an arbitrary target. The heap spraying technique can be used when the attacker has no control over the value read.

2.4 Integrity guarantees

Various integrity guarantees have been formulated to address different subsets of vulnerabilities or protect different attack targets. They may subsume, intersect, or complement each other, and they come with different costs. This section classifies them to understand better the inherent overheads and various tradeoffs between protection and enforcement cost for different combinations of integrity guarantees.

Spatial safety guarantees the absence of bound errors, and is typically provided by high-level languages such as Java. Spatial safety is highly desirable, but can be expensive to enforce because it requires frequent runtime checks. Thus, for low-level languages, it is often enforced only partially, by focusing on high-risk operations, such as memory writes and string operations, or high-risk attack targets, such as return addresses and heap metadata. For example, enforcing *call-stack integrity* is cheap because integrity checks can be localised to function returns [48]. Similarly, enforcing *heap metadata integrity* is cheap because integrity checks can be localised into memory management routines [128].

Temporal safety complements spatial safety by preventing accesses to memory after it has been deallocated and possibly reused to back another memory allocation while pointers to the original allocation remain in use. High-level languages, such as Java, address temporal safety using garbage collection or reference counting. In low-level languages, it can be enforced by tracking pointers and checking every memory access [11, 170]. This is expensive, but most of the cost can be shared with spatial protection. Other mechanisms like reference counting [70], conservative garbage collection [24], or region-based memory management [71] focus exclusively on temporal safety, but have backwards compatibility or performance problems. *Type homogeneity* [60, 59] is a weaker form of temporal safety that allows dangling pointers but constrains them to point to objects of the same type and alignment, making dangling pointer dereferences harder to exploit. Enforcement can be localised into memory management routines.

Exploiting a memory error to subvert security often violates additional integrity rules on top of memory safety that may be easier to enforce than memory safety. For example, *control-flow integrity* [86, 1, 2] builds on the observation that most attacks,

through temporal or spatial violations—the exact avenue of attack is irrelevant—will eventually divert normal control flow. It addresses a significant class of attacks and affords efficient implementations because checking is localised to control-flow transfers. It cannot, however, protect from attacks against non-control-flow data discussed in Section 2.3. Control-flow integrity is complemented by *data-flow integrity* [32] which handles non-control-flow data corruption, at significantly higher cost, by interposing on memory accesses. Violations of *information-flow integrity*, *e.g.* when the program-counter register points to or is loaded with untrusted input data, also signify an attack, but the information flow tracking [109] necessary to detect them is expensive in software. *Non-executable data* builds on the premise that control flow will eventually be diverted to attacker-provided code masked as data, and enforces a separation between data and code, to prevent malicious code injection. It is cheap using hardware support available on modern x86 processors, but cannot protect against the non-control-flow and return-to-libc attacks discussed in Section 2.3.

Some integrity guarantees can be probabilistic, *e.g.* by relying on randomisation or obfuscation [67]. These can be cheaper to enforce, but may still allow attacks if the attacker can try repeatedly, or may affect a significant portion of a large vulnerable population, *e.g.* in the case of a large-scale worm attack. Obfuscation is often vulnerable to information leakage attacks, which are often harmless otherwise, but can be used to break obfuscation's secrecy assumptions [135, 142].

This dissertation introduces new integrity guarantees that are among the most effective and at the same time most efficient to enforce. The next chapter discusses baggy bounds checking, an efficient and backwards-compatible variation of bounds checking.

Chapter 3

Baggy bounds checking

Safe languages enforce spatial memory-safety by bounds-checking array accesses using bounds stored at the base of arrays. The main challenge for retrofitting such bounds-checking to C programs are internal pointers (pointers to some element of an array or sub-object of a structure) and the lack of a built-in mechanism to determine the bounds given an arbitrary pointer.

One approach for retrofitting bounds checking to C is to use so called "fat pointers" to augment the pointer representation in memory with the bounds of the pointer's target (*e.g.* [11]). Spatial safety is enforced by consulting these bounds whenever a pointer is dereferenced. This technique, however, suffers from a number of problems. Most importantly, it changes the memory layout of structure fields, breaking binary compatibility with pre-existing code that has not been compiled with fat-pointer support, such as libraries used by the program. Backwards compatibility, however, is a major requirement for widespread adoption. Further problems may necessitate source-code modifications: memory accesses to pointer variables are no longer atomic, which may break concurrent programs, and programs using unions containing overlapping pointer and integer fields enable inconsistent updates of the address component of fat pointers through the overlapping integer union-field.

Another approach developed to solve the problems of fat pointers is to use a separate data structure to associate pointers with their bounds (*e.g.* [82]). This solves the problem of backwards compatibility with binary libraries by storing bounds out-of-band, but introduces additional overhead for retrieving bounds from this data structure. If this overhead is too high, it is unlikely the solution will be adopted.

This chapter describes and evaluates a bounds-checking mechanism that remains backwards compatible by not changing the pointer representation, but which substantially reduces the performance overhead associated with storing bounds out-of-band. I found that enforcing baggy bounds (introduced in Section 1.4.2) enables new implementation options that lead to cheaper spatial protection. This mechanism is called baggy bounds checking (BBC).

3.1 Overview

Figure 3.1 shows the overall system architecture of BBC. It converts source code to an intermediate representation (IR), identifies potentially unsafe pointer-arithmetic operations, and inserts checks to ensure their results are within bounds. Then, it links

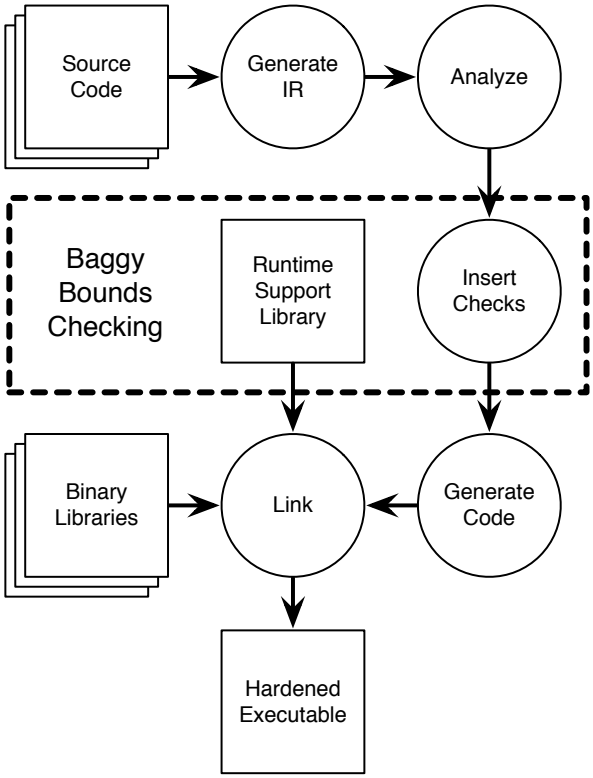


Figure 3.1: Overall bounds-checking system architecture, with BBC's mechanisms highlighted within the dashed box.

the generated code with a runtime support library and binary libraries—compiled with or without checks—to create an executable hardened against bound errors.

Overall, the system is similar to previous backwards-compatible bounds-checking systems for C, and follows the general bounds-checking approach introduced by Jones and Kelly [82]. Given an in-bounds pointer to an object, this approach ensures that any derived pointer points to the same object. It records bounds information for each object in a *bounds table*. The bounds are associated with memory ranges, instead of pointers. This table is updated on allocation and deallocation of objects: this is done by the `malloc` family of functions for heap-based objects, on function entry and exit for local variables, and on program startup for global variables. (The `alloca` function is supported too.)

I will use the example in Figure 3.2 to illustrate how the Jones and Kelly approach bounds-checks pointer arithmetic. The example is a simplified web server with a buffer-overflow vulnerability. It is inspired by a vulnerability in nullhttpd [111] that can be used to launch a non-control-data attack [37]. Line 7 was rewritten into lines 8 and 9 to highlight the erroneous pointer arithmetic in line 8.

When the web server in Figure 3.2 receives a CGI command, it calls function `ProcessCGIRequest` with the message it received from the network and its size as arguments. This function copies the command from the message to the global variable `cgiCommand` and then calls `ExecuteRequest` to execute the command. The variable `cgiDir` contains the pathname of the directory with the executables that can be invoked by CGI commands. `ExecuteRequest` first checks that `cgiCommand` does not

```
1  char cgiCommand[1024];
2  char cgiDir[1024];
3
4  void ProcessCGIRequest(char *msg, int sz) {
5    int i = 0;
6    while (i < sz) {
7      // cgiCommand[i] = msg[i];
8      char *p = cgiCommand + i;
9      *p = msg[i];
10     i++;
11   }
12
13   ExecuteRequest(cgiDir, cgiCommand);
14 }
```

Figure 3.2: Example vulnerable code: simplified web server with a buffer overflow vulnerability. Line 7 is split into lines 8 and 9 to highlight the erroneous pointer arithmetic operation.

contain the substring `"\\.."` and then it concatenates `cgiDir` and `cgiCommand` to obtain the pathname of the executable to run. Unfortunately, there is a bound error in line 8: if the message is too long, the attacker can overwrite `cgiDir`, assuming the compiler generated a memory layout where `cgiDir` immediately follows `cgiCommand`. This allows the attacker to run any executable (for example, a command shell) with the arguments supplied in the request message. This is one of the most challenging types of attacks to detect because it is a non-control-data attack [37]: it does not violate control-flow integrity.

The system identifies at compile time lines 8 and 9 as containing potentially-dangerous pointer arithmetic, in this case adding `i` to pointer `cgiCommand` and indexing array `msg`, and inserts code to perform checks at runtime. It also inserts code to save the bounds of memory allocations in the bounds table. In the case of global variables such as `cgiCommand` and `cgiDir`, this code is run on program startup for each variable.

At runtime, the checks for the dangerous pointer arithmetic in line 8 use the source pointer holding the address of `cgiCommand` to look up in the table the bounds of the memory pointed to. The system then performs the pointer arithmetic operation, in this case computing index `i`, but before using the result to access memory, it checks if the result remains within bounds. If the resulting address is out-of-bounds, an exception is raised. By checking all pointer arithmetic, the Jones and Kelly approach maintains the invariant that every pointer used by the program is within bounds. That is why pointers can be assumed to be within bounds prior to every pointer arithmetic operation.

While not the case in this example, according to the C standard, an out-of-bounds pointer pointing one element past an array can be legally used in pointer arithmetic to produce an in-bounds pointer. In common practice, arbitrary out-of-bounds pointers

are used too. That is why the Jones and Kelly approach was subsequently modified [130] from raising an exception to marking out-of-bounds pointers and only raising an exception if they are dereferenced. Marked pointers must be handled specially to retrieve their proper bounds. This mechanism is described in detail in Section 3.5.

To reduce space and time overhead at runtime, the solutions presented in this dissertation, including BBC, perform a *safety* analysis to compute instructions and objects that are *safe*. A pointer arithmetic operation is safe if its result can be proven at compile-time to always be within bounds. The result of a safe pointer arithmetic operation is a safe pointer. Objects that are accessed only through safe pointers, or not accessed through pointers at all, are themselves safe. The system does not have to instrument safe pointer arithmetic operations, and does not have to enter safe objects into the bounds table. Variable `i` in the example of Figure 3.2 is safe, because it is never accessed through a pointer in the program, unless we consider the implicit stack-frame pointer used by the generated machine code to access local variables relative to the function's stack frame, in which case `i` is still safe, as the constant offsets used in pointer arithmetic accessing local variables are within bounds assuming the stack-frame pointer is not corrupted. (BBC protects the stack-frame pointer through its bounds checks, and Chapter 4 offers even stronger guarantees.)

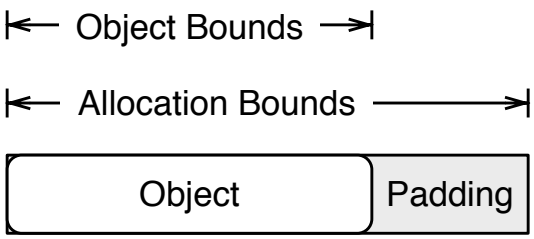


Figure 3.3: Baggy bounds encompass the entire memory allocated for an object, including any padding appended to the object. Note that for compound objects such as structures and arrays, only the bounds of the most outer object are considered.

Baggy bounds checking differs from prior work based on the Jones and Kelly approach in its runtime mechanisms, highlighted in Figure 3.1 with a dashed box. Instead of exact bounds, it enforces baggy bounds. As shown in Figure 3.3, baggy bounds checking enforces the allocation bounds which include the object and additional padding. The padding controls the size and memory alignment of allocations. This is used by BBC to improve performance by enabling a very compact representation for the bounds. Previous techniques recorded two values in the bounds table for every object: a pointer to the start of the object and its size, which together require at least eight bytes on 32-bit systems. Instead, BBC pads and aligns objects (including heap, local, and global variables) to powers of two, which enables using a single byte to encode bounds by storing the binary logarithm of the allocation size in the bounds table (using C-like notation):

```
e = log2(size)
```

Given this information and a pointer `p`, pointing possibly anywhere within a suitably aligned and padded object, BBC can recover the allocation size and a pointer to the start of the allocation with:

```
size = 1 << e
base = p & ~(size - 1)
```

Next, I discuss how BBC satisfies the requirements for protection, performance, and backwards compatibility; how it can support pointers one or more bytes beyond the target object; and additional implementation details followed by an experimental evaluation.

3.2 Protection

BBC can detect bounds violations across allocation bounds, but silently tolerates violations of the object bounds that stay within the allocation bounds. For BBC to be secure, it is crucial that such undetected violations cannot breach security.

It is clear that the overwrite targets of Section 2.3, including return addresses, function pointers or other security critical data, cannot be corrupted since accesses to the padding area cannot overwrite other objects or runtime metadata. Also, reads cannot be exploited to leak sensitive information, such as a password, from another live object.

However, allowing reads from an uninitialised padding area can access information from freed objects that are no longer in use. Such reads could leak secrets or access data whose contents are controlled by the attacker. Recall from Section 2.3 that pointer reads from attacker-controlled or uninitialised memory are particularly dangerous. This vulnerability is prevented by clearing the padding on memory allocation.

3.3 Performance

Baggy bounds enable two performance improvements over traditional bounds checking. First, the compact bounds representation can be used to simplify and optimise the data structure mapping memory addresses to bounds. Second, baggy bounds can be checked more efficiently using bitwise operations without explicitly recovering both bounds to perform two arithmetic comparisons, saving both machine registers and instructions. These optimisation opportunities come at the cost of increased memory usage due to the extra padding introduced by baggy bounds, and increased processing due to having to zero this extra padding on memory allocation. As we shall see in the experimental evaluation, however, simplifying the data structures can save considerable space over previous solutions, as well as increase performance, making the system's space and time overheads very competitive.

3.3.1 Efficient bounds lookup

The Jones and Kelly approach uses a *bounds table* to map pointers to their object bounds. Previous solutions [82, 130, 57] have implemented the bounds table using a

splay tree. BBC, on the other hand, implements the bounds table using a contiguous array.

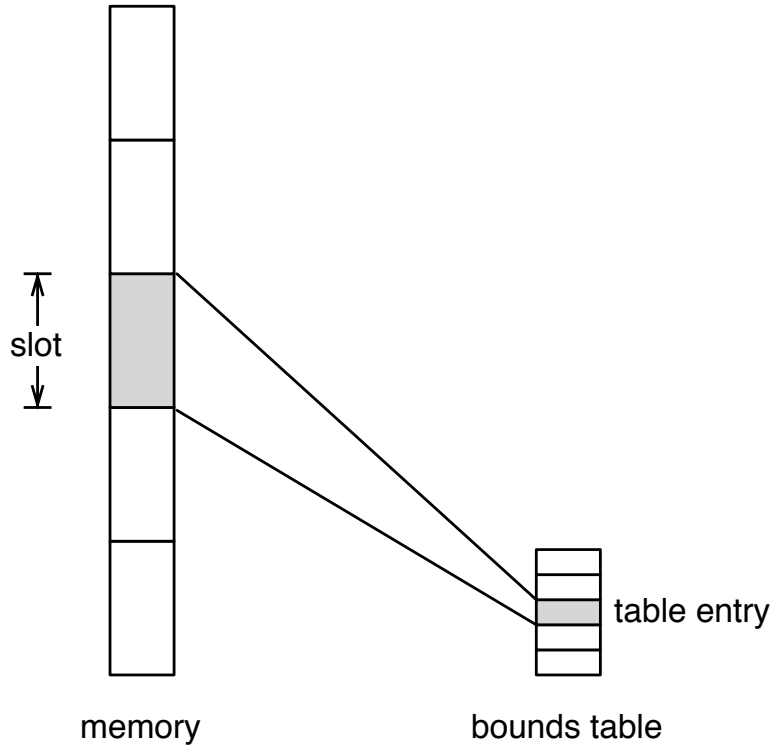


Figure 3.4: The bounds table can be kept small by partitioning memory into slots.

For baggy bounds, this array can be small because each entry uses a single byte. Moreover, by partitioning memory into aligned *slots* of *slotsize* bytes, as shown in Figure 3.4, and aligning objects to slot boundaries so that no two objects share a slot, the bounds table can have one entry per slot rather than one per byte. Now the space overhead of the table is $1/slotsize$, which can be tuned to balance between memory waste due to padding and table size.

Accesses to the table are fast because it is an array. To obtain a pointer to the table entry corresponding to an address, $\lfloor \frac{address}{slotsize} \rfloor$ is computed by right-shifting the address by the constant $log_2(slotsize)$, and is added to the constant base address of the table. The resulting pointer can be used to retrieve the bounds information with a single memory access, instead of having to traverse and splay a splay tree (as in previous solutions [82, 130]).

3.3.2 Efficient bounds checks

In addition to improving table lookup, baggy bounds also enable more efficient bounds checks. In general, bounds checking the result `q` of pointer arithmetic on `p` involves two comparisons: one against the lower bound and one against the upper bound, as shown in Figure 3.5.

Pointer arithmetic operation

```
char *q = p + i;
```

Explicit bounds check

```
size = 1 << table[p>>slotsize];
base = p & ~(size - 1);

is_invalid = q < base || q >= base + size;
```

Optimised bounds check

```
is_invalid = ((p^q) >> table[p>>slotsize]) != 0;
```

Figure 3.5: Baggy bounds enables optimised bounds checks: we can verify that pointer `q` derived from pointer `p` is within bounds by simply checking that `p` and `q` have the same prefix with only the `e` least significant bits modified, where `e` is the binary logarithm of the allocation size. The check can be implemented using efficient unsigned shift operations.

Baggy bounds, however, enable an optimised bounds check that does not even need to compute the lower and upper bounds. It uses directly the value of `p` and the value of the binary logarithm of the allocation size, `e`, retrieved from the bounds table. The constraints on allocation size and alignment ensure that `q` is within the allocation bounds if it differs from `p` only in the `e` least significant bits. Therefore, it is sufficient to XOR `p` with `q`, right-shift the result by `e` and check for zero, as shown in Figure 3.5.

Furthermore, for pointers `q` where `sizeof(*q) > 1`, we also need to check that `(char *) q + sizeof(*q) - 1` is within bounds to prevent an access to `*q` from crossing the end of the object and the allocation bounds. Baggy bounds checking can avoid this extra check if `q` points to a built-in type. Aligned accesses to these types cannot overlap an allocation boundary because their size is a power of two and is less than *slotsize* (assuming a sufficiently large choice of *slotsize* such as 16). In the absence of casts, all accesses to built-in types generated by the compiler are aligned. Enforcing alignment is cheap, because only casts have to be checked. When checking accesses to structures not satisfying these constraints, both checks are performed.

3.4 Interoperability

As discussed in Section 1.3, BBC must work even when instrumented code is linked against libraries that are not instrumented. Protection must degrade gracefully with uninstrumented libraries, and instrumented code must not raise false positives due to objects allocated in uninstrumented libraries. This form of interoperability is important because some libraries are distributed in binary form only.

Library code works with BBC because the size of pointers, and therefore the memory layout of structures, does not change. However, it is also necessary to ensure graceful degradation when bounds are missing, so that instrumented code can access

memory allocated in an uninstrumented library without raising an alert. Such interoperability is achieved by using the maximum allocation size (*e.g.* the entire address-space) as the default value for bounds-table entries, and making sure that instrumented code restores the default value after deallocating an object. This ensures that table entries for local and global variables of uninstrumented libraries default to the value corresponding to a maximal allocation, allowing instrumented code to perform checks as normal when accessing such memory, but effectively disabling protection for such objects. Heap allocations in library code, on the other hand, can be intercepted at link time, or the system allocator can be replaced, to pad and align them. This would enable bounds checks when accessing library-allocated dynamic memory, but is not supported in the current prototype; instead, we rely again on the default maximal bounds for interoperability alone.

3.5 Out-of-bounds pointers

A complication arises from C pointers legally pointing outside their object's bounds. Such pointers should not be dereferenced but can be compared and used in pointer arithmetic that can eventually result in a valid pointer that may be dereferenced by the program. Out-of-bounds pointers challenge the Jones and Kelly approach because it relies on in-bounds pointers to retrieve object bounds.

The C standard only allows out-of-bounds pointers to one element past the end of an array. Jones and Kelly [82] support these legal out-of-bounds pointers by adding one byte of padding to every object. I did not use this technique because it interacts poorly with the constraints on allocation sizes: adding one byte to an allocation can double the allocated size in the frequent case where the requested allocation size is already a power of two.

To make things worse, many programs violate the C standard and generate illegal but harmless out-of-bounds pointers that are never dereferenced. Examples include faking a base one array by decrementing the pointer returned by `malloc` and other equally tasteless uses. CRED [130] improved on the Jones and Kelly bounds checker by tracking such pointers using another auxiliary data structure. I did not use this approach because it adds overhead to deallocations of heap and local objects: when an object is deallocated, this auxiliary data structure must be searched to remove entries tracking out-of-bounds pointers to the object. Worse, entries in this auxiliary data structure may accumulate indefinitely until their referent object is deallocated—potentially never.

My solution can directly handle out-of-bounds pointers within $slotsize/2$ bytes from the original object as follows. First, it marks out-of-bounds pointers to prevent them from being dereferenced, relying on the memory protection hardware to prevent dereferences by setting the top bit in these pointers (as in [57]) and by restricting the program to the lower half of the address space (this is the case in 64-bit operating systems and is also often the default for 32-bit operating systems such as Microsoft Windows). The unmarked out-of-bounds pointer can be recovered by clearing the top bit.

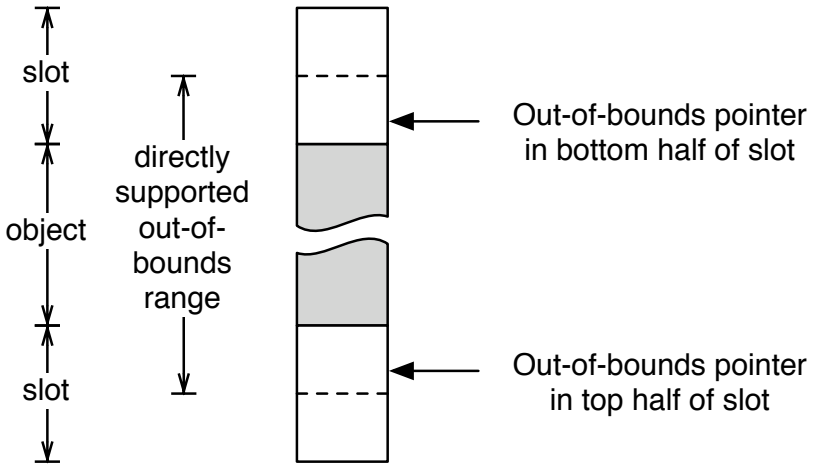


Figure 3.6: We can tell whether a pointer that is out-of-bounds by less than *slotsize*/2 is below or above an allocation. This lets us correctly adjust it to get a pointer to the object by respectively adding or subtracting *slotsize*.

The next challenge is to recover a pointer to the referent object from the out-of-bounds pointer without resorting to an additional data structure. This is possible for the common case of *near* out-of-bounds pointers pointing at most *slotsize*/2 bytes before or after the allocation bounds. Since the allocation bounds are aligned on slot boundaries, a near out-of-bounds pointer is below or above the allocation depending on whether it lies in the top or bottom half of an adjacent memory slot respectively, as illustrated in Figure 3.6. Thus, a pointer within the referent object can be recovered from a near out-of-bounds pointer by adding or subtracting *slotsize* bytes accordingly. This technique cannot handle out-of-bounds pointers more than *slotsize*/2 bytes outside the original allocation but, in Section 3.9.2, I show how to take advantage of the spare bits in pointers on 64-bit architectures to increase this range. It is also possible to support the remaining cases of out-of-bounds pointers using previous techniques [130, 104]. The advantage of using my mechanism for programs with non-standard out-of-bounds pointers is that the problems of previous techniques (including runtime and memory overheads) are limited to a small (or zero) subset of out-of-bounds pointers; of course, programs that stick to the C standard suffer no such problems.

Moreover, pointers that are provably not dereferenced can be allowed to go out-of-bounds. This can be useful for supporting idioms where pointers go wildly out-of-bounds in the final iteration of a loop, *e.g.*:

```
1 for (i = 0; ; i<<=2)
2     if (buf + i >= buf + size)
3         break;
```

Relational operators (*e.g.* >=) must be instrumented to support comparing an out-of-bounds pointer with an in-bounds one: the instrumentation must clear the top bit of the pointers before comparing them. Interestingly, less instrumentation would be necessary to support the legal case of one element past the end of an array only, because setting the top bit would not affect the unsigned inequality testing used for

pointers. Pointer differences have to be instrumented in a similar way to inequality testing—this cannot be avoided. There is no need, however, to instrument equality testing because the result will be correct whether the pointers are out-of-bounds or not.

Like previous bounds checking solutions [82, 130, 57] based on the Jones and Kelly approach, BBC has difficulties sharing out-of-bounds pointers with uninstrumented code. However, out-of-bounds pointers passed as function arguments, *e.g.* to denote the end of arrays, can be supported by unmarking pointers before calling foreign functions. Out-of-bounds pointers used for other purposes, *e.g.* stored in shared data structures, remain problematic. However, previous work [130] did not encounter such problems in several million lines of code, and nor has my own evaluation.

3.6 Static analysis

Bounds checking has relied heavily on static analysis to optimise performance. Checks can be eliminated when it can be statically determined that a pointer is *safe*, *i.e.* always within bounds, or that a check is made redundant by a previous check. Furthermore, checks (or even just the bounds lookup) can be hoisted out of loops.

Rather than implementing a sophisticated analysis as in previous work, I focused on making checks efficient. Nevertheless, my prototype implements a simple intraprocedural pointer-range analysis to detect pointers that are always within bounds, and I investigate a transformation to hoist checks out of loops.

3.6.1 Pointer-range analysis

The pointer-range analysis is a simplified version of the one described in [172]. It collects sizes of aggregate objects (*i.e.* `structs`) and arrays that are known statically. Then it uses data-flow analysis to compute the minimum size of the objects each pointer can refer to and the maximum offset of the pointer into these objects, to determine whether accesses through a pointer are always within bounds. When the analysis cannot compute this information or the offset can be negative, it conservatively assumes a minimum size of zero. The implementation can track constant offsets and offsets that can be bounded using Phoenix's built-in value range information for numeric variables. Given information about the minimum object sizes, the maximum pointer offsets within these objects, and the size of the intended write, the analysis checks if writes through the pointer are always in bounds. If they are, the corresponding access is marked safe and does not have to be checked at runtime.

3.6.2 Safe objects

Aligning local variables of arbitrary sizes at runtime is expensive, because it is implemented by over-allocating stack memory on function entry and aligning the frame pointer. In practice, this also inhibts the frame-pointer-ommission optimisation, which is important for register-starved architectures such as the 32-bit x86. Static analysis helps reduce the number of functions that require stack-frame alignment, by finding

objects that cannot be accessed in unsafe ways. These are called *safe* objects, since every access to them is itself safe. The current prototype only pads and aligns local variables that are indexed unsafely in the enclosing function, or whose address is taken, and therefore possibly leaked to other functions that may use them unsafely. These variables are called *unsafe*.

3.6.3 Loop cloning

Since optimisation of inner loops can have a dramatic impact on performance, I experimented with hoisting bounds table lookups out of loops when all accesses inside a loop body are to the same object. Unfortunately, performance did not improve significantly, probably because my bounds lookups are inexpensive and hoisting can adversely affect register allocation.

Hoisting the whole check out of a loop is preferable, and possible when static analysis can determine symbolic bounds on the pointer values in the loop body. However, hoisting the check out is only possible if the analysis can determine that these bounds are guaranteed to be reached in every execution. Figure 3.7 shows an example where the loop bounds are easy to determine but the loop may terminate before reaching the upper bound. Hoisting the check out would trigger a false alarm in runs where the loop breaks before violating the bounds.

I experimented with generating two versions of the loop code, one with checks and one without, and adding code to switch between the two versions on loop entry. In the example of Figure 3.7, the transformed code looks up the bounds of p, and if n does not exceed the size, it runs the unchecked version of the loop; otherwise, it runs the checked version to avoid a false positive.

3.7 Instrumentation

3.7.1 Bounds table

In implementing the bounds table, I chose a *slotsize* of 16 bytes which is small enough to avoid penalising small allocations but large enough to keep the table memory use low. Therefore, $1/16^{th}$ of the virtual address space is reserved for the bounds table. Since pages are allocated to the table on demand, this increases memory utilisation by only 6.25%. On program startup, the address space required for the bounds table is reserved, and a vectored exception handler (a Windows mechanism similar to UNIX signal handlers) is installed to capture accesses to unallocated pages and allocate missing table pages on demand. All the bytes in these pages are initialised by the handler to the value 31, representing bounds encompassing all the memory addressable by BBC programs on the x86 (an allocation size of 2^{31} at base address 0). This prevents out-of-bounds errors when instrumented code accesses memory allocated by uninstrumented code, as discussed in Section 3.4. The memory alignment used by the system memory allocator on 32-bit Windows is 8 bytes, but a *slotsize* of 16 bytes could be supported using a custom memory allocator.

The table can be placed in any fixed memory location. The current prototype places the base of the table at address 40000000h. This has the downside that this 32-bit con-

```
1  for (i = 0; i < n; i++) {
2     if (p[i] == 0) break;
3     ASSERT(IN_BOUNDS(p, &p[i]));
4     p[i] = 0;
5  }
```

```
1  if (IN_BOUNDS(p, &p[n-1])) {
2     for (i = 0; i < n; i++) {
3        if (p[i] == 0) break;
4        p[i] = 0;
5     }
6  } else {
7     for (i = 0; i < n; i++) {
8        if (p[i] == 0) break;
9        ASSERT(IN_BOUNDS(p, &p[i]));
10       p[i] = 0;
11    }
12 }
```

Figure 3.7: The compiler's range analysis can determine that the range of variable i is at most $0 \ldots n-1$. However, the loop may exit before i reaches $n-1$. To prevent erroneously raising an alarm in that case, the transformed code falls back to an instrumented version of the loop if the hoisted check fails.

stant has to be encoded in the instrumentation, increasing code bloat. Using a base address of 0h reduces the number of bytes needed to encode the instructions that access the table by omitting the 32-bit constant. This arrangement can still accommodate the null pointer dereference landing zone at address 0h (a protected memory range that ensures null pointer dereferences raise an access error). The lower part of the table can be access protected because it is the unused image of the table memory itself. I experimented with placing the base of the table at address 0h; however, it had little impact on the runtime overhead in practice.

3.7.2 Memory layout

BBC must ensure that all memory allocations, including dynamic, static, and automatic variables, are aligned and padded appropriately.

For heap allocations, a custom memory allocator is used to satisfy the size and alignment constraints. A binary buddy allocator was chosen that provides low external fragmentation at the cost of internal fragmentation due to rounding allocation sizes to powers of two. The latter is, in general, a shortcoming, but is exactly what is needed for baggy bounds. The buddy allocator implementation supports a minimum allocation size of 16 bytes, which matches the chosen *slotsize* parameter, to ensure that no two objects share the same slot. BBC replaces calls in the program to the stan-

dard memory allocation functions with calls to implementations based on the buddy allocator.

The stack frames of functions that contain unsafe local variables are aligned at runtime by enlarging the stack frame and aligning the frame pointer, while global and static variables are aligned and padded at compile time. Most compilers support aligning variables using special declaration attributes; BBC uses the same mechanism implicitly.

Unsafe function arguments need special treatment, because padding and aligning them would violate the calling convention. Instead, they are copied by the function prologue to appropriately aligned and padded local variables and all references in the function body are changed to use their copies (except for uses by `va_list` that need the address of the last explicit argument to correctly extract subsequent arguments). This preserves the calling convention while enabling bounds checking for function arguments accessed in unsafe ways.

Unfortunately, the Windows runtime cannot align stack objects to more than 8k nor static objects to more than 4k (configurable using the `/ALIGN` linker switch). To remove this limitation, large automatic and static allocations could be replaced with dynamic allocations, or the language runtime could be modified to support larger alignment requirements. Instead, the current prototype deals with this by setting the bounds table entries for such objects to 31, effectively disabling checks (but the performance impact of the checks remains).

3.7.3 Maintaining the bounds table

BBC also adds instrumentation to maintain the bounds table on memory allocation and deallocation. The buddy allocator's functions set the bounds table entries for heap allocations. There is no need, however, to reset bounds table entries on heap deallocations for interoperability, because they are only reused within the buddy system. Function prologues are instrumented to update the appropriate bounds table entries and function epilogues to reset table entries to 31, to ensure interoperability with uninstrumented code reusing stack memory. Bounds table entries for static variables are initialised on program start up by an initialisation routine using object address and size information stored in a special section of the object files compiled with BBC.

3.7.4 Clearing the padding

Finally, BBC adds instrumentation to zero the padding on memory allocation and protect against reads from uninitialised memory as discussed in Section 3.2. The memory allocation functions zero the padding area after dynamic objects, and the function prologues zero the padding of local variables.

Zeroing the padding can increase space and time overhead for large padding areas whose memory pages would not be otherwise touched by the program. This overhead can be avoided by relying on the operating system to zero-initialise pages on demand and avoiding zeroing untouched pages. Similar performance issues related to clearing large amounts of memory are discussed in [38]. Moreover, memory allo-

cators (including the BBC buddy allocator) perform similar optimisations for `calloc` in order to avoid unnecessary page accesses to over-provisioned memory allocations that would not be touched otherwise. The current prototype supports this for heap allocations by reusing the mechanism used for `calloc`.

3.7.5 Inserted checks

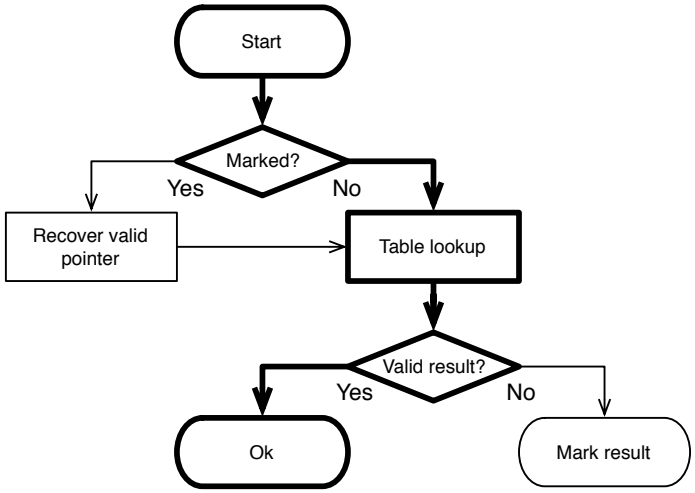
Checks are added for all pointer arithmetic operations, including array or pointer indexing, but, following [57], pointer arithmetic operations generated by the compiler to access scalar fields in structures are not instrumented. This facilitates a direct comparison with that work. My prototype could be easily modified to perform these checks, *e.g.* using the technique described in [51].

When checking pointer arithmetic, source pointers marked as out-of-bounds require special treatment to recover a valid pointer to the referent object. A critical optimisation avoids the cost of explicitly checking for out-of-bounds pointer in the common case of in-bounds pointers. Instead of explicitly checking if a pointer is marked out-of-bounds in the fast path (Figure 3.8(a)), out-of-bounds pointers are made to trigger a bounds error (Figure 3.8(b)). This is achieved by zeroing the bounds table entries for out-of-bounds pointers. Since out-of-bounds pointers have their top bit set, mapping all virtual memory pages in the top half of the bounds table to a shared zero page ensures that the error handler is invoked on any arithmetic operation on a pointer marked out-of-bounds. This handler checks whether the source pointer is marked, to determine whether it is a genuine out-of-bounds result, and repeats the check as required.

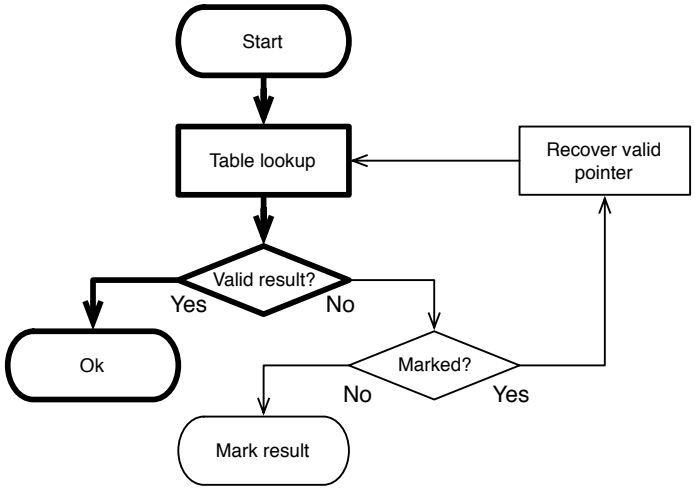
Figure 3.9 shows the `x86` code sequence inserted before the vulnerable pointer arithmetic operation. First, the source pointer, `cgiCommand`, is right shifted (line 3) to obtain the index of the bounds table entry for the corresponding memory slot. Next e, the binary logarithm of the allocation size, is loaded from the bounds table into register `al` (line 4). Then `p`, the result of the pointer arithmetic, is XORed with `cgiCommand`, the source pointer, and right shifted by `al` to discard the bottom bits (lines 5–7). If `cgiCommand` and `p` are both within the allocation bounds they can only differ in the e least significant bits (as explained in Section 3.3.2). So if the zero flag is set, `p` is within the allocation bounds. Otherwise, the `slowPath` function is called.

The `slowPath` function starts by checking if the source pointer has been marked out-of-bounds. If true, it obtains the referent object as described in Section 3.5, resets the top bit of `p`, and returns the result if it is within bounds. Otherwise, as in the attack of the example, the result is out-of-bounds. If the result is out-of-bounds by more than half a slot, the function signals an error. Otherwise, it marks the result out-of-bounds and returns it. Any attempt to dereference the returned pointer will trigger an exception. To avoid disturbing register allocation in the fast path, the rarely called `slowPath` function uses a special calling convention that saves and restores all registers.

As discussed in Section 3.7.5, `sizeof(*p)` is added to the result followed by a second check if the pointer is not a pointer to a built-in type (although this could also be avoided if the size of the structure is small). In the example, the extra check is avoided because `cgiCommand` is a `char *`.



(a) Unoptimised out-of-bounds checking.



(b) Optimised out-of-bounds checking.

Figure 3.8: The optimised flow in (b) requires only one comparison in the fast path shown with bold lines vs. two comparisons for the unoptimised case (a).

Similar to previous work, bounds-checking wrappers are provided for standard C library functions that accept pointers, such as `strcpy` and `memcpy`. Calls to these functions are replaced with calls to their wrappers during instrumentation.

3.8 Experimental evaluation

This section evaluates the performance of BBC using CPU-bound benchmarks written in C, and its effectiveness in preventing attacks from a buffer overflow suite. The practicality of the solution is further confirmed by building and measuring the performance of real-world security-critical code.

pointer arithmetic

```
1  p = cgiCommand + i;
```

bounds lookup

```
2  mov eax, cgiCommand
3  shr eax, 4
4  mov al, byte ptr [TABLE+eax]
```

bounds check

```
5  mov ebx, cgiCommand
6  xor ebx, p
7  shr ebx, al
8  jz ok
9    p = slowPath(cgiCommand, p)
10 ok:
```

Figure 3.9: Code sequence inserted to check unsafe pointer arithmetic.

3.8.1 Performance

I evaluated the time and peak-memory overhead of BBC using the Olden benchmarks [31] and the SPEC CINT2000 [156] integer benchmarks. I chose these benchmarks to allow a comparison against results reported for some other solutions [57, 170, 104]. In addition, to enable a more detailed comparison with splay-tree-based approaches—including measuring their space overhead—I implemented a BBC variant which uses the splay tree code from previous systems [82, 130]. This implementation uses the standard allocator and lacks support for illegal out-of-bounds pointers, but instruments the same operations as BBC. All benchmarks were compiled with the Phoenix compiler using `/O2` optimisation level and run on a 2.33 GHz Intel Core 2 Duo processor with 2 GB of RAM. For each experiment, I present the average of 3 runs; the variance was negligible.

I did not run `eon` from SPEC CINT2000 because it uses C++ features which are not supported in the current prototype, such as operator `new`. For the splay-tree-based implementation only, I did not run `vpr` due to the lack of support for illegal out-of-bounds pointers. I also could not run `gcc` because of code that subtracted a pointer from a NULL pointer and subtracted the result from NULL again to recover the pointer. Running this would require more comprehensive support for out-of-bounds pointers (such as that described in [130]).

I made the following modifications to some of the benchmarks: First, I modified `parser` from SPEC CINT2000 to fix an overflow that triggered a bound error when using the splay tree. It did not trigger an error with baggy bounds checking because in the benchmark run, the overflow was entirely contained in the allocation. The unchecked program also survived the bug because the object was small enough for the overflow to be contained even in the padding added by the standard allocator.

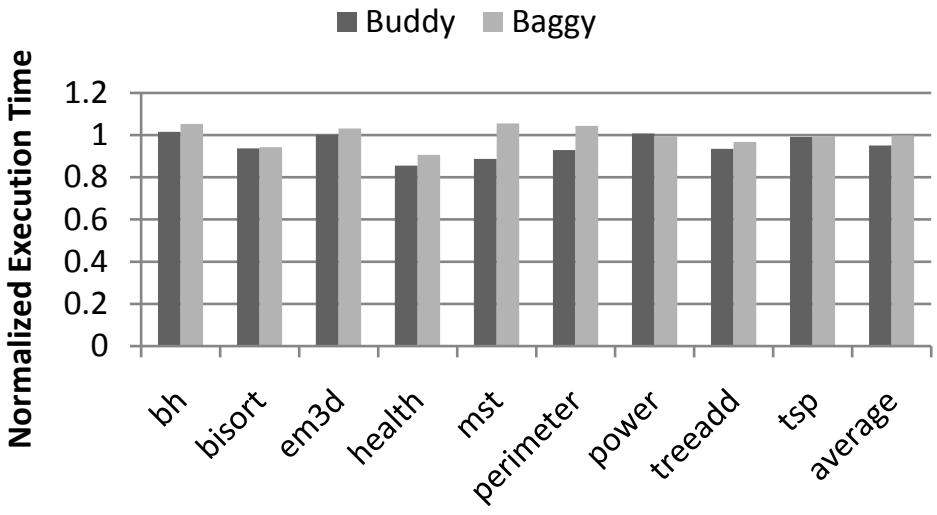


Figure 3.10: Execution time for the Olden benchmarks using the buddy allocator vs. the full BBC system, normalised by the execution time using the standard system allocator without instrumentation.

Second, I had to modify `perlbmk` by changing two lines to prevent an out-of-bounds arithmetic operation whose result is never used and `gap` by changing 5 lines to avoid an out-of-bounds pointer. Both cases can be handled by the extension described in Section 3.9, but are not covered by the small out-of-bounds range supported by the 32-bit implementation (or the lack of support by the splay-tree-based implementation).

Finally, I modified `mst` from Olden to disable a custom memory allocator. This illustrates a limitation in backwards-compatibility where programs require some tweaking to take full advantage of checking. In this case merely changing a preprocessor definition was sufficient. Also, such programs would still work without tweaks, albeit without protection against heap-based memory-errors. This limitation is shared with all systems offering protection at the memory block level [82, 130, 170, 57, 5].

I first ran the benchmarks replacing the standard allocator with the buddy allocator to isolate its effects on performance, and then I ran them using BBC. For the Olden benchmarks, Figure 3.10 shows the execution time and Figure 3.11 the peak memory usage.

Figure 3.10 shows that some benchmarks in the Olden suite (`mst`, `health`) run significantly faster with the buddy allocator than with the standard one. These benchmarks are memory intensive and any memory savings reflect on the running time. Figure 3.11 shows that the buddy system uses less memory for these than the standard allocator. This is because these benchmarks contain numerous small allocations for which the padding to satisfy alignment requirements and the per-allocation metadata used by the standard allocator exceed the internal fragmentation of the buddy system.

This means that the average time overhead of the full system across the entire Olden suite is zero, because the positive effects of using the buddy allocator mask the costs of checks. The time overhead of the checks alone as measured against the buddy allocator as a baseline is 6%. The overhead of the fastest previous bounds-

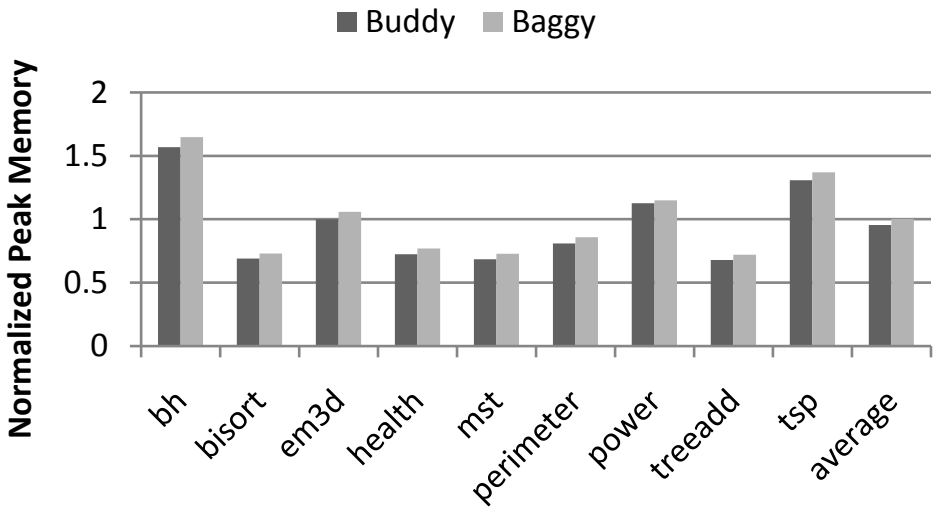


Figure 3.11: Peak memory use with the buddy allocator alone vs. the full BBC system for the Olden benchmarks, normalised by peak memory using the standard allocator without instrumentation.

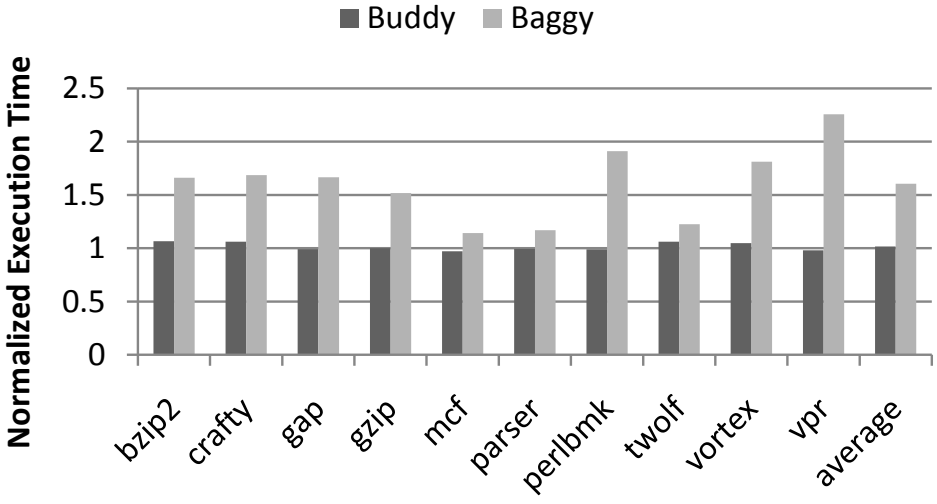


Figure 3.12: Execution time for SPEC CINT2000 benchmarks using the buddy allocator vs. the full BBC system, normalised by the execution time using the standard system allocator without instrumentation.

checking system [57] on the same benchmarks and offering same protection (modulo allocation vs. object bounds) is 12%. Moreover, their system uses a technique (pool allocation) which could be combined with BBC. Based on the breakdown of results reported in [57], their overhead measured against a baseline using just pool allocation is 15%, and it seems more reasonable to compare these two numbers, as both the buddy allocator and pool allocation can be in principle applied independently on either system.

Next I measured the system using the SPEC CINT2000 benchmarks. Figures 3.12 and 3.13 show the time and space overheads for SPEC CINT2000 benchmarks.

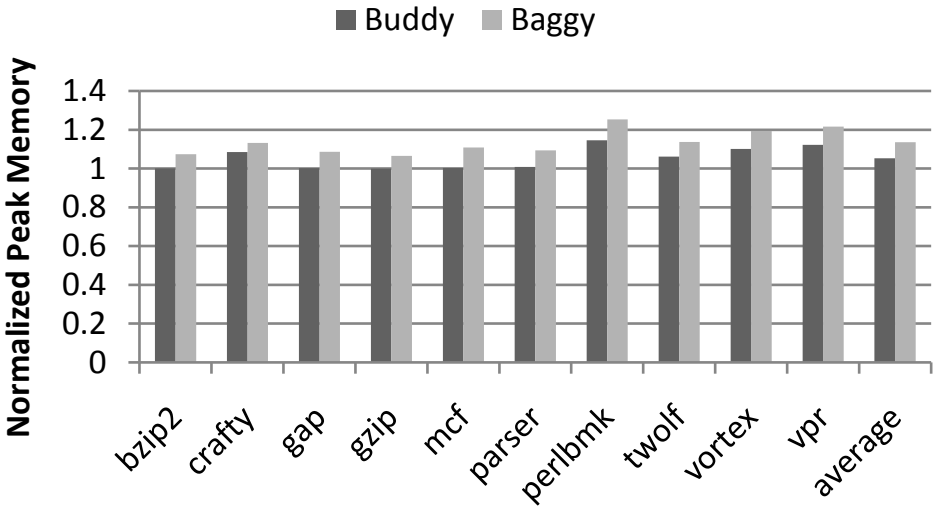


Figure 3.13: Peak memory use with the buddy allocator alone vs. the full BBC system for SPEC CINT2000 benchmarks, normalised by peak memory using the standard allocator without instrumentation.

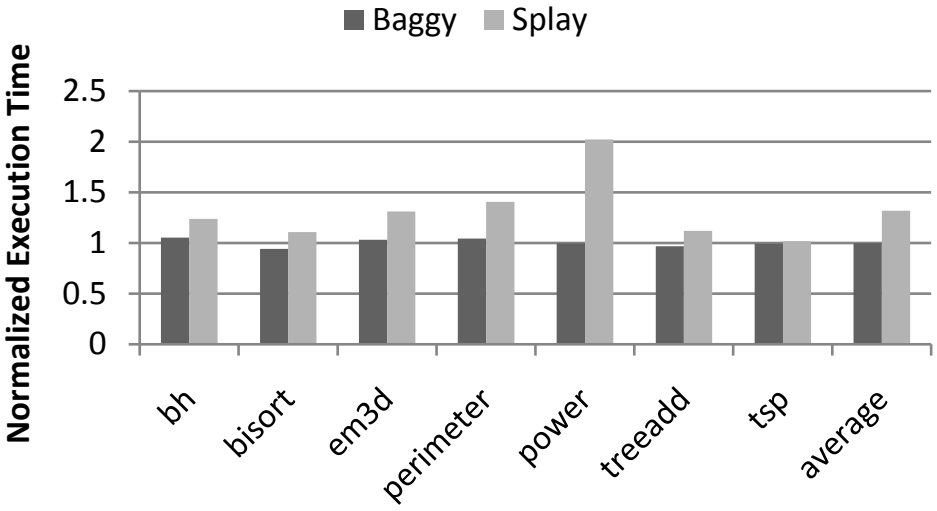


Figure 3.14: Execution time of baggy bounds checking vs. using a splay tree for the Olden benchmark suite, normalised by the execution time using the standard system allocator without instrumentation. Benchmarks `mst` and `health` used too much memory and thrashed so their execution times are excluded.

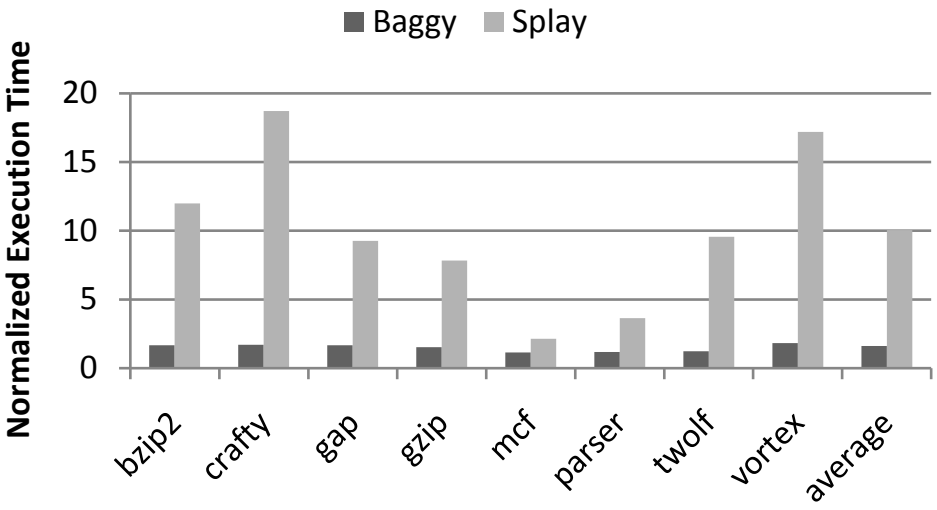


Figure 3.15: Execution time of baggy bounds checking vs. using a splay tree for SPEC CINT2000 benchmarks, normalised by the execution time using the standard system allocator without instrumentation.

The use of the buddy allocator has little effect on performance in general. The average runtime overhead of the full system with the benchmarks from SPEC CINT2000 is 60%. `vpr` has the highest overhead of 127% because its frequent use of illegal pointers to fake base-one arrays invokes the slow path. I observed that adjusting the allocator to pad each allocation with 8 bytes from below decreases the time overhead to 53% with only 5% added to the memory usage, although in general I did not investigate tuning the benchmarks like this. Interestingly, the overhead for `mcf` is a mere 16% compared to the 185% in [170] but the overhead of `gzip` is 55% compared to 15% in [170]. Such differences in performance are due to different levels of protection such as checking structure field indexing and checking dereferences, the effectiveness of different static analysis implementations in optimising away checks, and the different compilers used.

To isolate these effects, I also measured BBC using the standard memory allocator and the splay tree implementation from previous systems [82, 130]. Figure 3.14 shows the time overhead for baggy bounds versus using a splay tree for the Olden benchmarks. The splay tree runs out of physical memory for the last two Olden benchmarks (`mst`, `health`) and slows down to a crawl, so I exclude them from the average of 30% for the splay tree. Figure 3.15 compares the time overhead against using a splay tree for the SPEC CINT2000 benchmarks. The overhead of the splay tree exceeds 100% for all benchmarks, with an average of 900% compared to the average of 60% for baggy bounds checking.

Perhaps the most interesting result in the evaluation was space overhead. Previous solutions [82, 130, 57] do not report on the memory overheads of using splay trees, so I measured the memory overhead of BBC when using splay trees and compared it with the memory overhead of BBC when using the baggy-bounds buddy allocator. Figure 3.16 shows that BBC has negligible memory overhead for Olden, as opposed to the splay tree version's 170% overhead. Interestingly Olden's numerous small al-

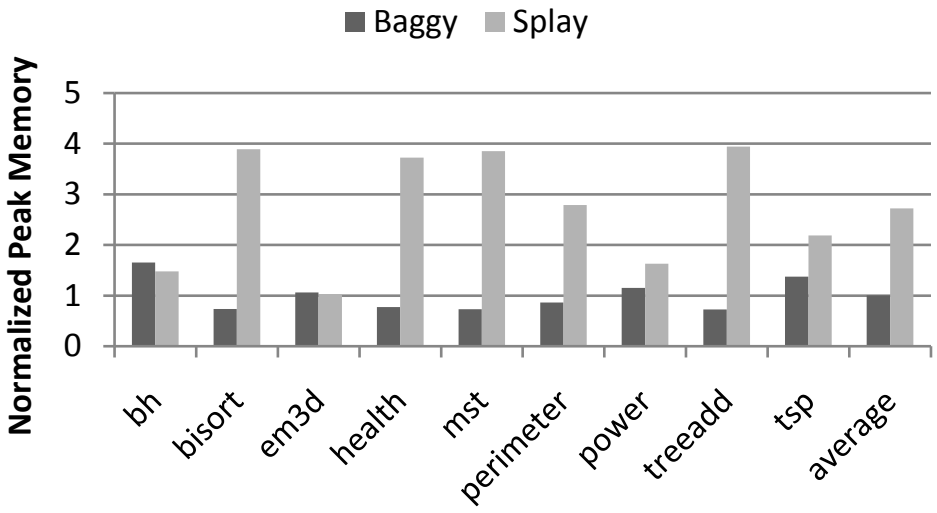


Figure 3.16: Peak memory use of baggy bounds checking vs. using a splay tree for the Olden benchmark suite, normalised by peak memory using the standard allocator without instrumentation.

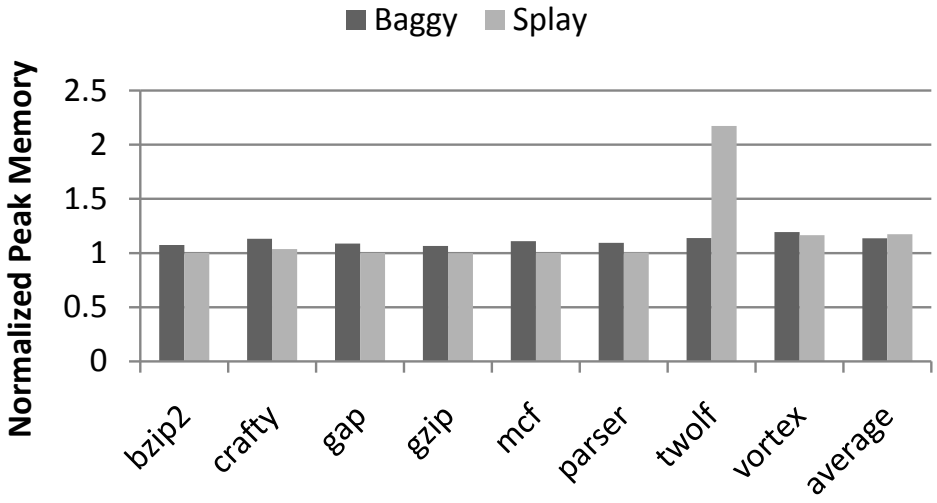


Figure 3.17: Peak memory use of baggy bounds checking vs. using a splay tree for SPEC CINT2000 benchmarks, normalised by peak memory using the standard allocator without instrumentation.

locations demonstrate the splay tree's worst case memory usage by taking up more space for the entries than for the objects.

On the other hand, Figure 3.17 shows that the splay tree version's space overhead for most SPEC CINT2000 benchmarks is very low. The overhead of BBC, however, is even less (15% vs. 20%). Furthermore, the potential worst case of double the memory was not encountered for baggy bounds in any of the experiments, while the splay tree did exhibit greater than 100% overhead for one benchmark (`twolf`).

The memory overhead is also low, as expected, compared to approaches that track metadata for each pointer. For example, Xu *et al.* [170] report 331% for Olden, and Nagarakatte *et al.* [104] report an average of 87% using a hash-table (and 64% using a contiguous array) for Olden and a subset of SPEC CINT and SPEC CFP. For the pointer-intensive Olden benchmarks alone, their overhead increases to more than about 260% (or about 170% using the array). These systems suffer high memory overheads per pointer to provide temporal protection [170] or sub-object protection [104].

3.8.2 Effectiveness

I evaluated the effectiveness of BBC in preventing buffer overflows using the benchmark suite from [166]. The attacks required tuning to have any chance of success, because BBC changes the stack frame layout and copies unsafe function arguments to local variables while the benchmarks use the address of the first function argument to find the location of the return address they aim to overwrite.

BBC prevented 17 out of 18 buffer overflows in the suite. It failed, however, to prevent the overflow of an array inside a structure from overwriting a pointer inside the same structure. This pointer was used to overwrite arbitrary memory, and if it was a function pointer, it could have been used to directly execute arbitrary code. This limitation is shared with other systems that detect memory errors at the level of memory blocks [82, 130, 170, 57]. However, systems storing a base pointer and size out-of-band can provide a second level of defence for overwritten pointers because the associated bounds remain intact and can prevent violations when the overwritten pointer is used. Unfortunately, this is not the case for baggy bounds, because, rather than a pair of base address and size, it stores the bounds relative to the pointer value.

3.8.3 Large security critical applications

Finally, to verify the usability of BBC, even with the current prototype's limited support for out-of-bounds pointers, I built and measured a few additional large and security-critical applications. Table 3.1 lists the total number of lines compiled in all experiments.

I built the OpenSSL toolkit version 0.9.8k [115] that is comprised of about 400 KSLOC, and executed its test suite measuring a 10% time and an 11% memory overhead.

Then I built and measured two web servers, Apache [152] and NullHTTPd [111]. Running NullHTTPd revealed three bounds violations similar to, and including, the one reported in [32]. I used the Apache benchmark utility to compare the throughput over a LAN connection of the instrumented and uninstrumented versions of both

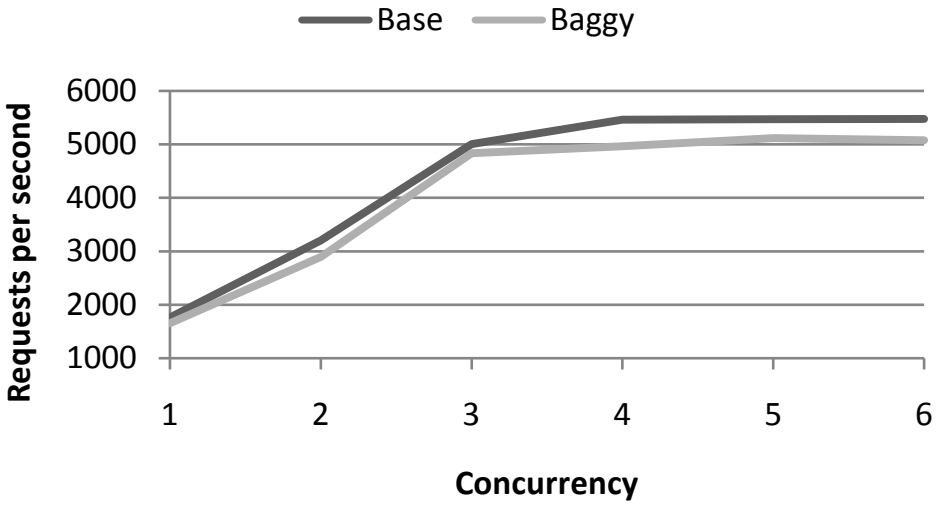


Figure 3.18: Throughput of Apache web server for varying numbers of concurrent requests.

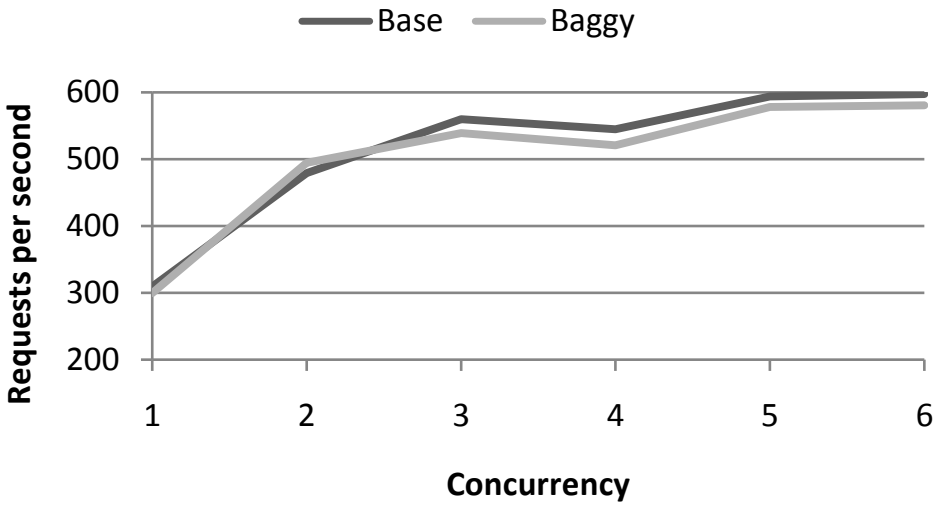


Figure 3.19: Throughput of NullHTTPd web server for varying numbers of concurrent requests.

web servers. I managed to saturate the CPU by using the keep-alive option of the benchmarking utility to reuse connections for subsequent requests. I issued repeated requests for the servers' default pages and varied the number of concurrent clients until the throughput of the uninstrumented version levelled off (Figures 3.18 and 3.19). Then I verified that the server's CPU was saturated, and measured a throughput decrease of 8% for Apache and 3% for NullHTTPd.

Finally, I built `libpng`, a notoriously vulnerability-prone library for processing images in the PNG file format [26]. `Libpng` is used by many applications to display images. I successfully ran its test program for 1000 PNG files between 1–2kB found on a desktop machine, and measured an average runtime overhead of 4% and a peak memory overhead of 3.5%.

Program	KSLOC
openssl-0.9.8k	397
Apache-2.2.11	474
nullhttpd-0.5.1	2
libpng-1.2.5	36
SPEC CINT2000	309
Olden	6
Total	1224

Table 3.1: Source lines of code in programs successfully built and run with BBC.

3.9 64-Bit Architectures

This section investigates ways to further optimise my solution taking advantage of 64-bit architectures. The key observation is that pointers in 64-bit architectures have spare bits to use. Current models of AMD64 processors use 48 out of 64 bits in pointers requiring the remaining bits to be sign extended on accesses, as shown in Figure 3.20 (a), with Windows further limiting this to 43 bits for user space programs, as shown in Figure 3.20 (b). Thus 21 bits in the pointer representation remain unused, and several of those which are used could be retargeted without much harm. Next, I describe two uses for these spare bits, and present a performance evaluation on AMD64.

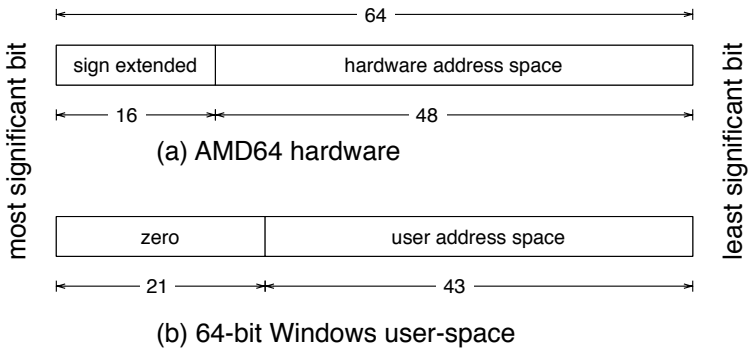


Figure 3.20: Use of pointer bits by AMD64 hardware and user-space Windows applications.

3.9.1 Size tagging

Since baggy bounds occupy less than a byte, they can fit in a 64 bit pointer's spare bits, removing the need for a separate data structure. These *tagged pointers* work similarly to fat pointers but have several advantages.

First, tagged pointers retain the size of regular pointers, avoiding fat pointers' register and memory waste. Moreover, their memory stores and loads are atomic, unlike fat pointers that can break code which relies on this. Finally, they preserve the memory layout of structures, overcoming the main drawback of fat pointers: lack of interoperability with uninstrumented code.

For interoperability, instrumented code must be able to use pointers from uninstrumented code and vice versa. I achieve the former by interpreting the default zero value found in unused bits of user-space pointers as maximal bounds, so checks on pointers missing bounds information will succeed. Supporting interoperability in the other direction is harder because the extra bits are not sign extended as expected by the hardware, raising a hardware exception when uninstrumented accesses memory through a tagged pointer.

I used the paging hardware to address this by mapping all addresses that differ only in their tag bits to the same physical memory. This way, unmodified binary libraries can dereference tagged pointers, and instrumented code avoids the cost of clearing the tag too.

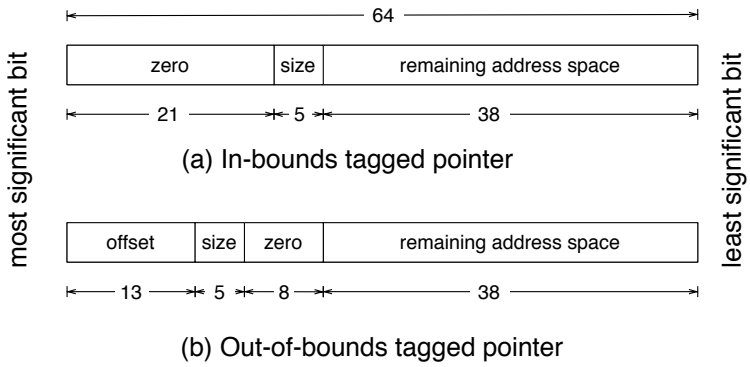


Figure 3.21: Use of pointer bits by in-bounds and out-of-bounds tagged pointers.

Using 5 bits to encode the size, as shown in Figure 3.21 (a), allows checking object sizes up to 2^{32} bytes. To use the paging mechanism, these 5 bits have to come from the 43 bits supported by the Windows operating system, thus leaving 38 bits of address space for programs.

To support 5 bits for bounds, 32 different virtual address regions must be mapped to the same physical memory. I implemented this entirely in user space using the `CreateFileMapping` and `MapViewOfFileEx` Windows API functions to replace the process image, stack, and heap with a file backed by the system paging file (Windows terminology for an anonymous mapping in operating systems using `mmap`) and mapped at 32 different virtual addresses in the process address space.

Now that 5 address bits are effectively ignored by the hardware, they can be used to store the size of memory allocations. For heap allocations, `malloc` wrappers set the tag bits in returned pointers. For locals and globals, the address taking operator "`&`" is instrumented to properly tag the resulting pointer. The bit complement of the size logarithm is stored to enable interoperability with untagged pointers by interpreting their zero bit pattern as all bits set (representing a maximal allocation of 2^{32}).

With the bounds embedded in pointers, there is no need for a memory lookup to check pointer arithmetic. Figure 3.22 shows the AMD64 code sequence for checking pointer arithmetic using a tagged pointer. First, the encoded bounds are extracted from the source pointer by right shifting a copy to bring the tag to the bottom 8 bits of the register and XORing them with the value `0x1f` to recover the size logarithm by inverting the bottom 5 bits. Then the result of the arithmetic is checked by XORing

pointer arithmetic

```
1  p = cgiCommand + i;
```

code inserted to extract bounds

```
2  mov rax, cgiCommand
3  shr rax, 26h
4  xor rax, 1fh
```

code inserted to check bounds

```
5  mov rbx, cgiCommand
6  xor rbx, p
7  shr rbx, al
8  jz ok
9    p = slowPath(buf, p)
10 ok:
```

Figure 3.22: AMD64 code sequence inserted to check unsafe arithmetic with tagged pointers. Note that the `al` register is an alias for the 8 least significant bits of the `rax` register.

the source and result pointers, shifting the result by the tag stored in `al`, and checking for zero.

Similarly to the table-based implementation, checks on out-of-bounds pointers trigger a bounds error to avoid an explicit check in the common case. To cause this, the bits holding the size are set to zero for out-of-bounds pointers and the size is stored using 5 more bits in the pointer, as shown in Figure 3.21 (b).

3.9.2 Increased out-of-bounds range

The spare bits can also store an offset for adjusting an out-of-bounds pointer to recover the address of its referent object. There are 13 bits available for storing this offset, as shown in Figure 3.21 (b). These bits can count slot or even allocation-size multiples, increasing the supported out-of-bounds range to at least 2^{16} bytes above or below an allocation. As in the 32-bit case, the hardware prevents dereferencing out-of-bounds pointers by detecting the presence of a non-zero out-of-bounds offset in the top bits.

Increasing the out-of-bounds range using this technique can be used independently of using bits in the pointer representation for storing bounds, *i.e.*, it can also be used with bounds stored in a table. When looking up a pointer in the table, however, the top bits of the pointer used for the out-of-bounds offset have to be masked off to get a valid table index (except for one bit used for making bounds checks fail for out-of-bounds pointers, as discussed in Section 3.7.5).

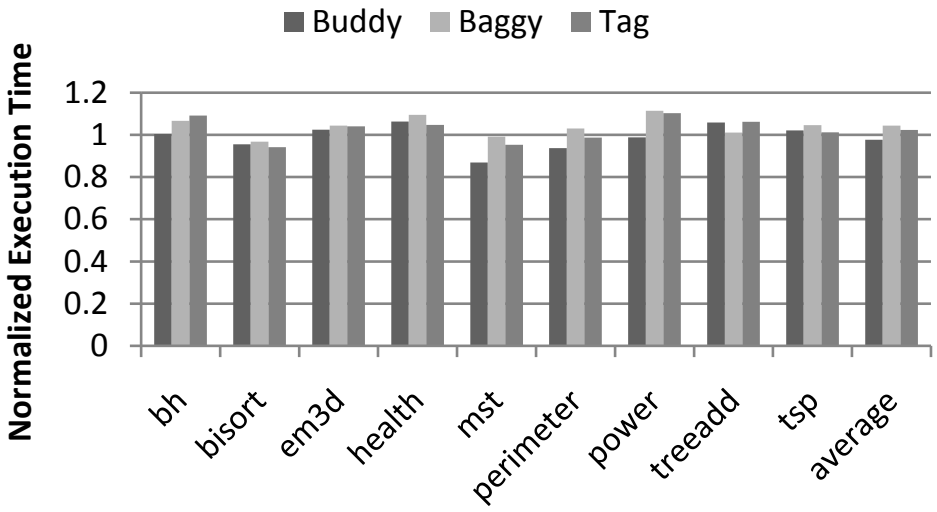


Figure 3.23: Normalised execution time on AMD64 with Olden benchmarks.

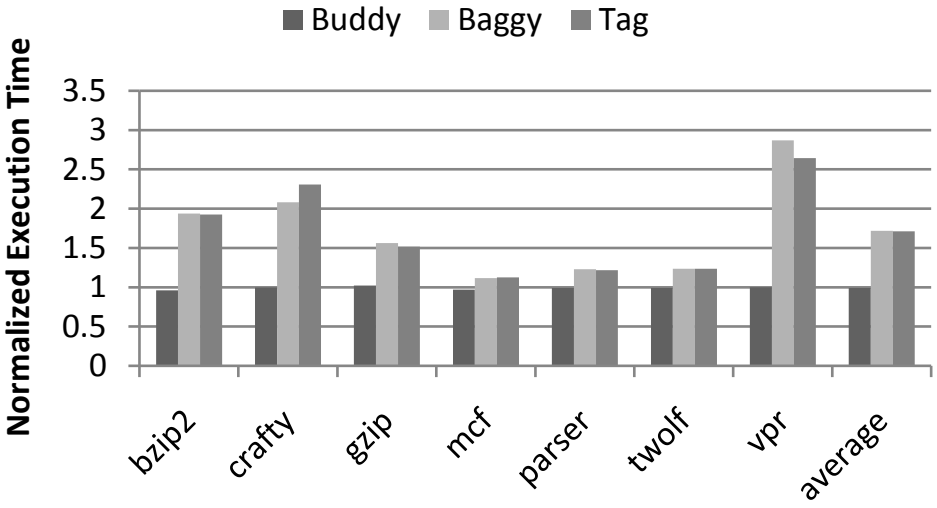


Figure 3.24: Normalised execution time on AMD64 with SPEC CINT2000 benchmarks.

3.9.3 Experimental evaluation

I evaluated baggy bounds checking on AMD64 using the subset of benchmarks from Section 3.8.1 that run unmodified on 64 bits. I measured the system using a contiguous array against the system using tagged pointers (Baggy and Tag in the figure legends respectively). I also measured the overhead using the buddy allocator only.

The multiple memory mappings complicated measuring memory use because Windows counts shared memory multiple times in peak memory reports. To overcome this, I measured memory use without actually tagging the pointers, to avoid touching more than one address for the same memory, but with the memory mappings in place to account for at least the top-level memory-management overheads.

Figures 3.23 and 3.24 show the time overhead. The average when using a table-based implementation on 64-bits is 4% for Olden and 72% for SPEC CINT2000—close to the 32-bit results of Section 3.8. Figures 3.25 and 3.26 show the space overhead. The average using a table is 21% for Olden and 11% for SPEC CINT2000. Olden's space overhead is higher than the 32-bit version; unlike the 32-bit case, the buddy allocator contributes to this overhead by 14% on average.

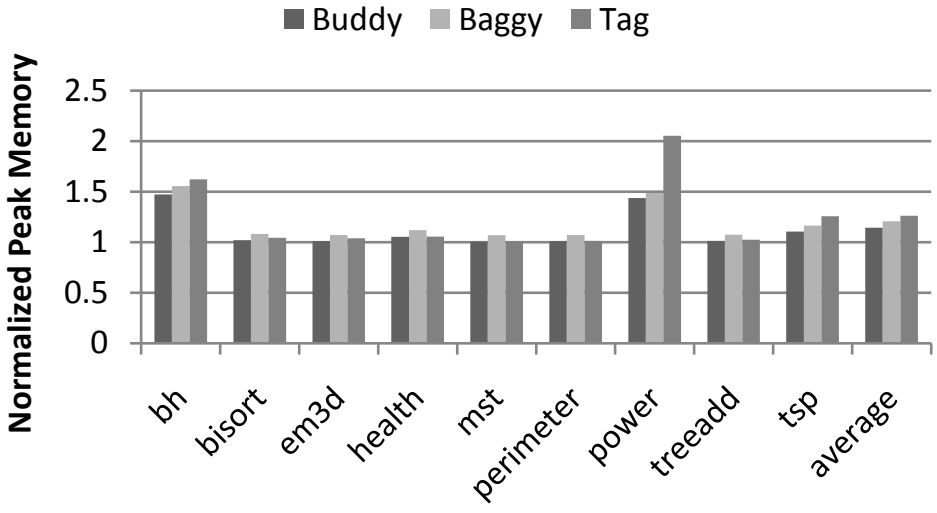


Figure 3.25: Normalised peak memory use on AMD64 with Olden benchmarks.

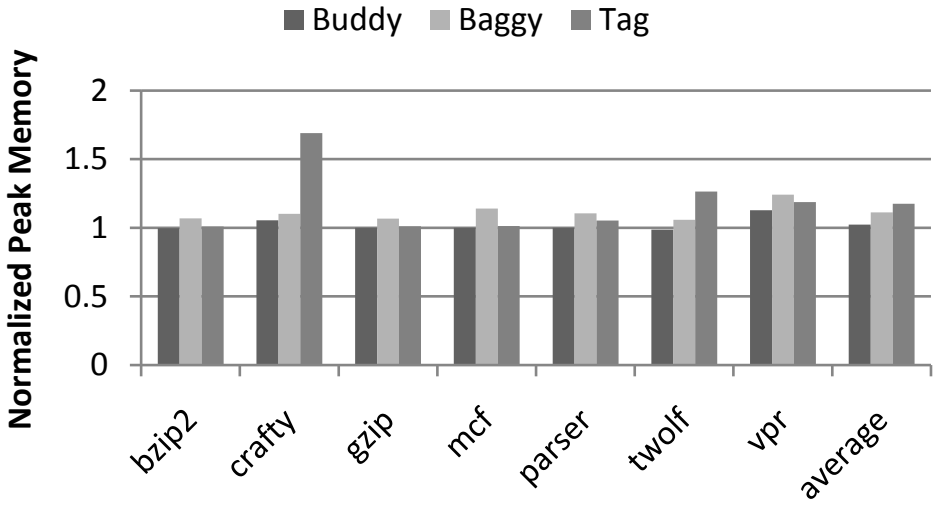


Figure 3.26: Normalised peak memory use on AMD64 with SPEC CINT2000 benchmarks.

Tagged pointers, are 1–2% faster on average than using a table, but slower for `bh` and especially `crafty`. Tagged pointers also use about 5% less memory for most benchmarks, as expected from avoiding the table's space overhead, except for a few such as `power` and `crafty`. These exceptions arise because the prototype does not map pages to different addresses on demand, but instead maps 32 30-bit regions of

virtual address space on program startup. Hence the fixed overhead is notable for these benchmarks whose absolute memory usage is low.

While mapping multiple views was implemented entirely in user-space, a robust implementation would probably require kernel support. The gains, however, appear too small to justify the complexity.

3.10 Discussion

3.10.1 Alternative implementations

Two alternative 64-bit implementations are briefly discussed here. If code is transformed by relocating unsafe local variables to the heap, which may have some runtime cost, both the table and the tagged-pointer variants can be improved in new ways. Instead of a buddy system, segregated allocation zones can be used. Placing allocations of the same size adjacent to each other enables some optimisations. In the table-based variant, slot sizes can increase by orders of magnitude, decreasing table memory waste and improving cache performance. In the case of tagged pointers, allocation zones can be placed in virtual addresses matching their size tags, removing the need for instrumentation to set the tag bits. This almost makes tagged pointers practical on 32-bit architectures, as address bits are reused for the bounds. In addition, safe variables and library data can be loaded to high addresses, with all size tag bits set (corresponding to maximal bounds), removing the need for interoperability instrumentation.

3.10.2 False positives and negatives

Second-guessing programmers' intent based on C source code may lead to false positives. While safe languages can unambiguously enforce array bounds, nested objects obscure the bounds that should be enforced in C programs. For example, when using a pointer to an array element, the array element may itself be an array (or treated as a byte array). Should accesses be limited to the inner array or allowed to traverse the entire array? Another example is a pointer to an array in a structure. Programmers can use the `offsetof` macro or similar constructs to legitimately access other structure fields.

For these reasons, spatial safety in C is usually enforced at the outer object level. Crossing memory allocation boundaries is undefined, and programs are unlikely to intentionally violate this, both for portability reasons and because optimising compilers may reorder variables. This rule, however, can only detect memory errors that cross object boundaries, introducing the possibility of false negatives, *e.g.* when overflowing an array in a structure overwrites a pointer in the same structure. The solution in Chapter 4 uses out-of-band information about pointers to protect against such attacks by constraining subsequent uses of pointers overwritten this way. BBC, however, cannot use out-of-band information to support sub-object protection, because the bounds stored in a table are relative to the address in the pointer.

Even this rule, however, may cause false positives: I have encountered two. Firstly, compiling `gcc` from SPEC CINT2000 benchmarks without defining `__STDC__` gener-

ates code that uses `va_arg` and a local variable instead of a named argument to access the unnamed arguments. This is the only case of intentional access across objects I have encountered. The other case involved an unused invalid read in the `h264ref` benchmark in the SPEC CINT2006 suite:

```
1 int dd, d[16];
2 for (dd=d[k=0]; k<16; dd=d[++k])
3     // ...
```

Here the bogus result of the buffer read past the end of the buffer is never used, rendering this violation harmless and hard to notice, even though it is clearly a bug according to ISO C. Such problems, however, are less likely to go unnoticed with writes, because they may corrupt memory. The solution in Chapter 4 can avoid such false positives by checking only writes.

3.10.3 Temporal safety

BBC addresses spatial safety with increased performance and backwards compatibility, but requires additional support to address temporal safety. Options include conservative garbage collection [24], reference counting [70], and region-based memory management [71]. These may be more efficient than checking every potentially unsafe memory access [11, 170], but introduce some backwards-compatibility problems.

Conservative garbage collection [24] can be used with C programs and has adequate performance for some user-space applications [177, 55], but its sensitivity to pointers hidden from the garbage collector (*e.g.* pointers referring to an object by pointing to a known offset outside the object's bounds and pointers stored as a value XORed with an integer) may require source code modifications to avoid false positives. However, if used merely for safety, *alongside* manual memory management instead of replacing it, by garbage collecting only memory that has been manually deallocated, source code modifications and false positives can be avoided at the cost of false negatives—which, if rare, are more acceptable in practice. False negatives are rare because they require a pointer that is both dangling and also hidden from the garbage collector.

Another approach is to take advantage of the vast virtual address space of 64-bit architectures (43 bits in Windows) to avoid reusing virtual memory. The physical memory for address ranges containing only freed objects can be cleared and returned to the operating system. Eventually, address space will be exhausted too, but this can be alleviated by scanning memory to conservatively reuse address space that is not referenced by used memory, similar to conservative garbage collection but without the need to consider cycles. Alternatively, the address space can be reused after exhaustion, limiting coverage but avoiding the sweep overhead. The problem of this approach is that fragmentation causes increased memory usage: a single small object can hold back an entire page of physical memory.

Type homogeneity, introduced by Dhurjati *et al.* [60, 59] (discussed in Section 2.4), offers a compromise between security and cost, by preventing pointers and plain data from reusing each other's memory. Dhurjati *et al.* [60] refrain from a simple implementation for temporal protection that could use a disjoint memory pool for each allocation site in the code, opting for a sophisticated region inference algorithm that also helps with spatial protection. In fact, imprecision in region inference may lead

to less temporal protection than the simple implementation. Fortunately, the simple implementation is compatible with BBC. The memory consumption of the simple scheme can be improved by reusing physical memory pages backing deallocated objects, and it is less sensitive to fragmentation, as it can reuse memory within pools. The resulting increase in address space use is not a problem in 64-bit systems, and can be addressed in 32-bit systems with infrequent conservative garbage collection, or simply ignored by allowing address space to wrap around and reuse pages in the hope that the delay was sufficient to thwart attacks.

The next chapter presents an alternative solution that is even faster at the expense of some spatial protection, and also adds a degree of temporal protection.

Chapter 4

Write integrity testing

This chapter presents WIT, a defence against memory errors that is fast, offers broad coverage against attacks, is backwards-compatible, and does not incur false positives.

WIT differs from BBC and other bounds checking systems in how it addresses the fundamental cost of tracking the intended referent objects of pointers to ensure dereferences stay within bounds. Some bounds checking systems address it by attaching metadata to pointers, while the Jones and Kelly approach (used by BBC) avoids pointer metadata by instrumenting pointer arithmetic to ensure pointers keep pointing to their intended referent. BBC is fast because of its simple, streamlined runtime pointer tracking. WIT goes even further by statically approximating tracking to streamline checks even more, and using static analysis to increase protection when possible.

At compile time, WIT uses interprocedural points-to analysis [76] to compute the control-flow graph and the set of objects that can be written by each instruction in the program. At runtime, WIT enforces *write integrity* (Section 1.4.2), that is, it prevents instructions from modifying objects that are not in the set computed by the static analysis. Additionally, WIT inserts small objects called *guards* between the original objects in the program. Since the guards are not in any of the sets computed by the static analysis, this allows WIT to prevent sequential overflows and underflows even when the static analysis is imprecise. WIT also enforces *control-flow integrity* [1, 2], that is, it ensures that the control-flow transfers at runtime are allowed by the control-flow graph computed by the static analysis.

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

WIT also assigns a colour to indirect call instructions and to the entry points of functions that have their address taken in the source code and thus may be called indirectly such that all functions that may be called by the same instruction have the same colour. WIT instruments the code to record function colours in the colour table and to check indirect calls. The indirect call checks look up the colour of the target address in the table and check if it matches the colour of the indirect call instruction. These checks together with the write checks ensure control-flow integrity.

Control-flow integrity prevents the attacker from bypassing the checks and provides an effective second line of defence against attacks that are not detected by the write checks.

These mechanisms allow WIT to provide broad coverage using lightweight runtime instrumentation. In the rest of this chapter I will explain WIT, and evaluate its coverage and performance.

4.1 Overview

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

Figure 4.1: Example vulnerable code: simplified web server with a buffer overflow vulnerability.

WIT has both a compile-time and a runtime component. I will use the familiar example from Chapter 3, repeated in Figure 4.1, to illustrate how both components work. Recall that the web server of the example receives a request for a CGI command of unknown size to be stored in the fixed size variable `cgiCommand`. By overflowing `cgiCommand`, an attacker can overwrite `cgiDir` and coerce the web server into executing a program from anywhere in the file system.

At compile time, the system uses a points-to analysis [76] to compute the set of objects that can be modified by each instruction in the program. For the example in Figure 4.1, the analysis computes the set `{i}` for the instructions at lines 5 and 8, and the set `{cgiCommand}` for the instruction at line 7.

To reduce space and time overhead at runtime, the system performs a *write safety* analysis that identifies instructions and objects that are *safe*. An instruction is safe if it cannot violate write integrity and an object is safe if all instructions that modify the object (according to the points-to analysis) are safe. In particular, all read instructions are safe, as well as write instructions to constant offsets from the data section or the frame pointer, assuming the frame pointer is not corrupted (which WIT can guarantee since saved frame-pointers are not within the write set of any program instruction). In the example, the write safety analysis determines that instructions 5 and 8 are safe because they can only modify `i` and, therefore, `i` is safe. It also determines that

the arguments to `ProcessCGIRequest` are safe. In contrast, instruction 7 is not safe because it may modify objects other than `cgiCommand` depending on `i`'s value.

The results of the points-to and write safety analysis are used to assign a *colour* (small integer) to each write instruction and to each object in the program. Preferably distinct colours are assigned to each unsafe object under the constraint that each instruction must have the same colour as the objects it can write. The colour 0 is assigned to all safe objects and all safe instructions to save bits used to represent colours. Colours 1 and 2 are reserved; their use will be explained in the following sections. In the example, variables `msg`, `sz`, `i` and `c` and instructions 5 and 7 are assigned colour 0 because they are safe. The variable `cgiCommand` and instruction 7 are assigned colour 3, and variable `cgiDir` colour 4.

To reduce the false negative rate due to the imprecision of the points-to analysis and merging of intersecting points-to sets of write instructions, small guards are inserted between the unsafe objects in the original program. Guard objects have colour 0 or 1 (1 is reserved for guards on the heap, as explained later). These colours are never assigned to unsafe instructions in the program to ensure that WIT can detect attempts to overwrite guards or safe objects.

The points-to analysis is also used to compute the functions that can be called by each indirect call instruction in the program. Indirect call instructions and the functions they call are also assigned colours. WIT attempts to assign distinct colours to each function while ensuring that each instruction and the functions it can call have the same colour. The set of colours assigned to functions is disjoint from the set of colours assigned to unsafe objects, to safe objects, and to guards. This prevents unsafe instructions from overwriting code and prevents control transfers outside code regions.

WIT adds extra compilation phases that insert instrumentation to enforce write integrity and control-flow integrity. There are four types of instrumentation: to insert guards, to maintain the colour table, to check writes, and to check indirect calls. Guards are eight bytes long in my implementation. The code in the example is instrumented to add guards just before `cgiCommand`, between `cgiCommand` and `cgiDir`, and just after `cgiDir`. No guards are inserted around the arguments to `ProcessCGIRequest` and the local variable `i` because they are safe.

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

The checks on writes compare the colour of the instruction performing the write to the colour of the storage location being written. If the colours are different, they raise a security exception. The colour of each instruction is known statically and write checks use the colour table to look up the colour of the location being written. No checks have to be inserted for safe instructions. In the example in Figure 4.1, WIT adds write checks only before instruction 7 to check if the location being written has colour 3. It does not add write checks before lines 5 and 8 because these instructions are safe.

WIT also records in the colour table the colour of each function that can be called indirectly. It inserts instrumentation to update the colour table at program start-up time and to check the colour table on indirect calls. The indirect call checks compare the colour of the indirect call instruction and its target. If the colours are different, they raise an exception. There are no indirect calls in the example of Figure 4.1.

WIT can prevent all attacks that violate write integrity but, unlike baggy bounds used in Chapter 3, which attacks violate this property depends on the precision of the points-to analysis. For example, if two objects have the same colour, it may fail to detect attacks that use a pointer to one object to write to the other. The results show that the analysis is sufficiently precise for most programs to make this hard. Additionally, WIT can prevent many attacks regardless of the precision of the points-to analysis. For example, it prevents: attacks that exploit buffer overflows and underflows that only allow sequential writes to increasing or decreasing addresses until an object boundary is crossed (which are the most common type of memory-safety vulnerability); attacks that overwrite any safe objects (which include return addresses, exception handler pointers, and data structures for dynamic linking); and attacks that corrupt heap-management data structures.

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

WIT does not prevent out-of-bounds reads, and is, hence, vulnerable to disclosure of confidential data. It protects, however, indirectly against out-of-bounds reads of pointers. Storing pointer metadata out-of-band makes it hard to exploit out-of-bounds pointer reads without violating write integrity or control-flow integrity in the process. (Refer to Section 2.3 for a discussion of the threat posed by such out-of-bounds reads vs. out-of-bounds reads in general). Therefore, I chose not to instrument reads to achieve lower overhead, and to increase the precision of the analysis by avoiding merging sets due to constraints caused by read instructions which are otherwise not needed for writes.

WIT can prevent attacks on the example web server. The write check before line 8 fails and raises an exception if the attacker attempts to overflow `cgiCommand`. When `i` is 1024, the colour of the location being written is 0 (which is the colour of the guard) rather than 3 (which is the colour of `cgiCommand`). Even without guards, WIT would be able to detect this attack because the colours of `cgiCommand` and `cgiDir` are different.

4.2 Static analysis

I used points-to and safety analysis implementations based on [32] exploiting the Phoenix compiler framework [99]. These analyses require a pass over the whole program, which can take place during link time using the link-time code generation (LTCG) mode in modern compilers.

The interprocedural points-to analysis is due to Andersen [7], and is flow and context insensitive. While the tightest proven worst-case complexity for it is nearly cubic, it scales better on real-world programs [140]. It computes a points-to set for each pointer, which is the set of logical objects the pointer may refer to. The logical objects are local and global variables and dynamically allocated objects (for example, allocated with `malloc`). As in Andersen, a single logical object is used to represent all objects that are dynamically allocated at the same point in the program, but simple allocation wrappers could be detected and cloned at compile time to improve the analysis precision.

The analysis assumes, as the C standard prescribes, that the relative layout of independent objects in memory is undefined [12]. For example in Figure 4.1, it assumes that correct programs will not use the address of `cgiCommand[i]`, which points into the `cgiCommand` array, to write to `cgiDir`. Compilers already make this assumption when implementing standard optimisations. Programs can get away with violations involving unused reads. (Recall the out-of-bounds read in `h264ref` discussed in Section 3.10.2.) However, programs violating this assumption through writes can corrupt memory. WIT avoids false positives by only checking writes.

The analysis is conservative: a points-to set includes all objects that the pointer may refer to in executions that do not violate memory safety (but it may include additional objects). The conservative points-to information can be thought of as a static approximation of runtime referent object tracking. It has no runtime overhead, but some precision is lost.

The write safety analysis classifies instructions as safe or unsafe: an instruction is marked safe if it cannot violate write integrity. The analysis marks safe all instructions that only read data, or any which write a temporary, a local variable, or a global. These instructions are safe because they either only modify registers or they modify a constant number of bytes starting at a constant offset from the frame pointer or the data section. These constant offsets are generated by the compiler and linker, and can be assumed correct. The frame pointer is safe even with recursion and calls to `alloca` because the runtime ensures there is always a guard page at the end of the stack to prevent stack overflows. (The compiler ensures that the prologues of functions with frames larger than a page or functions using `alloca` check if the pages they need are resident, faulting on the guard page to trigger stack growth when necessary.) In addition, the simple intraprocedural pointer-range analysis from Chapter 3 is run while making the global pass over all source files to collect constraints for the points-to analysis. It marks as safe all instructions whose memory accesses are always in bounds. The write safety analysis does not introduce false negatives because control-flow integrity prevents the attacker from bypassing the checks or changing the data segment or the frame pointer.

The results of the points-to and write safety analysis are used to assign colours to objects and to unsafe instructions. An iterative process is used to compute *colour sets*, which include objects and unsafe pointer dereferences that must be assigned the same colour because they may alias each other. Initially, there is a separate colour set for each points-to set of an unsafe pointer: the initial colour set for a points-to set $p \rightarrow \{o_1, \ldots, o_n\}$ is $\{[p], o_1, \ldots, o_n\}$. Then intersecting colour sets are merged until a fixed point is reached. A distinct colour is assigned to each colour set: it is assigned to all objects in the colour set and all instructions that write pointer dereferences in the set. All the other objects in the original program are assigned colour zero. By only considering points-to sets of unsafe pointers when computing colours, the false negative rate and the overhead to maintain the colour table are reduced.

WIT uses a similar algorithm to assign colours to functions that have their address taken in the source code and thus may be called indirectly. The differences are that this version of the algorithm iterates over the points-to sets of pointers that are used in indirect call instructions (except indirect calls to functions in dynamically linked libraries), and that it only considers the objects in these sets that are functions. Compiler generated indirect calls through the Windows import address table (IAT) to library functions can be excluded because the IAT is protected from corruption by the write checks. Each colour set is assigned a different colour that is different from 0, 1, and the colours assigned to unsafe objects. The rest of the code is assigned colour zero.

4.3 Instrumentation

4.3.1 Colour table

WIT tracks colours in a table similar to the one used by BBC to track bounds, using 1 byte to represent the colour of a memory slot. The slot size, however, is 8 bytes, to match the alignment used by the 32-bit Windows standard memory allocator. Therefore, the table introduces a space overhead of 12.5%. Recall that to prevent any two objects having to share a slot, all unsafe objects must be aligned to multiples of 8 bytes. This does not introduce any memory overhead for dynamic allocations, since the memory allocator on Windows already aligns memory sufficiently. However, since the stack on 32-bit Windows is only four-byte aligned, enforcing 8 byte alignment for local variables by aligning the frame pointer in the function prologue would disrupt the *frame pointer omission* optimisation applied by compilers for using the frame pointer as an extra general purpose register. This optimisation is important for register-starved architectures such as the 32-bit x86.

WIT can satisfy the alignment constraints for unsafe local variables without aligning the stack frame in the function prologue by using the following technique. It forces unsafe objects and guard objects in the stack and data sections to be four byte aligned (which the compiler can satisfy with padding alone) and inserts additional four-byte aligned pads after unsafe objects. For an unsafe object of size s, the pad is eight-bytes long if $\lceil s/4 \rceil$ is even and four-bytes long if $\lceil s/4 \rceil$ is odd. WIT sets $\lceil s/8 \rceil$ colour-table entries to the colour of the unsafe object when the pad is four bytes long and $\lceil s/8 \rceil + 1$

when the pad is eight bytes long. This workaround is not needed in 64-bit Windows or operating systems having an 8- or 16-byte-aligned stack.

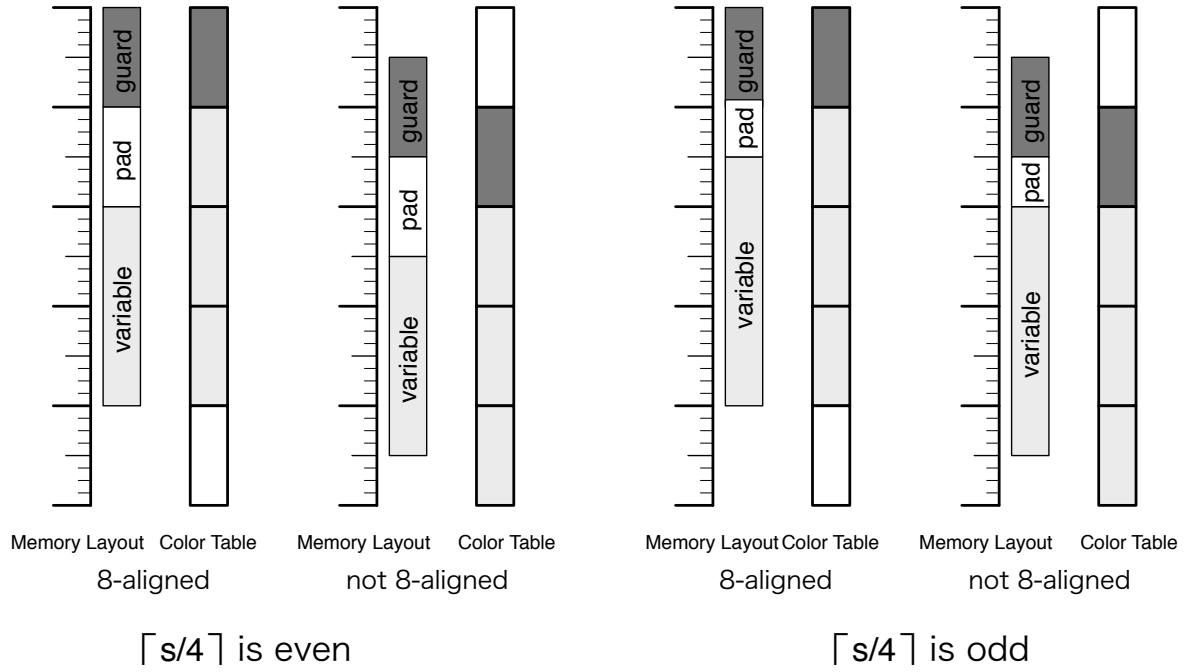


Figure 4.2: Ensuring that two objects with distinct colours never share the same eight-byte slot. The pad after unsafe objects takes the colour of the guard, the unsafe object, or both depending on the actual alignment at runtime. The lowest addresses are at the bottom of the figure.

Figure 4.2 shows how the extra padding works. Depending on the alignment at runtime, the pad gets the colour of the unsafe object, the guard, or both. All these configurations are legal because the pads and guards should not be accessed by correct programs and the storage locations occupied by unsafe objects are always coloured correctly. Conceptually, the pads allow the guards to "move" to ensure that they do not share a slot with the unsafe objects.

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

Eight bits are sufficient to represent enough colours because the write safety analysis is very effective at reducing the number of objects that must be coloured. However, it is possible that more bits will be required to represent colours in very large programs. If this ever happens, two solutions are possible: colour-table entries can be increased to 16 bits, or stay at 8 bits at the expense of more false negatives.

As with BBC, the colour table is implemented as an efficient array. For a Windows system with 2 GB of virtual address space available to the program, 256 MB of virtual address space is reserved for the colour table. A vectored exception handler is used to capture accesses to unallocated memory, allocate virtual memory, and initialise it to zero. The base of the colour table is currently at address 40000000h. So to compute

the address of the colour-table entry for a storage location, its address is shifted right by three, and added to 40000000h.

4.3.2 Guard objects

Colour assignments offer partial protection. For one, a single colour is used for all objects allocated at the same point in the program. Further precision is lost due to the conservative nature of interprocedural analysis. Finally, colour sets are merged when points-to sets of write instructions are overlapping (since the objects in the intersection must have a single colour). To guarantee a certain level of protection irrespective of the analysis' precision, small guards are inserted before and after unsafe objects. Thanks to these guards, WIT is guaranteed to detect overflows and underflows that write array elements sequentially without skipping memory until crossing an object boundary, which are the most common.

Each guard occupies an eight-byte slot and a single entry in the colour table. The instrumentation to insert these guards is different for the stack, heap, and global data sections.

To insert guards in the stack, I reimplemented the compiler phase that lays out local variables in a stack frame. My implementation segregates safe local variables from unsafe ones to reduce space overhead. First, it allocates contiguous storage for the safe local variables. Then it allocates storage for the guards, pads, and unsafe local variables. This enables inserting only $n + 1$ guards and pads for n unsafe local variables: the guard that prevents overflows of a variable prevents underflows of the next variable. As it will be explained in Section 4.3.3, the colours are set by code inserted into function prologues, thus the standard optimisation of overlapping in memory any frame variables with non-overlapping lifetimes must be disabled to prevent function prologues from setting conflicting colours for different variables. This optimisation could be supported by moving colour assignments from the prologue to appropriate locations in the function code.

As with BBC, the rare case where a function argument is written by an unsafe instruction is handled by inserting code in the function prologue to copy the unsafe argument into a local variable created for this purpose and rewriting the instructions to refer to the copy. This local variable is marked unsafe and gets guards and pads around it.

All heap-allocated objects are marked as unsafe, but my implementation avoids the space overhead of inserting pads or guards around them. The standard heap allocator in Windows 7, Windows Vista, Windows XP SP2, and Windows 2003 inserts an eight-byte header before each allocated object that can be used as a guard by simply setting its colour to 1 in the colour table. Since heap objects and headers are eight-byte aligned, no pads are required either. The space overhead saved by this optimisation could be significant for programs with many small allocations.

Guards and pads are added around all variables in the `.data` and `.bss` sections but not in the read-only data section (`.rdata`). Some space could be saved by segregating unsafe variables as in the stack, but my experiments show that this would have little impact on overall performance as the data section usually contains a small fraction of the program's objects.

Finally, an optimisation is possible to avoid the need for most guards between stack and global objects, by laying them out such that adjacent objects have different colours. This is not implemented in the current prototype.

4.3.3 Maintaining the colour table

Colour table pages are initialised to zero when they are first accessed. This ensures security by default: unsafe writes and indirect calls to an address are not allowed unless the corresponding colour-table entry has been set explicitly.

The colour-table entries for global variables are initialised at program startup. The guards of global variables have colour zero. The first instruction in functions is aligned on a 16-byte boundary, and colour-table entries corresponding to the first instructions of allowed indirect-call targets are also initialised at program startup. But the colour-table entries for objects on the stack and the heap are updated dynamically when the objects are allocated. The colour zero is used for guards of unsafe local variables.

An optimisation is used to reduce the cost of updating the colour table when a new stack frame is allocated. Instead of updating the colour-table entries for all objects in the stack frame, only the entries corresponding to unsafe local variables and their guards are updated on function entry. On function exit, the entries updated on function entry are reset to zero. This works because all safe objects have colour zero, which is the initial colour of all colour-table entries. When detecting the use of `alloca` at compile time (which is always possible because `alloca` cannot be used through a function pointer), WIT inserts code to set object and guard colours following the `alloca` invocation and code to clear the colours on function exit. This optimisation may result in false negatives with `setjmp/longjmp` due to uncleared colours, but will not cause false positives, and it was made to work with exceptions (Microsoft SEH).

The prologue and epilogue of functions with unsafe variables are instrumented to set and reset colour-table entries. The following code sequence is added to the prologue of a function with a single unsafe local variable that is 12 bytes in size and located at bytes `esp+0x44` to `esp+0x4F`, with 4 bytes of padding and guards before and after.

```
1 push ecx
2 lea ecx, [esp+0x3C]
3 shr ecx, 3
4 mov dword ptr [ecx+0x40000000], 0x00020200
5 pop ecx
```

This sequence saves `ecx` on the stack to use it as a temporary. (This can be avoided if a register is available.) Then it loads the address of the first guard (`esp+0x3C`) into `ecx` and shifts it by three to obtain the index of the guard's colour-table entry. It uses this index to set the colour-table entries to the appropriate colours. For an unsafe object of size s, $\lceil s/8 \rceil$ colour-table entries are set when $\lceil s/4 \rceil$ is odd and $\lceil s/8 \rceil + 1$ when $\lceil s/4 \rceil$ is even (see Section 4.3.1). To reduce the space and time overhead of the instrumentation, 2-byte and 4-byte moves are used whenever possible. In this example, the `mov` on line 4 updates the four colour-table entries using a single write instruction: the entries

corresponding to guard objects are set to 0 and those corresponding to the unsafe local variable are set to two. The base of the colour table is at address 40000000h. The final instruction restores the original value of `ecx`. The instrumentation for epilogues is identical but it sets the colour-table entries to zero.

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

The colour table is also updated when heap objects are allocated or freed. The program is instrumented to call wrappers of the allocation functions such as `malloc` and `calloc`. These wrappers receive the colour of the object being allocated as an additional argument. They call the corresponding allocator and then set the colour table entries for the allocated object to the argument colour. They set $\lceil s/8 \rceil$ entries for an object of size s using `memset`. They also set the colour-table entries for the eight-byte slots immediately before and after the object to colour 1. These two slots contain a chunk header maintained by the standard allocator in Windows that can double as a guard since it is never touched by application code. Calls to `free` are also replaced by calls to a wrapper. This wrapper sets the colour-table entries of the object being freed to zero and then invokes free. A different colour is used for guards in the heap to detect some invalid uses of free (as explained in the next section).

4.3.4 Inserted checks

Only writes performed by unsafe instructions have to be checked. The check has to look up the colour of the destination operand in the colour table and compare it to the colour of the instruction. If the colours are the same, the write can proceed. Otherwise, an exception is raised. For example, the following x86 assembly code is generated to check an unsafe write of `ebx` to the address in register `ecx` with the colour table starting at address 40000000h:

```
1      lea edx, [ecx]
2      shr edx, 3
3      cmp byte ptr [edx+0x40000000], 42
4      je L1
5      int 3
6  L1: mov byte ptr [ecx], ebx ; unsafe write
```

This code sequence loads the address of the destination operand into a register, and shifts the register right by three to obtain the operand's index in the colour table. Then it compares the colour in the table with 42, which is the colour of the unsafe instruction. If they are different, it executes `int 3`. This raises a software interrupt that invokes the debugger in debugging runs, or terminates execution in production runs. Other interrupts could be raised, but this one is convenient for debugging.

Additional memory management errors can be detected by treating `free` as an unsafe instruction that writes to the object pointed to by its argument. The wrapper for `free` receives the colour computed by the static analysis for the object being freed.

Then it checks if the pointer argument points to an object with this colour, if it is eight byte aligned, if it points in user space, and if the slot before this object has colour one. An exception is raised if this check fails. The first check prevents double frees because the colour of heap objects is reset to zero when they are freed. The last two checks prevent frees whose argument is a pointer to a non-heap object or a pointer into the middle of an allocated object. Recall that colour one is reserved for heap guards and is never assigned to other memory locations.

Checks are also inserted before indirect calls. These checks look up the colour of the target function in the colour table and compare this colour with the colour of the indirect call instruction. An exception is raised if they do not match. For example, the following x86 assembly code is generated to check an indirect call to the function whose address is in `edx` and whose colour is supposed to be 20.

```
1      shr edx, 3
2      cmp byte ptr [edx+0x40000000], 20
3      je L1
4      int 3
5  L1: shl edx, 3
6      call edx ; indirect call
```

The first instruction shifts the function pointer right by three to compute the colour table index of the first instruction in the target function. The `cmp` instruction checks if the colour in the table is 20, the colour for allowed targets for this indirect call instruction. If they are different, WIT raises an exception. If they are equal, the index is shifted left by three to restore the original function pointer value and the function is called.

Unlike the instruction sequence used for write checks, this sequence zeroes the three least significant bits of the function pointer value. WIT aligns the first instruction in a function on a 16-byte boundary (already the compiler default unless optimising for space), so this has no effect if the function pointer value is correct. But it prevents attacks that cause a control flow transfer into the middle of the first eight byte slot of an allowed target function. Therefore, this instruction sequence ensures that the indirect call transfers control to the first instruction of a call target that is allowed by the static analysis. The checks on indirect calls are sufficient to enforce control-flow integrity because all other control data is protected by the write checks.

4.3.5 Runtime library

WIT has a small runtime that, similar to BBC's, includes an initialisation function, a vectored exception handler to allocate memory reserved for the table on demand, and some wrappers for C runtime functions, including `malloc` and `free`. Unlike BBC, however, WIT does not replace the default memory allocator. The initialisation function prepares the colour table using `VirtualAlloc`, which reserves address space for the table, and installs an exception handler using `AddVectoredExceptionHandler` to allocate table pages when they are first accessed. The operating system zeros newly allocated table pages. The initialisation function sets the colour-table entries for globals and their guard objects, and for the entry points of indirect call targets. The C runtime (`libc`) is instrumented to invoke the WIT initialisation function.

The `libc` library is also instrumented to detect memory errors due to incorrect use of `libc` functions without having to provide wrappers for every one. However, instrumenting `libc` using the WIT version described so far would require a different `libc` binary for each program. Instead, a special WIT variant is used for libraries. It assigns the same well-known colour (different from zero or one) to all unsafe objects allocated by the library and inserts guards around these objects. All safe objects used by the library functions have colour zero. Before writes, the WIT variant for `libc` checks that the colour of the location being written is greater than one, that is, that the location is not a safe object or a guard object. These checks prevent `libc` functions from violating control-flow integrity. They also prevent all common buffer overflows due to incorrect use of `libc` functions. However, they cannot prevent attacks that overwrite an unsafe object by exploiting format-string vulnerabilities with the `%n` specifier. However, these can be prevented with static analysis [136, 12, 126, 35] and are in any case disallowed by some implementations [47].

Wrappers need to be written for functions that are not instrumented, including `libc` functions written in assembly (for example, `memcpy` and `strcpy`) and for system calls (for example, `recv`). These wrappers receive the colours of destination buffers as extra arguments and scan the colour-table entries corresponding to the slots written by the wrapped function to ensure that they have the right colour. Since the colour table is very compact, these wrappers introduce little overhead.

4.4 Protection

WIT prevents attacks that exploit buffer overflows and underflows by writing elements sequentially until an object boundary is crossed. These attacks are always prevented because the write checks fail when a guard is about to be overwritten. This type of attack is very common.

The write checks do not detect buffer overflows and underflows inside an object. For example, they will not detect an overflow of an array inside a C structure that overwrites a function pointer, a data pointer, or some security-critical data in the same function. In the first two cases, WIT can prevent the attacker from successfully exploiting this type of overflow because the indirect call checks severely restrict the targets of indirect calls and because the write checks may prevent writes using the corrupt data pointer. Most backwards-compatible C bounds checkers [82, 170, 130, 57] do not detect overflows inside objects and, unlike WIT, have no additional checks to prevent successful exploits. Moreover, enforcing the bounds of sub-objects is inherently ambiguous in C. Nevertheless, a recent proposal by Nagarakatte *et al.* [104] does offer support for detecting overflows inside objects, although this comes at a cost of up to 3× memory increase to store bounds for individual pointers.

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

assigned to indirect call targets are different from the colours assigned to unsafe objects and the rest of the code has colour zero.

WIT can prevent corruption of the heap-management data structures used by the standard allocator in Windows without any changes to the allocator code. The checks on `free` prevent corruption due to incorrect use of `free`, and the write checks prevent corruption by unsafe aligned writes because the data structures have colour one or zero. However, writes that are not aligned may overwrite the first few bytes of the heap metadata after an object. Misaligned writes generate exceptions in many architectures but they are allowed in the x86. To a large extent, misaligned writes can be disallowed cheaply, by checking the alignment at pointer casts. Adding eight bytes of padding at the end of each heap object can prevent corruption in all cases. In most applications, this adds little space and time overhead but it can add significant overhead in applications with many small allocations. This overhead may not be justifiable because most programs avoid misaligned writes for portability and performance, and recent versions of the Windows allocator can detect many cases of heap metadata corruption with reasonable overhead through sanity checks in memory allocation routines.

Control-flow integrity provides an effective second line of defence when the write checks fail to detect an attack, but which attacks violate control-flow integrity also depends on the precision of the points-to analysis. In the experiments of Section 4.6.5, the maximum number of indirect call targets with the same colour is 212 for `gap`, 38 for `vortex`, and below 7 for all the other applications.

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

WIT does not prevent out-of-bound reads. These can lead to disclosure of confidential data but are hard to exploit for executing arbitrary code without violating write integrity or control-flow integrity in the process, because writes and indirect calls through pointers retrieved by accessing out-of-bounds memory are still checked.

4.5 Interoperability

Interoperability requires running programs containing both instrumented and uninstrumented code without raising false alerts. Uninstrumented code is not disrupted, because WIT does not change the programmer visible layout of data structures in memory. All memory, however, has colour 0 by default, thus instrumented code would raise an alert when writing to memory allocated by uninstrumented code that has the default colour. Fortunately, heap allocations by uninstrumented code can be intercepted to assign a special colour at runtime and guards between allocations.

The data and code sections of uninstrumented modules can also be assigned special colours at startup. Guards, however, cannot be placed between global variables. With these in place, an error handler can check for the special colour before raising an alert.

However, the case of instrumented code accessing memory in stack frames of uninstrumented functions remains. This can happen if an uninstrumented function calls an instrumented function passing it a reference to a local variable. It can be avoided to a large extent by providing two versions for each function callable from uninstrumented code: one callable from instrumented code, and another from uninstrumented. All direct function calls in instrumented code can be modified to call the instrumented version using name mangling, while uninstrumented code defaults to the uninstrumented version with the unmangled name. Uninstrumented code, however, can still end up calling an instrumented function through a function pointer, and pass it an uncoloured local variable argument. To avoid false positives, this can be detected by the error handler by unwinding the stack of instrumented functions to decide whether the faulting address belongs to the stack frame of an instrumented function before raising an alert, but this has not been implemented in the current prototype.

4.6 Experimental evaluation

This section presents the results of experiments to evaluate the overhead of my implementation, the precision and impact of the points-to analysis on security, and WIT's effectiveness at preventing a broad range of real and synthetic attacks. WIT detects all the attacks in my tests and its CPU and memory overhead are consistently low.

4.6.1 Overhead for CPU-bound benchmarks

The first experiment measures the overhead added by WIT to 9 programs from the SPEC CINT2000 benchmark suite [156] (`gzip`, `vpr`, `mcf`, `crafty`, `parser`, `gap`, `vortex`, `bzip2` and `twolf`), and to 9 programs from the Olden [31] benchmark suite (`bh`, `bisort`, `em3d`, `health`, `mst`, `perimeter`, `power`, `treeadd`, and `tsp`). I did not run `gcc`, `eon`, and `perlbmk` from CINT2000 because of limitations of the points-to analysis implementation used, and do not include `voronoi` from Olden in the results because it is excluded from other studies. These programs were chosen to facilitate comparison with other techniques that have been evaluated using the same benchmark suites.

I compared the running time and peak physical memory usage of the programs compiled using Phoenix [99] with and without WIT's instrumentation. The programs were compiled with options `-O2` (maximise speed), `-fp:fast` (fast floating point model), and `-GS-` (no stack guards). WIT binaries are linked with the WIT runtime and a WIT-instrumented version of `libc` (see Section 4.3.5). The experiments ran on Windows Vista Enterprise, on an otherwise idle Dell OptiPlex 745 Workstation with a 2.46GHz Intel Core 2 processor and 3GB of memory. For each experiment, I present the average of 3 runs; the variance was negligible.

Figures 4.3 and 4.4 show the CPU overhead for the SPEC and Olden benchmarks with WIT. For SPEC, the average overhead is 10% and the maximum is 25%. For

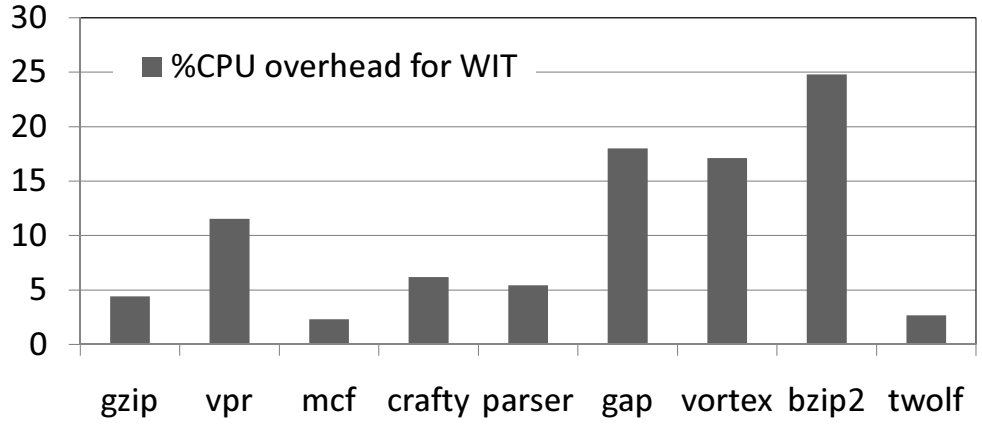


Figure 4.3: CPU overhead for SPEC benchmarks.

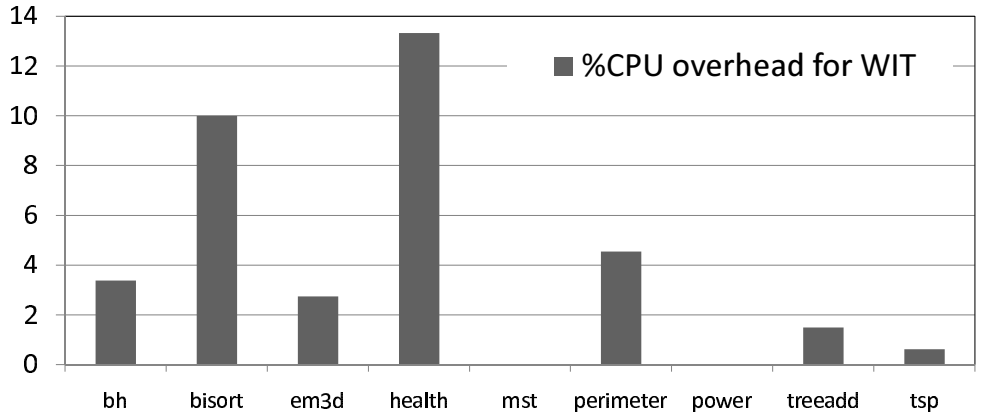


Figure 4.4: CPU overhead for Olden benchmarks.

Olden, the average overhead is 4% and the maximum is 13%. It is hard to do definitive comparisons with previous techniques because they use different compilers, operating systems and hardware, and they prevent different types of attacks. However, WIT's overhead can be compared with published overheads of other techniques on SPEC and Olden benchmarks. For example, CCured [105] reports a maximum overhead of 87% and an average of 28% for Olden benchmarks, but it slows down some applications by more than a factor of 9. The bounds-checking technique in [57] has an average overhead of 12% and a maximum overhead of 69% in the Olden benchmarks. WIT has three times lower overhead on average and the maximum is five times lower. Further comparisons are provided in Chapter 6.

Figures 4.5 and 4.6 show WIT's memory overhead for SPEC and Olden benchmarks. The overhead is low for all benchmarks. For SPEC, the average memory overhead is 13% and the maximum is 17%. For Olden, the average is 13% and the maximum is 16%. This overhead is in line with expectations: since WIT uses one byte in the colour table for each 8 bytes of application data, the memory overhead is close to 12.5%. The overhead can decrease below 12.5% because colour-table entries are not used for safe objects and most of the code. On the other hand, the overhead can grow above 12.5% because of guard objects and pads between unsafe objects, but the results show that this overhead is small. It is interesting to compare this overhead with that of previous

techniques even though they have different coverage. For example, CCured [105] reports an average memory overhead of 85% for Olden and a maximum of 161%. Xu *et al.* [170] report an average increase of memory usage by a factor of 4.31 for the Olden benchmarks and 1.59 for the SPEC benchmarks.

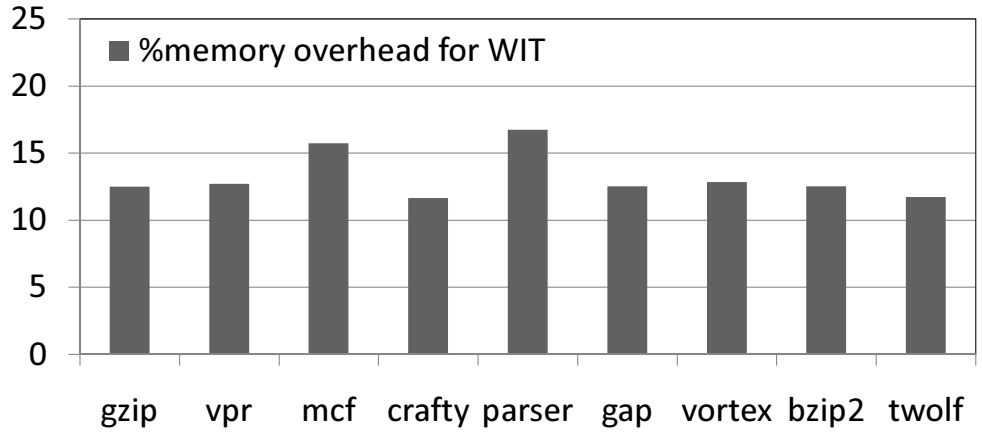


Figure 4.5: Memory overhead for SPEC benchmarks.

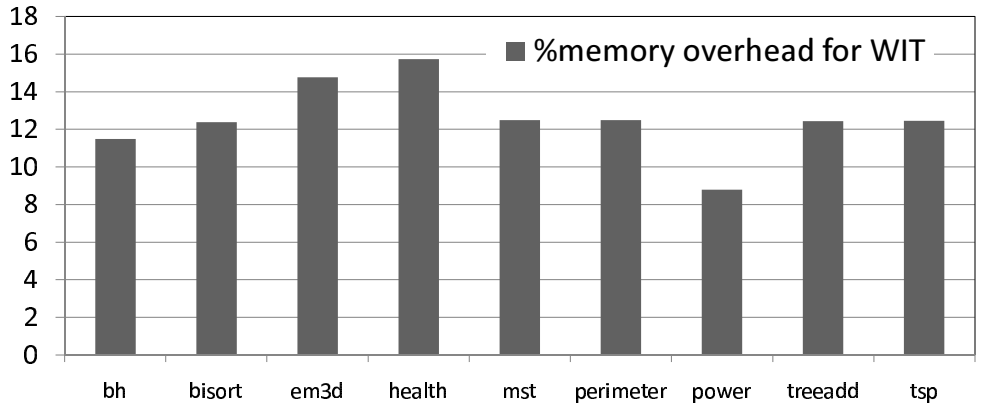


Figure 4.6: Memory overhead for Olden benchmarks.

Next, the SPEC benchmarks were used to break down the runtime overhead introduced by WIT. Parts of the instrumentation were removed in sequence. First, indirect-call checks were removed. Then instructions that perform write checks were removed, except for the first `lea` instruction (this maintains the register pressure due to the checks). Next the `lea` instructions were removed, which removes the register pressure added by write checks by freeing the registers they used. After that the instructions that set colours for heap objects were removed, and, finally, the setting of colours for stack objects.

Figure 4.7 shows a breakdown of the CPU overhead. For all benchmarks, the write checks account for more than half of the CPU overhead and a significant fraction of their overhead is due to register pressure. For `vortex`, `crafty` and `bzip2`, setting colours on stack allocations contributes significantly to the overhead because they have array local variables. For example, `bzip2` has large arrays which are not used in their entirety; nevertheless, colours need to be assigned each time. For `gap` and

vpr setting colours on heap allocations contributes 12% and 3% of the overhead but for all other benchmarks this overhead is negligible. The indirect-call checks have a negligible impact on performance except for gap where they contribute 16% of the overhead, because one of the main loops in gap looks up function pointers from a data structure and calls the corresponding functions. The gap overhead may thus be representative for C++ programs not otherwise evaluated in this work. These results suggest that improving write-safety analysis and eliminating redundant checks for overlapping memory slots in loops and structures could reduce the overhead even further.

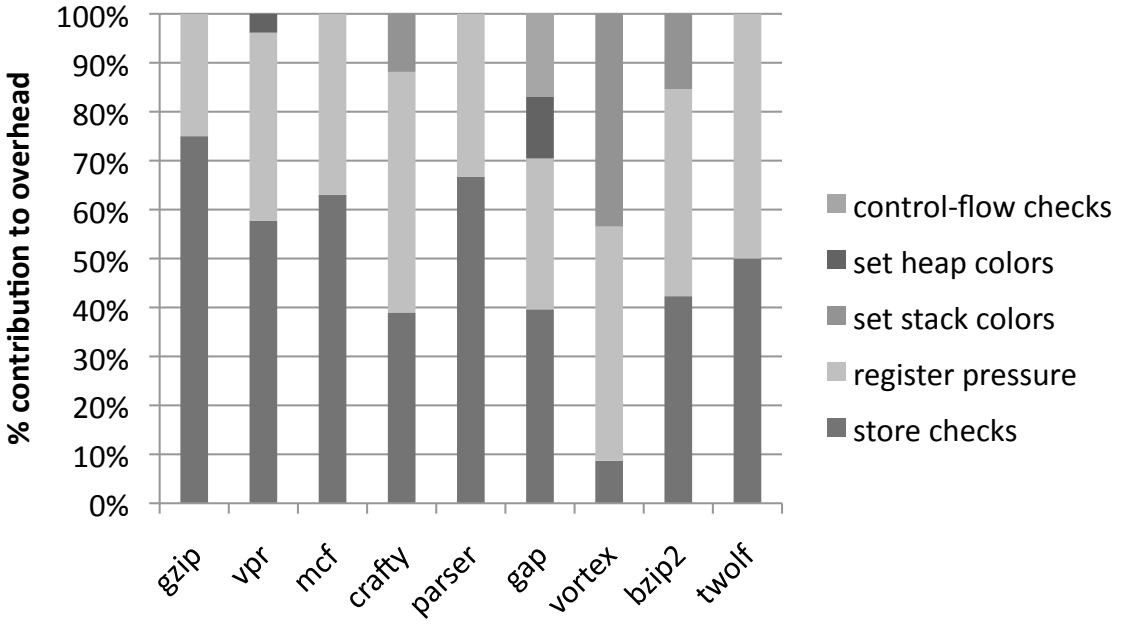


Figure 4.7: Contributions to CPU overhead of write integrity testing.

I also experimented with a version of the WIT runtime that adds 8 bytes of padding at the end of each heap object to protect heap metadata from hypothetical corruption by misaligned writes. In addition to wasted space, poor cache utilisation also increased the execution time. The average time to complete the SPEC benchmarks increased from 10% to 11% and the average memory overhead increased from 13% to 15%. The average time to complete the Olden benchmarks increased from 4% to 7% and the average memory overhead increased from 13% to 63%. The overhead increased significantly in the Olden benchmarks because there are many small allocations. As discussed earlier, it seems likely that the increased security does not justify the extra overhead.

4.6.2 Overhead for a web server

The benchmarks used in the previous sections are CPU-intensive. They spend most time executing instrumented code at user level. The instrumentation overhead is

likely to be higher in these benchmarks than in other programs where it would be masked by other overheads. Therefore, I also measured the overhead added by the instrumentation to the `NullHTTPd` web server [111] running the SPEC WEB 1999 [156] benchmark.

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

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

4.6.3 Synthetic exploits

I ran a benchmark described by Wilander and Kamkar [166] containing 18 attacks exploiting buffer-overflow vulnerabilities. The attacks are classified according to the technique they use to overwrite their target, the location of the buffer they overflow, and the type of the target. It uses two techniques to overwrite targets, such as control-flow data. The first overflows a buffer until the target is overwritten. The second overflows a buffer until a pointer is overwritten, and uses an assignment through the pointer to overwrite the final target. The attacks can overflow buffers located in the stack or in the data segment, and they can target four types of data: the return address on the stack, the old base pointer on the stack, and function pointers and longjmp buffers in the stack or in the data segment.

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

4.6.4 Real vulnerabilities

The final experiments test WIT's ability to prevent attacks that exploit real vulnerabilities in `SQL Server`, `Ghttpd`, `NullHTTPd`, `Stunnel`, and `libpng`. The results are summarised in Table 4.1.

Application	Vulnerability	Exploit	Detected?
`NullHTTPd`	heap-based buffer overflow	overwrite cgi-bin configuration data	yes
`SQL Server`	stack-based buffer overflow	overwrite return address	yes
`Stunnel`	format string	overwrite return address	yes
`Ghttpd`	stack-based buffer overflow	overwrite return address	yes
`Libpng`	stack-based buffer overflow	overwrite return address	yes

Table 4.1: Real attacks detected by WIT.

`SQL Server` is a relational database from Microsoft that was infected by the infamous Slammer [102] worm. The vulnerability exploited by Slammer causes `sprintf` to overflow a stack buffer. WIT was used to compile the `SQL Server` library with the vulnerability. WIT detects Slammer when the `sprintf` function tries to write over the guard object inserted after the vulnerable buffer.

`Ghttpd` is an HTTP server with several vulnerabilities [63]. The vulnerability tested is a stack-buffer overflow when logging `GET` requests inside a call to `vsprintf`. WIT detects attacks that exploit this vulnerability when `vsprintf` tries to write over the guard object at the end of the buffer.

`NullHTTPd` is another HTTP server. This server has a heap-overflow vulnerability that can be exploited by sending HTTP `POST` requests with a negative content length field [64]. These requests cause the server to allocate a heap buffer that is too small to hold the data in the request. While calling `recv` to read the `POST` data into the buffer, the server overwrites the heap-management data structures maintained by the C library. This vulnerability can be exploited to overwrite arbitrary words in memory. I attacked `NullHTTPd` using the technique described by Chen *et al.* [37]. This attack inspired the example in Section 4.1, and works by corrupting the CGI-BIN configuration string. This string identifies a directory holding programs that may be executed while processing HTTP requests. Therefore, by corrupting it, the attacker can force `NullHTTPd` to run arbitrary programs. This is a non-control-data attack because the attacker does not subvert the intended control-flow in the server. WIT detects the attack when the wrapper for the `recv` call is about to write to the guard object at the end of the buffer.

`Stunnel` is a generic tunnelling service that encrypts TCP connections using SSL. I studied a format-string vulnerability in the code that establishes a tunnel for SMTP [65]. An attacker can overflow a stack buffer by sending a message that is passed as a format string to the `vsprintf` function. WIT detects the attack when `vsprintf` attempts to write the guard object at the end of the buffer.

`Libpng` is a library for processing images in the PNG file format [26]. Many applications use `libpng` to display images. I built a test application distributed with `libpng` and attacked it using the vulnerability described in [155]. The attacker can supply a malformed image file that causes the application to overflow a stack buffer. WIT detects the attack when a guard object is about to be written.

4.6.5 Analysis precision

The precision of the points-to analysis and its impact on security was evaluated experimentally. WIT was used to compile nine programs from the SPEC CINT2000 benchmark suite [156] (`gzip`, `vpr`, `mcf`, `crafty`, `parser`, `gap`, `vortex`, `bzip2` and `twolf`). During compilation, the number of colours used in each benchmark and the number of memory write instructions with each colour were collected. Then, at runtime, the maximum number of *objects* with each colour at any given time was measured, where an object is a local variable, a global variable, or an object allocated dynamically using `malloc`, `calloc`, `realloc`, or `alloca`. Combining these measurements provides an upper bound on the number of objects writable by each instruction at runtime. This upper bound is computed assuming a vulnerability that allows an unsafe instruction to write to any object with the same colour as the instruction, ignoring constraints imposed by the program code and guards.

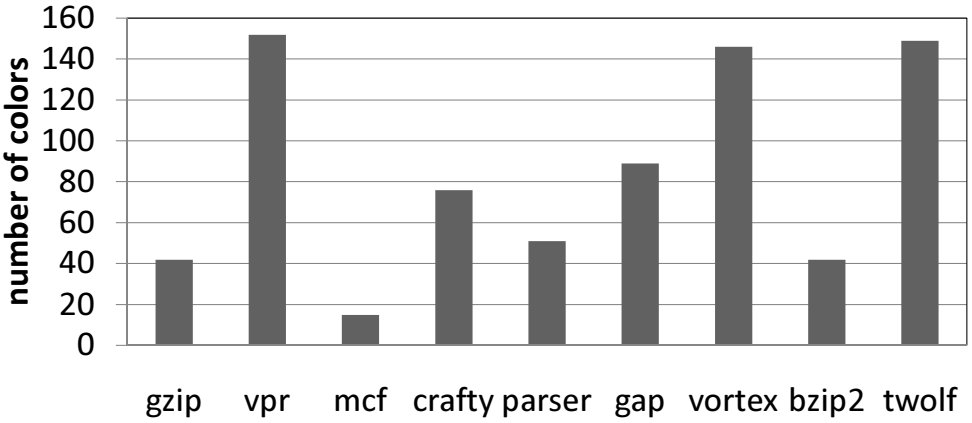


Figure 4.8: Number of colours for SPEC benchmarks.

Figure 4.8 shows the number of colours used by objects and functions in these benchmarks, and Figure 4.9 shows a cumulative distribution of the fraction of memory write instructions versus the upper bound on the number of objects writable by each instruction. For example, the first graph in Figure 4.9 shows that 88% of the memory write instructions in `bzip` can write at most one object at runtime, 99.5% can write at most two objects, and all instructions can write at most three objects. Therefore, even in this worst case, the attacker can only use a pointer to one object to write to another in 12% of the write instructions and in 96% of these instructions it can write to at most one other object. In practice, the program code and the guards will further reduce the sets of objects writable by each instruction.

The results in Figure 4.9 show that the precision of the points-to analysis can vary significantly from one application to the other. For all applications except `mcf` and

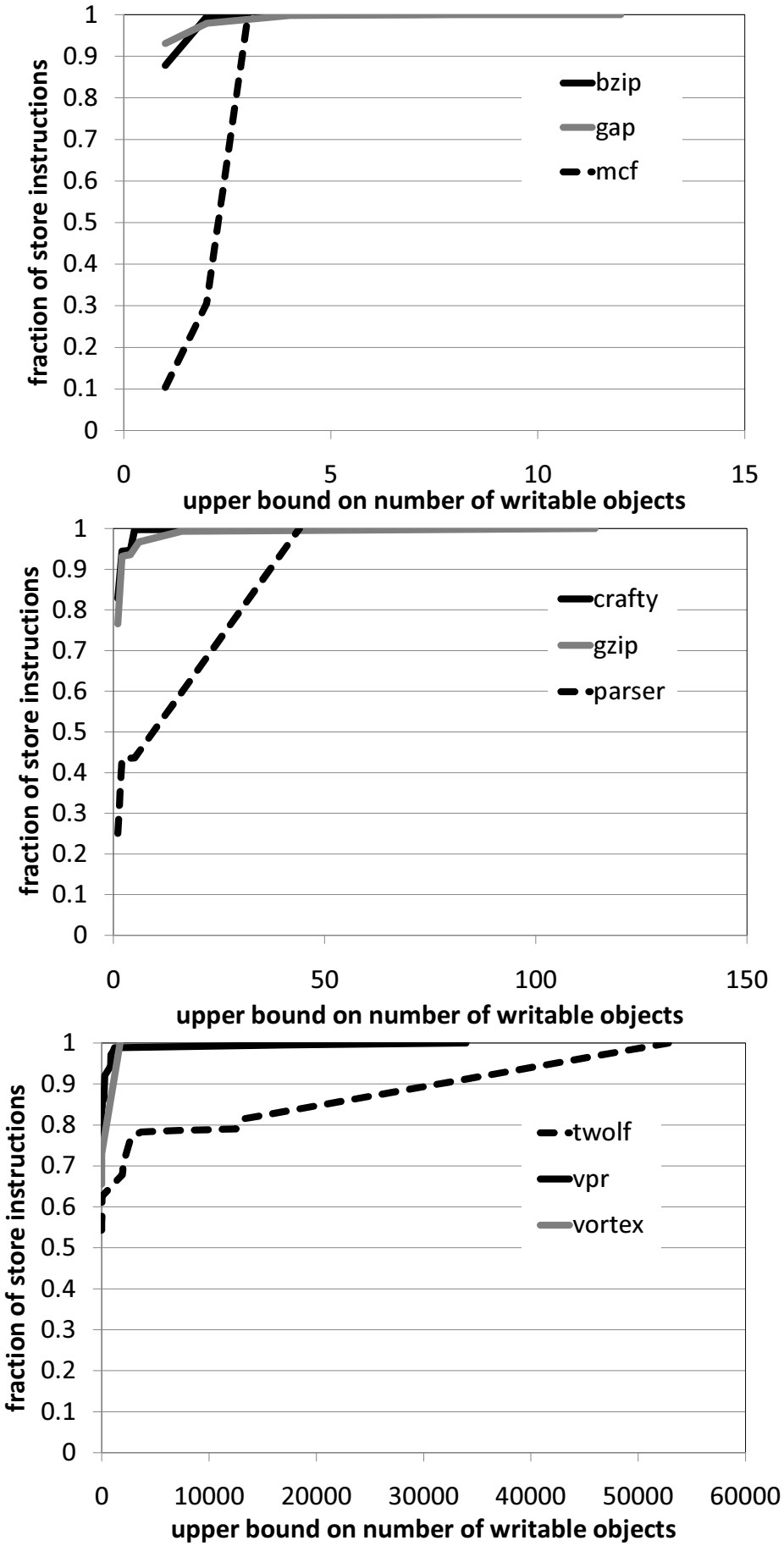


Figure 4.9: Cumulative distribution of the fraction of store instructions versus the upper bound on the number of objects writable by each instruction.

`parser`, the attacker cannot make the majority of instructions write to incorrect objects. For `bzip`, `gap`, `crafty`, and `gzip`, 93% of the instructions can write to at most one incorrect object in the worst case. The precision is worse for `twolf`, `vpr` and `vortex` because they allocate many objects dynamically. However, the fraction of instructions that can write a large number of objects is relatively small. These constraints can render a fraction of the bugs in a program unexploitable by preventing compromised write instructions from affecting security critical data. The fraction varies depending on the precision of the analysis and the location of bugs and security critical data in the program, but if a bug is covered, the protection cannot be worked around.

4.7 Discussion

WIT improves the efficiency of runtime checks by giving up direct detection of invalid memory reads and approximating the correctness of memory writes. Nevertheless, it guarantees the detection of sequential buffer-overflows and prevents branching to arbitrary control-flow targets, and it can further increase the protection depending on the precision of a pointer analysis. Interestingly, WIT improves protection over BBC in three cases: it addresses some temporal memory-errors and memory errors inside structures by constraining subsequent uses of hijacked pointers, and it provides control-flow integrity.

The key advantages, however, are that WIT is 6 times faster on average than BBC for the SPEC benchmarks; achieves consistently good performance by avoiding a slow path for handling out-of-bounds pointers; and is more immune to false positives. These benefits justify the required tradeoffs.

BBC and WIT were designed for user-space programs. The next chapter discusses practical memory safety for kernel extensions.

Chapter 5

Byte-granularity isolation

Bugs in kernel extensions written in C and C++ remain one of the main causes of poor operating-system reliability despite proposed techniques that isolate extensions in separate protection domains to contain faults. Previous fault-isolation techniques for commodity systems either incur high overheads to support unmodified kernel extensions, or require porting for efficient execution. Low-overhead isolation of existing kernel extensions on standard hardware is a hard problem because extensions communicate with the kernel using a complex interface, and they communicate frequently.

This chapter discusses byte-granularity isolation (BGI), a new software fault isolation technique that addresses this problem. BGI is based on WIT, and can also detect common types of memory errors inside kernel extensions (those that can be detected by WIT using a single colour for unsafe objects), but extends WIT's mechanisms to isolate kernel extensions in separate protection domains that share the same address space. Like WIT, its threat model does not aim to protect confidentiality or mitigate malicious kernel extensions, but adds protection for the availability of the kernel in the presence of memory errors in its extensions. The solution is efficient and offers fine-grained temporal and spatial protection that makes it possible to support legacy APIs like the Windows Driver Model (WDM) [114] that is used by the vast majority of Windows extensions. BGI also ensures type safety for kernel objects by checking that API functions are passed objects of the expected type at runtime.

5.1 Overview

Figure 5.1 highlights some of the difficulties in isolating existing kernel extensions. It shows how a simplified file system driver might process a read request in the Windows Driver Model (WDM) [114]. This example illustrates how BGI works. Error handling is omitted for clarity.

At load time, the driver registers the function *ProcessRead* with the kernel. When an application reads data from a file, the kernel creates an I/O Request Packet (IRP) to describe the request and calls the driver to process it. `ProcessRead` sends the request to a disk driver to read the data from disk and then decrypts the data (by XORing the data with the key). `SetParametersForDisk` and `DiskReadDone` are driver functions used by `ProcessRead` that are not shown in the figure.

```
1  void ProcessRead(PDEVICE_OBJECT d, IRP *irp) {
2    KEVENT e;
3    int j = 0;
4    PIO_STACK_LOCATION isp;
5
6    KeInitializeEvent(&e, NotificationEvent, FALSE);
7
8    SetParametersForDisk(irp);
9    IoSetCompletionRoutine(irp, &DiskReadDone, &e,
10                              TRUE, TRUE, TRUE);
11   IoCallDriver(diskDevice, irp);
12   KeWaitForSingleObject(&e, Executive, KernelMode,
13                              FALSE, NULL);
14
15   isp = IoGetCurrentStackLocation(irp);
16   for (; j < isp->Parameters.Read.Length; j++) {
17     irp->AssociatedIrp.SystemBuffer[j] ^= key;
18   }
19   IoCompleteRequest(irp, IO_DISK_INCREMENT);
```

Figure 5.1: Example extension code: processing a read request in an encrypted file system driver.

`ProcessRead` starts by initialising an event synchronisation object `e` allocated on the stack. Then it modifies the IRP to tell the disk driver what to do. Windows drivers are stacked and IRPs contain a stack with locations for each driver holding its parameters and a completion routine that works like a return address. `ProcessRead` sets the parameters for the disk in the next IRP stack location using the driver function `SetParametersForDisk` and it sets `DiskReadDone` as the completion routine (with argument `&e`). Then is passes the IRP to the disk driver and waits on `e`. When the disk read completes, the kernel calls the `DiskReadDone` function, which signals event `e`. This unblocks `ProcessRead` allowing it to decrypt the data read from disk. When it finishes, the driver calls `IoCompleteRequest` to signal the kernel it has completed processing the request.

The example shows some of the complexity of the WDM interface and it shows that drivers interact with the kernel frequently. The vast majority of Windows kernel extensions use the WDM interface, making any practical solution have a significant impact. However, previous fault isolation techniques targeting existing extensions and commodity operating systems [51, 163, 97, 149, 147, 174] are unable to provide strong isolation guarantees for WDM drivers with acceptable overhead.

Some previous solutions, such as SafeDrive by Zhou *et al.* [174], do not prevent temporal errors; for example, if `DiskReadDone` erroneously completes the IRP, the decryption code might overwrite a deleted buffer. Software-based fault isolation solutions [163, 97, 159] offer coarse-grained protection by limiting extensions to only write within a memory range. Since the stacks and heaps used by extensions lie within this range, these techniques cannot prevent writes to kernel objects stored in this memory

while they are being used by the kernel. Furthermore, these solutions fail to ensure that these objects are initialised before they are used. These are serious limitations, as corrupted kernel objects in driver memory can propagate corruption to the kernel's fault domain. For example, they do not prevent writes to event `e` in the example above. This is a problem because the event object includes a linked list header that is used internally by the kernel. Thus, by overwriting or failing to initialise the event, the driver can cause the kernel to write to arbitrary addresses.

These techniques also perform poorly when extensions interact with the kernel frequently because they copy objects passed by reference in cross-domain calls. For example, they would copy the buffer to allow *ProcessRead* to write to the buffer during decryption. XFI by Erlingsson *et al.* [159] avoids the copy but falls back to slower checks for such cases. Finally, solutions relying on hardware page protection for isolation such as Nooks by Swift *et al.* [149, 147] incur additional overhead when switching domains, which is a frequent operation in device drivers that often perform little computation and lots of communication.

BGI is designed to have adequate spatial and temporal resolution to avoid these problems. It can grant access to precisely the bytes that a domain should access, and can control precisely when the domain is allowed to access these bytes because it can grant and revoke access efficiently. Therefore, BGI can provide strong isolation guarantees for WDM drivers with low overhead and no changes to the source code.

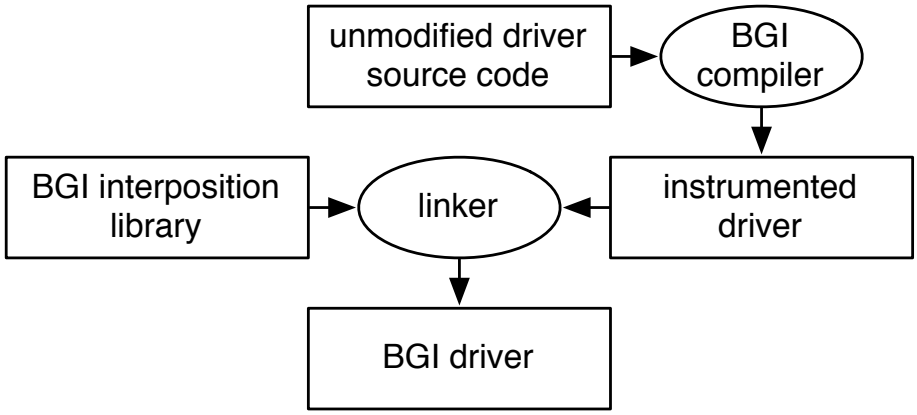


Figure 5.2: Producing a BGI extension.

BGI assigns an access control list (ACL) to every byte of virtual memory (Section 5.2). To store ACLs compactly, they are encoded as small integers, so BGI can store them in a table with 1 byte for every 8 bytes of virtual memory—as in WIT—when all bytes in the 8-byte slot share the ACL and the ACL has a single entry (assuming domains with default access are not listed). It adds, however, a slow path to cover the general case when not all bytes in a slot share an ACL, and reserves a table entry value to indicate such cases. WIT's compile-time changes to the layout of data ensure this slow path is rarely taken. Support for the general case, however, is still necessary in BGI because, unlike in WIT, individual structure fields in kernel objects can have different ACLs, and fields cannot be aligned to avoid breaking binary compatibility with the rest of the kernel.

BGI is not designed to isolate malicious code. It assumes that attackers can control the input to a driver but they do not write the driver code. It is designed to contain faults due to errors in the driver and to contain attacks that exploit these errors. Device-driver writers are trusted to use the BGI compiler.

Windows kernel extensions are written in C or C++. To isolate a Windows kernel extension using BGI, a user compiles the extension with the BGI compiler and links it with the BGI interposition library, as shown in Figure 5.2. BGI-isolated extensions can run alongside trusted extensions on the same system, as shown in Figure 5.3. The instrumentation inserted by the compiler along with the interposition library can enforce fault isolation as described in the following sections.

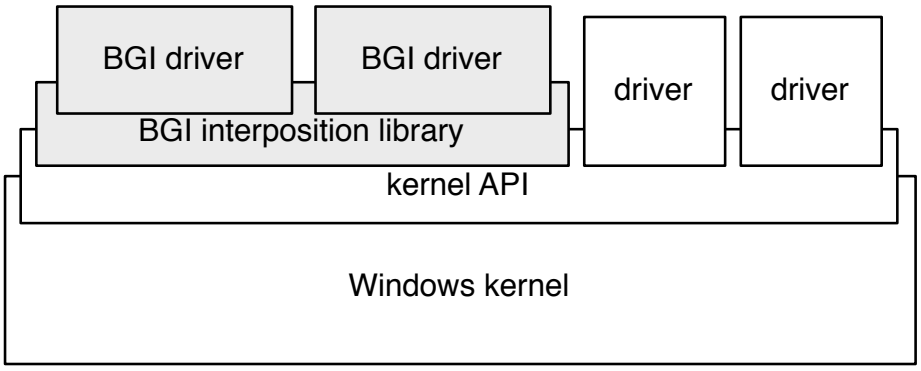


Figure 5.3: Kernel address space with BGI.

5.2 Protection model

BGI runs code in separate protection domains to contain faults. There is one trusted domain where the kernel and trusted extensions run and one or more untrusted domains. Each untrusted domain can run one or more untrusted extensions. A thread is said to be running in a domain when it executes code that runs in that domain. BGI does not constrain memory accesses performed by threads running in the trusted domain.

Virtual memory includes both *system space* and *user space*. System space is shared by all processes while each process has a private user space. All BGI domains share the system space and all the process user spaces, but BGI associates an ACL with each byte of virtual memory to control memory accesses. An ACL lists the domains that can access that particular byte, and the access rights they have to it. Some extensions can directly access the user-space memory of the process that initiates the I/O operation. Note that extensions never access physical addresses directly.

The basic access rights in BGI are *read* and *write*. There are also several *icall*, *type*, and *ownership* rights. All access rights allow read access and, initially, all domains have the read right to every byte in virtual memory. Furthermore, a domain cannot have more than one of these rights to the same byte of virtual memory. Like WIT, BGI does not constrain read accesses, because checking reads in software is expensive and

the other checks prevent errors due to incorrect reads from propagating outside the domain.

BGI grants an *icall* right on the first byte of functions that can be called indirectly or passed in kernel calls that expect function pointers. This is used to prevent extensions from bypassing checks and to ensure that control-flow transfers across domains target only allowed entry points. Cross-domain control transfers through indirect calls need to check for this right, but are otherwise implemented as simple function calls without stack switches, extra copies, or page table changes.

Type rights are used to enforce dynamic type safety for kernel objects. There is a different type right for each type of kernel object, for example, mutex, device, I/O request packet, and deferred procedure call objects. When an extension initialises a kernel object stored in the extension's memory, the interposition library grants the appropriate type right on the first byte of the object, and grants write access to the fields that are writable by the extension according to the API rules and revokes write access to the rest of the object. Access rights change as the extension interacts with the kernel; for example, rights are revoked when the object is deleted, or the extension no longer requires access to the object (*e.g.* when a driver completes processing an I/O request packet). The interposition library also checks if extensions pass objects of the expected type in cross-domain calls. This ensures type safety: extensions can only create, delete, and manipulate kernel objects using the appropriate interface functions. Type safety prevents many incorrect uses of the kernel interface.

The last type of right, ownership rights, are used to keep track of the domain that should free an allocated object and which deallocation function to use. The ownership rights for objects are associated with their adjacent guards, rather than objects' bytes (these guards were discussed in Section 4.3.2).

The interposition library and the code inserted by the compiler use two primitives to manipulate ACLs: *SetRight*, which grants and revokes access rights, and *CheckRight*, which checks access rights before memory accesses. When a thread running in domain d calls $\mathit{SetRight}(p, s, r)$ with $r \neq \mathit{read}$, BGI grants d access right r to all the bytes in the range $[p, p + s)$. For example, $\mathit{SetRight}(p, s, \mathit{write})$ gives the domain write access to the byte range. To revoke any other access rights to the byte range $[p, p + s)$, a thread running in the domain calls $\mathit{SetRight}(p, s, \mathit{read})$. To check ACLs before a memory access to byte range $[p, p + s)$, a thread running in domain d calls $\mathit{CheckRight}(p, s, r)$. If d does not have right r to all the bytes in the range, BGI raises a software interrupt. BGI defines variants of *SetRight* and *CheckRight* that are used with *icall* and *type* rights. $\mathit{SetType}(p, s, r)$ marks p as the start of an object of type r and size s that can be used by the domain, and also prevents writes to the range $[p, p + s)$. $\mathit{CheckType}(p, r)$ is equivalent to $\mathit{CheckRight}(p, 1, r)$.

Figure 5.4 shows an example partition of five kernel drivers into domains. Driver 1 runs in the trusted domain with the kernel code because it is trusted. The other drivers are partitioned into two untrusted domains. There is a single driver in domain d_2 but there are three drivers in domain d_1. Frequently, the functionality needed to drive a device is implemented by multiple driver binaries that communicate directly through custom interfaces. These drivers should be placed in the same domain. Figure 5.5 shows example ACLs for the domains in Figure 5.4. The greyed boxes correspond to the default ACL that only allows domains to read the byte. The other ACLs grant

some of the domains more rights on accesses to the corresponding byte; for example, one of them grants domains d_1 and d_2 the right to use a shared lock.

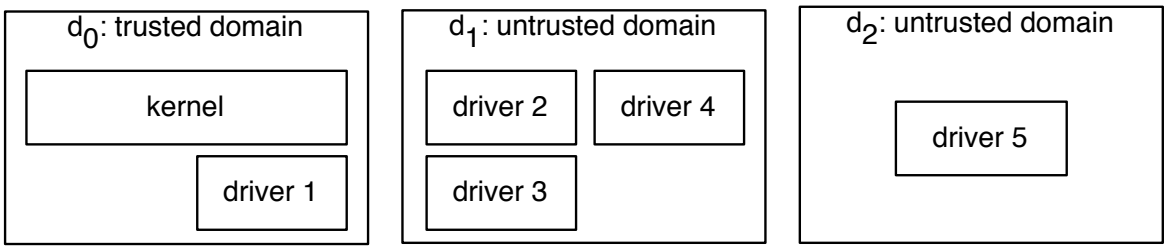


Figure 5.4: Example partition into domains.

5.3 Interposition library

An untrusted extension is linked with the BGI interposition library that mediates all communication between the extension and the kernel. The library contains two types of wrappers: *kernel wrappers* which wrap kernel functions that are called by the extension (*e.g.* `KeInitializeEvent` and `IoSetCompletionRoutine` in the example of Figure 5.1) and *extension wrappers* which wrap extension functions that are called by the kernel (*e.g.* `ProcessRead` and `DiskReadDone` in the example). Wrappers use the memory protection primitives to grant, revoke, and check access rights according to the semantics of the functions they wrap.

Since these primitives target the domain of the thread that calls them, wrappers execute in the extension's domain, even though they are trusted. A kernel wrapper does not trust the caller but trusts the callee. It checks rights to the arguments supplied by the extension, and may revoke rights to some of those arguments. It next calls the wrapped kernel function, and it may grant rights to some objects returned by the function. The sequence of steps executed by an extension wrapper is different, because the caller is trusted but the callee is not. An extension wrapper may grant rights to some arguments, and then will call the wrapped extension function. After its return, it may revoke rights to some arguments, and check values returned by the extension. Different types of argument and result checks are used for different types of rights: *write*, *ownership*, *icall*, and *type*.

Write and ownership checks prevent extension errors from making threads in other domains write to arbitrary locations. A kernel wrapper calls $\mathit{CheckRight}(p, s, \mathit{write})$ for each memory range argument, $[p, p+s)$, that may be written to by the function being wrapped. An extension wrapper performs similar checks on memory ranges returned by the extension that may be written to by the caller.

Write access is granted and revoked by the interposition library and code inserted by the compiler. The wrapper for the extension initialisation function grants write access to static variables. Write access to automatic variables is granted and revoked by code inserted by the compiler. Write access to dynamic variables is granted by the kernel wrappers for heap allocation functions, and revoked by the wrappers for deallocation functions. The wrappers for allocation functions also grant *ownership*

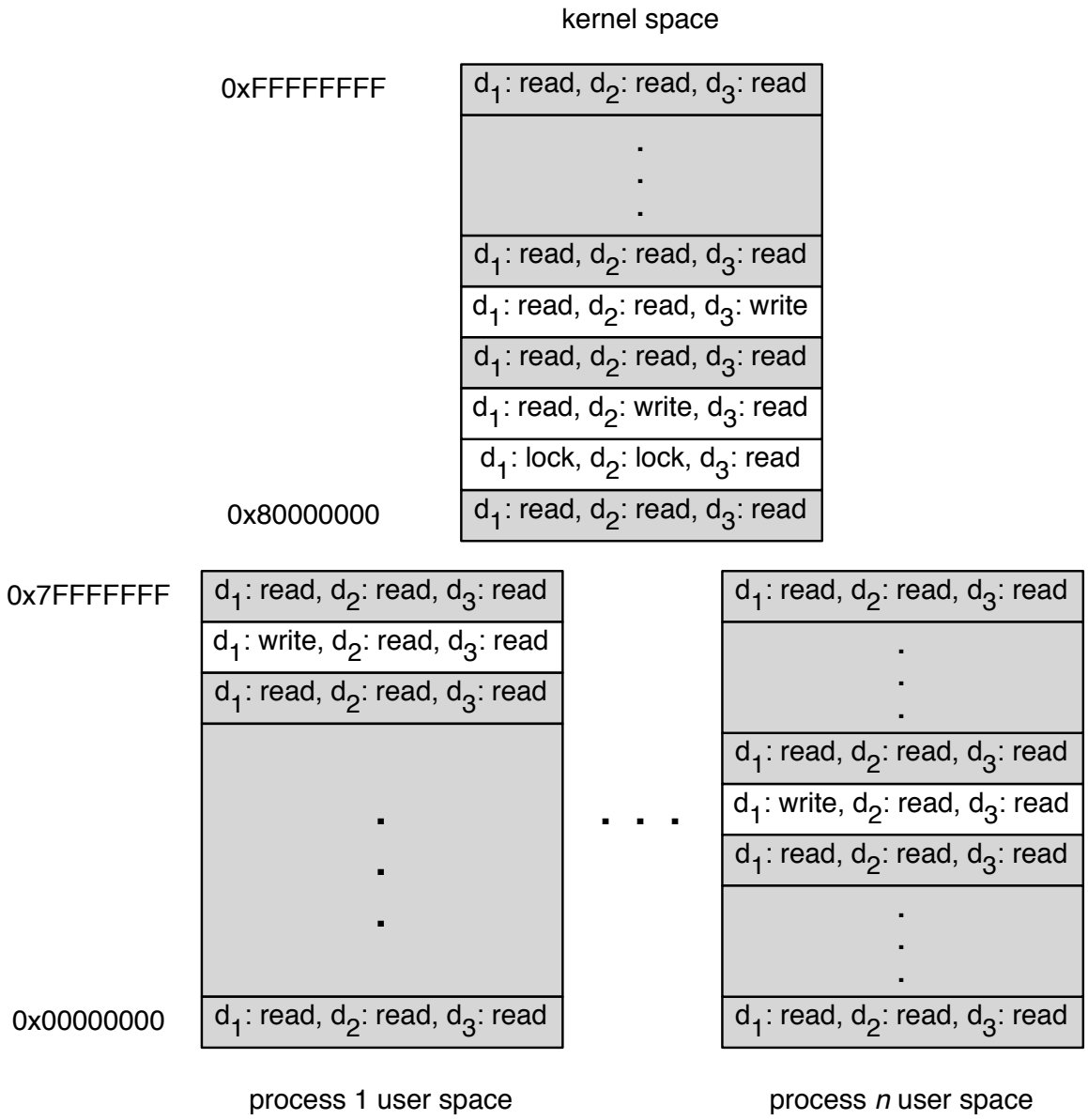


Figure 5.5: Example access control lists (ACLs).

rights on a guard (Section 4.3.2) before or after the allocated memory (depending on whether the allocation is smaller or larger than a page). *Ownership* rights are used to identify the allocation function and the domain that allocated the memory. The wrappers for deallocation functions check that the calling domain has the appropriate ownership right and that is has write access to the region being freed. If these checks fail, they signal an error. Otherwise, they revoke the ownership right and write access to the memory being freed. This ensures that only the domain that owns an object can free the object, that it must use the correct deallocation function, and that an object can be freed at most once.

Call checks prevent extension errors from making threads in other domains execute arbitrary code. Some kernel functions take function pointer arguments. Since the

kernel may call the functions they point to, the interposition library checks if the extension has the appropriate *icall* right to these functions. Kernel wrappers call $\mathit{CheckType}(p, \mathit{icall}N)$ on each function pointer argument p before calling the kernel function they wrap, where N is the number of stack bytes used by the arguments to an indirect call through p. The `stdcall` calling convention used in Windows drivers requires the callee to remove its arguments from the stack before returning (and does not support the `vararg` feature). Therefore, the *icall* rights encode N to prevent stack corruption when functions with the wrong type are called indirectly. Conversely, extension wrappers check function pointers returned by extension functions.

The *icall* rights are also granted and revoked by the interposition library with help from the compiler. The compiler collects the addresses of all functions whose address is taken by the extension code and the number of bytes consumed by their arguments on the stack. This information is stored in a section in the extension binary. The wrapper for the driver initialisation function calls $\mathit{SetType}(p, 1, \mathit{icall}N)$ for every pair of function address p and byte count N in this section to associate an *icall* right with the entry point of the function. When kernel functions return function pointers, their wrappers replace these pointers by pointers to the corresponding kernel wrappers and grant the appropriate *icall* rights. Since BGI does not grant *icall* rights in any other case, cross-domain calls into the domain can only target valid entry points: functions whose address was taken in the code of an extension running in the domain and kernel wrappers whose pointers were returned by the interposition library.

Type checks are used to enforce a form of dynamic type safety for kernel objects. There is a different *type* right for each type of kernel object. When a kernel function allocates or initialises a kernel object with address p, size s, and type t, its wrapper calls $\mathit{SetType}(p, s, t)$ and grants write access to any fields that can be written directly by the extension. The wrappers for kernel functions that receive kernel objects as arguments check if the extension has the appropriate *type* right to those arguments, and wrappers for kernel functions that deallocate or uninitialise objects revoke the *type* right to the objects. Since many kernel objects can be stored in heap or stack memory allocated by the extension, BGI also checks if this memory holds active kernel objects when it is freed. Together, these checks ensure that extensions can only create, delete, and manipulate kernel objects using the appropriate kernel functions. Moreover, extensions in an untrusted domain can only use objects that were received from the kernel by a thread running in the domain.

The type checks performed by BGI go beyond traditional type checking because the type, *i.e.*, the set of operations that are allowed on an object, changes as the extension interacts with the kernel. BGI implements a form of dynamic typestate [143] analysis. For example, in Figure 5.1, the extension wrapper for `ProcessRead` grants *irp* right to the first byte of the IRP. This allows calling `IoSetCompletionRoutine` and `IoCallDriver`, that check if they receive a valid IRP. But the *irp* right is revoked by the wrapper for `IoCallDriver` to prevent modifications to the IRP while it is used by the disk driver. The extension wrapper for `DiskReadDone` grants the *irp* right back after the disk driver is done. Then the right is revoked by the wrapper for `IoCompleteRequest` because the IRP is deleted after completion. These checks enforce interface usage rules that are documented but were not previously enforced.

In addition to using access rights to encode object state, some kernel wrappers use information in the fields of objects to decide whether a function can be called without corrupting the kernel. This is safe because BGI prevents the extension from modifying these fields.

```
1  VOID
2  _BGI_KeInitializeDpc(PRKDPC d,
3        PKDEFERRED_ROUTINE routine, PVOID a) {
4    CheckRight(d, sizeof(KDPC), write);
5    CheckFuncType(routine, PKDEFERRED_ROUTINE);
6    KeInitializeDpc(d, routine, a);
7    SetType(d, sizeof(KDPC), dpc);
8  }
9
10 BOOLEAN
11 _BGI_KeInsertQueueDpc(PRKDPC d, PVOID a1, PVOID a2) {
12    CheckType(d, dpc);
13    return KeInsertQueueDpc(d, a1, a2);
14 }
```

Figure 5.6: Example kernel wrappers for KeInitializeDpc and KeInsertQueueDpc.

Figure 5.6 shows two example kernel wrappers. The first one wraps the `KeInitializeDpc` function that initialises a data structure called a deferred procedure call (DPC). The arguments are a pointer to a memory location supplied by the extension that is used to store the DPC, a pointer to an extension function that will be later called by the kernel, and a pointer argument to that function. The wrapper starts by calling $\mathit{CheckRight}(d, \mathit{sizeof}(\mathit{KDPC}), \mathit{write})$ to check if the extension has write access to the memory region where the kernel is going to store the DPC. Then it checks if the extension has the appropriate *icall* right to the function pointer argument. `CheckFuncType` is a macro that converts the function pointer type into an appropriate *icall* right and calls *CheckType*. In this case, it calls $\mathit{CheckType}(\mathit{routine}, \mathit{icall}16)$, where 16 is the number of stack bytes used by the arguments to an indirect call to a routine of type `PKDEFERRED_ROUTINE`. If these checks succeed, the DPC is initialised and the wrapper grants the *dpc* right to the extension for the byte pointed to by *d*; note that this simultaneously revokes write access to the DPC object. It is critical to prevent the extension from writing directly to the object because it contains a function pointer and linked list pointers. If the extension corrupted the DPC object, it could make the kernel execute arbitrary code or write to an arbitrary location. `KeInsertQueueDpc` is one of the kernel functions that manipulate DPC objects. Its wrapper performs a type check to ensure that the first argument points to a valid DPC. These type checks prevent several incorrect uses of the interface, including preventing a DPC object from being initialised more than once or being used before it is initialised.

In Figure 5.6, the wrapper for `KeInitializeDpc` passes the function pointer `routine` to the kernel. In some cases, the wrapper replaces the function pointer supplied by the extension by a pointer to an appropriate extension wrapper and replaces its

pointer argument with a pointer to a structure containing the original function and its argument. This is not necessary in the example because `routine` returns no values and it is invoked with arguments that the extension already has the appropriate rights to (`d`, `a`, `a1`, and `a2`).

In collaboration with Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Austin Donnelly, Paul Barham, and Richard Black [33], I implemented 262 kernel wrappers and 88 extension wrappers. These cover the most common WDM, WDF, and NDIS interface functions [101] and include all interface functions used by the drivers in the experiments. Most of the wrappers are as simple as the ones in Figure 5.6 and could be generated automatically from source annotations similar to those proposed by Hackett *et al.* [73] and Zhou *et al.* [174]. There are 16700 lines of code in the interposition library. Although writing wrappers represents a significant amount of work, it only needs to be done once for each interface function by the OS vendor. Driver writers do not need to write wrappers or change their source code.

5.4 Instrumentation

The BGI compiler inserts four types of instrumentation in untrusted extensions. It redirects kernel function calls to their wrappers in the interposition library, adds code to function prologues and epilogues to grant and revoke access rights for automatic variables, and adds access-right checks before writes and indirect calls.

The compiler rewrites all calls to kernel functions to call the corresponding wrappers to ensure that all communication between untrusted extensions and the kernel is mediated by the interposition library. The compiler also modifies extension code that takes the address of a kernel function to take the address of the corresponding kernel wrapper in the interposition library. This ensures that indirect calls to kernel functions are also redirected to the interposition library.

The compiler inserts calls to *SetRight* in function prologues to grant the calling domain write access to local variables on function entry. In the example in Figure 5.1, it inserts *SetRight*(`&e, sizeof(e)`, *write*) in the prologue of the function `ProcessRead`. To revoke access to local variables on function exit, the compiler modifies function epilogues to first verify that local variables do not store active kernel objects and then call *SetRight* to revoke access.

The compiler inserts a check before each write in the extension code to check if the domain has write access to the target memory locations. It inserts *CheckRight*$(p, s, write)$ before a write of s bytes to address p. The compiler also inserts checks before indirect calls in the extension code by adding a *CheckType*$(p, icallN)$ before any indirect call through pointer p that uses N bytes for arguments in the stack.

The checks inserted by the compiler and those performed by the interposition library are sufficient to ensure control-flow integrity: untrusted extensions cannot bypass the checks inserted by the compiler; indirect calls (either in the driver code or on its behalf in the kernel) target functions whose address was taken in the extension code or whose address was returned by the interposition library; returns transfer control back the caller; and, finally, exceptions transfer control to the appropriate han-

dler. Like WIT, BGI does not need additional checks on returns or exception handling because write checks prevent corruption of return addresses and exception handler pointers.

BGI can also prevent many attacks internal to a domain. These are a subset of the attacks detected by WIT that includes the most common attacks. Control-flow integrity prevents the most common attacks that exploit errors in extension code because these attacks require control flow to be transferred to injected code or to chosen locations in code that is already loaded. BGI also prevents sequential buffer overflows and underflows that can be used to mount attacks that do not violate control-flow integrity. These are prevented, as in WIT, when write checks detect attempts to overwrite guards placed between consecutive dynamic, static, and automatic memory allocations.

The instrumentation inserted by BGI uses a table similar to WIT and BBC to stores ACLs in the form of small integers. This enables an efficient software implementation for enforcing byte-granularity memory protection.

5.4.1 Encoding rights

BGI stores ACLs compactly by encoding a domain and an access right as a small integer, a *d-right* (short for domain right). When a protection domain is instantiated, a number of distinct access rights is reserved for it, equal to the number of distinct access rights used by the extensions in the domain. These d-rights are unique across domains except for the one corresponding to the default *read* access right, which is zero for all domains. When a domain is destroyed, its d-rights can be reclaimed for use by other domains.

Two optimisations are necessary to reduce the number of bits needed to encode d-rights. First, the number of distinct access rights used in extensions is reduced by exploiting a form of subtyping polymorphism in the kernel. Several types of kernel objects have the same supertype: they have a common header with an integer field that indicates the subtype. For example, there are 17 subtypes of *dispatcher object* in the WDM interface, which include events, mutexes, DPCs, and threads. A single type right can be used for the supertype combined with checking the field that indicates the subtype when necessary. This is safe because writes to this field are prevented. This optimisation is very effective because this pattern is common.

Second, all *icall* rights are collapsed into a single d-right per domain, and the number of bytes used by the function arguments is stored as an integer just before the function code. The write checks prevent untrusted extensions from modifying this integer. The indirect-call checks start by checking if the domain has *icall* right to the first byte in the target function and then checks the number of bytes stored before the function. (Faster checks using a separate table for function rights could be used but it would require additional address space; and, in practice, indirect-call checks are not a significant overhead for device drivers.)

Currently, the 32-bit x86 implementation of BGI uses 1-byte d-rights, supporting 256 distinct d-rights. BGI required 47 distinct access rights across 16 test drivers. If every untrusted domain used all these rights, BGI could support five separate untrusted domains. In practice, however, BGI should be able to support 15 or more independent

untrusted domains in addition to the trusted domain, because many rights are used only by a particular type of driver (*e.g.*, a filesystem or a network driver) or by drivers that use a particular version of the interface (*e.g.*, WDF). The number of supported domains should be sufficient for most scenarios because many drivers developed by the operating system vendor may run in the trusted domain, and each untrusted domain may run several related drivers. Moreover, a 64-bit implementation could use 2-byte d-rights, with similar space overhead by using 16-byte slots, raising the number of supported d-rights to 65k and making the solution future proof.

5.4.2 ACL tables

ACLs in BGI are stored in tables, much like colours in WIT and bounds in BBC. An ACL table has a 1-byte entry holding a d-right for every 8 bytes of memory. BGI, however, faces two new challenges. First, kernel extensions may access both kernel memory and user-space memory, *e.g.* some drivers access user-space buffers directly. The kernel space is shared between all processes and can use a single table. Each process, however, has its own user-space memory that needs a separate table in order to protect user-space data from corruption. The second challenge is that multiple d-rights may apply to a single 8-byte memory slot. This can happen because multiple domains may have rights on the same memory, or because a domain does not have the same right for every byte in the slot.

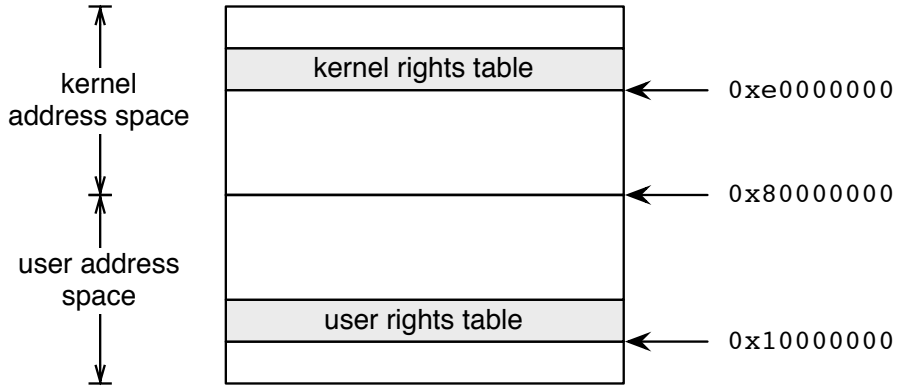


Figure 5.7: Kernel and user ACL tables in an x86 Windows address space.

The first challenge is addressed by reserving virtual address space for the kernel-space table in kernel space and reserving virtual address space for a user-space table in the virtual address space of each process. Therefore, there is a single kernel table and a user table per process that is selected automatically by the virtual memory hardware. Figure 5.7 shows the location of the user and kernel tables in the address space of an x86 Windows operating system. The kernel reserves virtual address space for the kernel table at address `0xe0000000` when the system boots, and reserves virtual address space in every process at address `0x10000000` when the process is created. The kernel allocates physical pages to the tables on demand when they are first accessed and zeroes them to set access rights to *read* for all domains. This prevents

incorrect accesses by default, and also protects the tables themselves from being overwritten. Since some extension code cannot take page faults, the kernel was modified to preallocate physical pages to back kernel table entries that correspond to pinned virtual memory pages.

The same strategy could be used to implement BGI on the x64 architecture. Even though it is necessary to reserve a large amount of virtual memory for the tables in a 64-bit architecture, only top level table entries need to be allocated to do this. Additional page metadata and physical pages only need to be allocated to the tables on demand.

The second challenge is addressed by making multiple d-rights for a slot rare, and handling the remaining cases using a special d-right value *conflict* in the table. A check that encounters the *conflict* right fails, and invokes an error handler. This handler checks whether the table value is *conflict*, and in that case, consults an auxiliary sparse data structure—a conflict table—storing access right information for every byte of the 8-byte slot in question. A conflict table is a splay tree that maps the address of a slot to a list of arrays with 8 d-rights. Each array in the list corresponds to a different domain and each d-right in an array corresponds to a byte in the slot. A kernel conflict-table is used for slots in kernel space and a user conflict-table per process is used for slots in user space. Conflict tables are allocated in kernel space and each process object includes a pointer to its user conflict-table.

5.4.3 Avoiding accesses to conflict tables

To achieve good performance, both in space and in time, BGI must minimise conflict-table accesses using several optimisations to ensure that most slots have a single associated d-right.

First, BGI does not restrict read accesses. Therefore, supporting the common cases of read-read and read-write sharing across domains does not require accesses to conflict tables. It is also rare for two extensions to have non-*read* access rights to the same byte at the same time. In fact, BGI does not allow by design an untrusted domain to have a non-*read* right to a byte of memory that is writable by another untrusted domain. Similarly, different domains never have *icall* rights to the same function because they are only granted *icall* rights to functions in extensions they contain or to wrappers in the copies of the interposition library that is linked statically with those extensions. But it is possible for different extensions to have *type* rights to the same kernel object.

Using techniques from BBC and WIT, BGI also ensures that, for most slots, a domain has the same access right to all the bytes in the slot, by modifying the data layout of global and local variables. BGI uses an 8-byte slot, as in WIT, because dynamic kernel memory allocators in x86 Windows allocate 8-byte aligned memory in multiples of 8 bytes, similar to their user-space counterparts. Local variables are padded as in WIT to overcome the default stack alignment of 4 bytes on 32-bit Windows.

A naive implementation of *SetType* and *CheckType* would always access the kernel conflict table because they set and check access rights to the first byte of an object. BGI takes advantage of the fact that most objects are 8-byte aligned to implement these primitives more efficiently. The optimised implementation of $SetType(p, s, r)$

checks if p is 8-byte aligned and s is at least 8. If this check succeeds, it uses the first 8 bytes to store the type of the object by executing $SetRight(p, 8, r); SetRight(p + 8, s - 8, read)$, which avoids the access to the conflict table. Otherwise, it executes $SetRight(p, 1, r); SetRight(p + 1, s - 1, read)$ as before. Similarly, $CheckType(p, r)$ checks if p is 8-byte aligned and the d-right in the kernel table corresponds to access right r for the domain. Only if this check fails, does it access the conflict table to check if the byte pointed to by p has the appropriate d-right. To further reduce the number of accesses to the conflict table, local variables and fields in local driver `structs` are aligned on 8-byte boundaries if they have a kernel object type. (This can be achieved using appropriate compiler-attributes for the definitions of the kernel object structures in header files). Functions are 16-byte aligned.

A final optimisation avoids accesses to the conflict table while allowing a domain to have different access rights to the bytes in a slot in two common cases: when a domain has right *write* to the first half of the bytes in the slot and *read* to the second half of the bytes, and, conversely, when a domain has right *read* to the first half of the bytes in the slot and *write* to the second half of the bytes. BGI reserves two additional d-rights per domain to encode these cases. This optimisation is effective in avoiding accesses to conflict tables when a domain is granted write access to individual fields in a kernel object whose layout cannot be modified. Such fields are typically 4-byte aligned.

In 64-bit architectures most fields in kernel objects are 8-byte aligned, thus accesses to the conflict table are minimised if 8-byte slots are used on 64-bits. These techniques, however, remain relevant if the slot size on 64-bit architectures is increased to 16 bytes (to allow encoding d-rights using 2 bytes, or to reduce memory overhead).

5.4.4 Table access

The optimisations described above enable implementations of *SetRight* and *CheckRight* with similar efficiency in the common cases to the code sequences of WIT, as shown in Chapter 4. However, some additional functionality is required to handle the BGI-specific cases.

In the common case, *SetRight* sets the access rights to all the bytes in a sequence of slots to *write* or *read*. Figure 5.8 shows an efficient code sequence that implements $SetRight(p, 32, write)$ in this case. In the examples in this section, the d-right for the *write* right and the domain that runs the sample code sequences is 0x02. Since the d-right values are immediates in the code, they must be changed when the extension is added to a domain to match the d-rights allocated to the domain. This is similar to a relocation and requires the compiler to record the locations of these constants in the code.

```
1 mov     eax, ebx
2 sar     eax, 3
3 btc     eax, 0x1C   ; complement specified bit
4 mov     dword ptr [eax], 0x02020202
```

Figure 5.8: Code sequence that implements $SetRight(p, 32, write)$ for the x86.

Initially, pointer p is in register `ebx` and it can point either to kernel or user space. The first instruction moves p into register `eax`. Next, the `sar` and `btc` instructions compute the address of the entry in the right table without checking if p points to kernel or user space and without using the base addresses of the tables. Figure 5.7 helps clarify how this works. Addresses in user space have the most significant bit set to 0 and addresses in kernel space have the most significant bit set to 1. The `sar` instruction shifts `eax` right by 3 bits and makes the 3 most significant bits equal to the most significant bit originally in `eax`. (Note the use of arithmetic shift instead of the logical shift used in the user-space WIT solution.) After executing `sar`, the four most significant bits in `eax` will be `1111` for a kernel address and `0000` for a user address. The `btc` instruction complements the least significant of these 4 bits. So the most significant 4 bits in the result are `0xe` when p is a kernel address and `0x1` when it is a user address. The remaining bits in `eax` are the index of the table entry for the slot pointed to by p. The final `mov` instruction sets four entries in the table to `0x02`, which grants the domain write access to $[p, p + 32)$.

A similar code sequence can be used on the x64 architecture by replacing 32-bit by 64-bit registers (because pointers are 64 bits long), shifting by 4 instead of 3 (to use 16-byte slots), and complementing a different bit (because the table bases would be at different addresses).

```
1       push    eax
2       lea     eax, [ebp-38h]
3       sar     eax, 3
4       btc     eax, 0x1C
5       add     eax, 5
6       xor     dword ptr [eax-4], 0x02020202
7       jne     L1
8   L2: pop     eax
9       ...
10      ret     4
11      ...
12  L1: push    eax
13      lea     eax, [ebp-38h]
14      push    eax
15      push    6
16      call    _BGI_slowRevokeAccess
17      jmp     L2
```

Figure 5.9: Code sequence that revokes access to local variables on function epilogues.

The BGI compiler inserts a code sequence similar to the one in Figure 5.8 in the prologues of instrumented functions to grant write access to local variables. However, the code sequence to revoke write access to local variables on function exit is more complicated because it must check if a local variables stores an active kernel object. Figure 5.9 shows an example. The intuition is to use `xor` instead of `mov` to zero the local variables in a fast path, and check the `xor` instruction's result to determine whether

the values where not the expected ones. If *type* rights have been granted to the bytes in question the check will fail and a slow path is invoked to deal with kernel objects on the stack. The slow path must be able to restore the original d-right values, so the code sequence is modified to ensure `eax` points just after the last XORed byte. In detail, the sequence works as follows. The code stores the address of the guard before the first local variable in `eax` (after saving `eax`) and the `sar` and `btc` instructions compute the address of the kernel table entry for the guard. The `add` instruction updates `eax` to point to the table entry right after the last table entry modified by the xor. It adds 5 to `eax` to account for the guard slot before the local variable and the 4 slots occupied by the variable. If the local variable does not store any kernel object, the `xor` revokes access to the local variable and the branch is not taken. Otherwise, the branch is taken and the `_BGI_slowRevokeAccess` function is called. This function undoes the failed `xor` and checks if the table entries for the slots occupied by the local variable have d-rights corresponding to kernel objects. If it finds an active kernel object, it signals an error. When functions have more local variables, the compiler adds another `add`, `xor`, and `jne` for each variable. The address of the guard before the first local variable, the current value of `eax`, and the number of slots for the stack frame are passed to `_BGI_slowRevokeAccess`. `_BGI_slowRevokeAccess` works out whether the failed `xor` was a four, two, or one byte variant, it undoes all `xor`s up to the byte pointed to by `eax` (non-inclusive), and then scans all the slots for kernel objects.

The BGI compiler also inserts checks before writes and indirect calls. Figure 5.10 shows an efficient code sequence that implements $\mathit{CheckRight}(p, 1, \mathit{write})$. Initially, p is in `ebx`. The code computes the address of the table entry for the slot pointed to by p in the same way as *SetRight*. Then, the `cmp` instruction checks if the entry has d-right `0x02`. If the check fails, the code calls one of the `_BGI_slowCheck` functions. These functions receive a pointer to the memory range being checked and their name encodes the size of the range. In this case, the code calls `_BGI_slowCheck1`, which checks if the table entry contains a d-right that encodes write access to the half slot being accessed and, if this fails, checks the d-right for the appropriate byte in the conflict table. The indirect-call check is similar but it also checks the number of stack bytes stored before the function and it does not have a slow path because functions are always 16-byte aligned.

The BGI compiler uses simple static analysis to eliminate certain *SetRight* and *CheckRight* sequences. It does not add *SetRight* for local variables that are not arrays or `structs` and whose address is not taken. It also eliminates *CheckRight* before the writes to these variables. This analysis is based on the one used in Chapters 3 and 4, but is more conservative, to avoid eliminating checks for memory that may have its access rights change.

The interposition library implements $\mathit{SetRight}(p, s, r)$ similarly to the compiler, but uses `memset` to set d-rights when s is not known statically or is large. Additionally, it must deal with the case where p or $p + s$ are not 8-byte aligned. In this case, the interposition library sets d-rights as before for slots that are completely covered by the byte range, but calls a function to deal with the remaining slots; this function sets the corresponding table entries to the d-rights that encode write access to half a slot, when possible. Otherwise, it records d-rights for individual bytes in the appropriate conflict table. The interposition library implements *CheckRight* as in Figure 5.10, but

it iterates the check for larger memory ranges. To improve performance, it compares four d-rights at a time when checking write access to large memory ranges.

```
1       mov     eax, ebx
2       sar     eax, 3
3       btc     eax, 0x1C
4       cmp     dword ptr [eax], 2
5       je      L1
6       push    ebx
7       call    _BGI_slowCheck1
8   L1:
```

Figure 5.10: Code sequence that implements $CheckRight(p, 1, write)$ for the x86.

5.4.5 Synchronisation

BGI avoids synchronisation on table accesses as much as possible to achieve good performance. It uses enough synchronisation to avoid false positives but may fail to isolate errors in some uncommon racy schedules. It should be hard for attackers to exploit these races to escape containment given the assumption that they do not write the extension code.

In particular, no synchronisation is used for the common case of granting or revoking write access to all the bytes in a slot. Synchronisation is not necessary because: (1) by design, it is an error for an untrusted domain to have non-*read* rights to a byte of memory that is writable by another untrusted domain and (2) if threads running in the same domain attempt to set conflicting d-rights on the same byte, this means there is a pre-existing race in the code. Synchronisation is required, however, when granting and revoking *type* rights and when granting and revoking write access to half a slot. BGI uses atomic compare-and-swap on table entries to prevent false positives due to competing writes. Similarly, synchronisation is required when granting or revoking rights involves an access to a conflict table. An atomic swap is used to record the conflict in the appropriate table entry and a spin lock is used per conflict table. Since these tables are rarely used, contention for the lock is not a problem.

There is no synchronisation in the fast path when checking rights. To prevent false positives, the slow path retries the check after a memory barrier and uses spin locks to synchronise accesses to the conflict tables when needed. Furthermore, the right checks are not atomic with the access they check. This can never lead to false positives but may fail to prevent an invalid access in some schedules, because when the right is revoked there is a race between the check and the access.

5.5 Experimental evaluation

This section describes fault-injection experiments used to evaluate BGI's ability to isolate extensions and performance experiments to measure its overhead. The results show that BGI provides strong isolation guarantees with low overhead.

Table 5.1 lists the kernel extensions used to evaluate BGI. The `classpnp` driver implements common functionality required by storage class drivers like `disk`. The last five drivers sit at the bottom of the USB driver stack in Windows Vista. The `usbehci`, `usbuhci`, `usbohci`, and `usbwhci` drivers implement different USB host controller interfaces and `usbport` implements common functionality shared by these drivers. These 16 extensions have a total of more than 400,000 lines of code and use 350 different functions from WDM, IFS, NDIS, and KMDF interfaces [101]. The source code for the first 8 extensions is available in the Windows Driver Kit [101].

All experiments ran on HP xw4600 workstations with an Intel Core2 Duo CPU at 2.66 GHz and 4GB of RAM, running Windows Vista Enterprise SP1. The workstations were connected with a Buffalo LSW100 100 Mbps switching hub for the Intel PRO/100 driver tests and with an HP 10-GbE CX4 cable for the Neterion Xframe driver tests. Experiments with the FAT file system and disk drivers ran on a freshly formatted Seagate Barracuda ST3160828AS 160GB disk. Experiments with the USB drivers used a SanDisk 2GB USB2.0 Flash drive. The extensions were compiled, with and without BGI instrumentation, using the Phoenix [99] compiler with the -O2 (maximise speed) option. All the performance results are averages of at least three runs.

Extension	#lines	Description
`xframe`	39,952	Neterion Xframe 10Gb/s Ethernet driver
`fat`	68,409	FAT file system
`disk`	13,330	Disk class driver
`classpnp`	27,032	Library for storage class drivers
`intelpro`	21,153	Intel PRO 100Mb/s Ethernet driver
`kmdf`	2,646	Benchmark: store/retrieve buffer
`ramdisk`	1,023	RAM disk driver
`serial`	19,466	Serial port driver
`rawether`	5,523	Ethernet packet capture driver
`topology`	7,560	Network topology discovery driver
`usbhub`	74,921	USB hub driver
`usbport`	77,367	USB host controller port driver
`usbehci`	22,133	USB EHCI miniport driver
`usbuhci`	8,730	USB UHCI miniport driver
`usbohci`	8,218	USB OHCI miniport driver
`usbwhci`	5,191	USB WHCI miniport driver

Table 5.1: Kernel extensions used to evaluate BGI.

5.5.1 Effectiveness

BGI's effectiveness at detecting faults before they propagate outside a domain was measured by injecting faults into the `fat` and `intelpro` drivers, following a methodology similar to the one described in [174].

Bugs from the five types in Table 5.2 were injected into the source code of the drivers. Empirical evidence indicates that these types of bugs are common in operating systems [145, 39]. The random increment to loop upper bounds and to the

number of bytes copied was 8 with 50% probability, 8 to 1K with 44% probability and 1K to 2K with 6% probability.

Fault type	Description
flip if condition	swap the "then" and "else" cases
lengthen loop	increase loop upper bounds
larger memcpy	increase number of bytes copied
off by one	replace ">" with ">=" or similar
delete assignment	remove an assignment statement

Table 5.2: Types of faults injected in extensions.

Buggy drivers were produced by choosing a bug type and injecting five bugs of that type in random locations in the driver source. After excluding the buggy drivers that failed to compile without BGI, 304 buggy `fat` drivers were tested by formatting a disk, copying two files onto it, checking the file contents, running the Postmark benchmark [83], and finally running `chkdsk`. Then 371 buggy `intelpro` drivers were tested by downloading a large file over `http` and checking its contents.

There were 173 `fat` tests and 256 `intelpro` tests that failed without BGI. There were different types of failure: blue screen inside driver, blue screen outside driver, operating system hang, test program hang, and test program failure. A blue screen is inside the driver if the faulting thread was executing driver code. Injected faults *escape* when the test ends with a blue screen outside the driver or an operating system hang. These are faults that affected the rest of the system. Injected faults are *internal* when the test ends with one of the other types of failure.

driver	type	contained	not contained
`fat`	blue screen	45 (100%)	0
	hang	3 (60%)	2
`intelpro`	blue screen	116 (98%)	2
	hang	14 (47%)	16

Table 5.3: Number of injected faults contained.

Buggy drivers with escaping faults were isolated in a separate BGI domain and tested again. Table 5.3 reports the number of faults that BGI was able to contain before they caused an operating system hang or a blue screen outside the driver. The faults that caused hangs were due to infinite loops and resource leaks. These faults affect the rest of the system not by corrupting memory but by consuming an excessive amount of CPU or other resources. Even though BGI does not contain explicit checks for this type of fault, it surprisingly contains 60% of the hangs in `fat` and 47% in `intelpro`. BGI is able to contain some hangs because it can detect some internal driver errors before they cause the hang; for example, it can prevent buffer overflows from overwriting a loop control variable. Containment could be improved by modifying extension wrappers to impose upper bounds on the time to execute driver functions, and by modifying the kernel wrappers that allocate and deallocate resources to impose an upper bound on resources allocated to the driver [133, 167].

BGI can contain more than 98% of the faults before they cause a blue screen outside the drivers. These faults corrupt state: they violate some assumption about the state in code outside the driver. These are the faults that BGI was designed to contain and they account for 90% of the escaping faults in `fat` and 80% in `intelpro`. BGI can contain a number of interesting faults; for example, three buggy drivers complete IRPs incorrectly. This bug is particularly hard to debug without isolation because it can corrupt an IRP that is now being used by the kernel or a different driver. BGI contains these faults and pinpoints the call to the `IoCompleteRequest` that causes the problem.

One of the two faults that cause a blue screen outside the `intelpro` driver is due to a bug in another driver in the stack that is triggered by the modified behaviour in this variant of `intelpro`.

driver	detected	not detected
`fat`	54 (44%)	68
`intelpro`	36 (33%)	72

Table 5.4: Number of internal faults detected by BGI.

BGI's ability to detect internal faults was also evaluated by isolating and testing buggy drivers with internal faults in a separate BGI domain. Table 5.4 shows the number of internal errors that BGI can detect before they are detected by checks in the test program or checks in the driver code. The results show that BGI can simplify debugging by detecting many internal errors early.

5.5.2 Performance

The overhead introduced by BGI was measured using benchmarks. For the disk, file system, and USB drivers, performance was measured using the PostMark [83] file-system benchmark that simulates the workload of an email server.

For `fat`, PostMark was configured to use cached I/O with 10,000 files and 1 million transactions. For the other drivers, it was configured for synchronous I/O with 100 files and 10,000 transactions. Caching was disabled for these drivers because, otherwise, most I/O is serviced from the cache, masking the BGI overhead.

The `disk` and `classpnp` drivers were run in the same protection domain because `classpnp` provides services to `disk`. This is common in Windows: a port driver implements functionality common to several miniport drivers to simplify their implementation. Similarly, `usbport` and `usbehci` were run in the same domain. The other drivers were tested separately.

Throughput is measured in transactions per second (Tx/s) as reported by PostMark, and kernel CPU time as reported by `kernrate`, the Windows kernel profiler [101]. Table 5.5 shows the percentage difference in kernel CPU time and throughput. BGI increases kernel CPU time by a maximum of 10% in these experiments. The throughput degradation is negligible in the 4 test cases that use synchronous I/O because the benchmark is I/O bound. For `fat`, the benchmark is CPU bound and throughput decreases by only 12%.

Driver	Δ CPU(%)	Δ Tx/s(%)
disk+classpnp	2.88	-1.37
ramdisk	0.00	0.00
fat	10.01	-12.31
usbport+usbehci	0.91	0.00
usbhub	4.20	0.00

Table 5.5: Overhead of BGI on disk, file system, and USB drivers, when running PostMark.

Driver	Benchmark	Δ CPU(%)	Δ bps(%)
xframe	TCP Send	11.94	-2.53
	TCP Recv	3.12	-3.65
	UDP Send	16.06	-10.26
	UDP Recv	6.86	-0.18
intelpro	TCP Send	13.57	0.00
	TCP Recv	11.69	0.00
	UDP Send	-1.74	-0.41
	UDP Recv	3.89	-0.84

Table 5.6: Overhead of BGI on network device drivers.

These results are significantly better than those reported for Nooks [150]; for example, Nooks increased kernel time by 185% in a FAT file-system benchmark. BGI performs better because it does not change page tables or copy objects in cross-domain calls.

The next results measure the overhead introduced by BGI to isolate the network card drivers. For the TCP tests, socket buffers of 256KB and 32KB messages were used with `intelpro` and socket buffers of 1MB and 64KB messages with `xframe`. IP and TCP checksum offloading was enabled with `xframe`. For the UDP tests, 16-byte packets were used with both drivers. Throughput was measured with the `ttcp` utility and kernel CPU time with `kernrate`. These tests are similar to those used to evaluate SafeDrive [174]. Table 5.6 shows the percentage difference in kernel CPU time and throughput due to BGI.

The results show that isolating the drivers with BGI has little impact on throughput. There is almost no throughput degradation with `intelpro` because this is a driver for a slow 100Mb/s Ethernet card. There is a maximum degradation of 10% with `xframe`. BGI reduces UDP throughput with `xframe` by less than SafeDrive [174]: 10% versus 11% for sends and 0.2% versus 17% for receives. But it reduces TCP throughput by more than SafeDrive: 2.5% versus 1.1% for sends and 3.7% versus 1.3% for receives. The SafeDrive results, however, were obtained with a Broadcom Tigon3 1Gb/s card while `xframe` is a driver for a faster 10Gb/s Neterion Xframe E.

The average CPU overhead introduced by BGI across the network benchmarks is 8% and the maximum is 16%. For comparison, Nooks [150] introduced a CPU overhead of 108% on TCP sends and 46% on TCP receives using similar benchmarks with 1Gb/s Ethernet. Nexus [167] introduced a CPU overhead of 137% when streaming a video

Buffer size	Δ Tx/s(%)		
	BGI	XFI fast path	XFI slow path
1	-8.76	-6.8	-5.9
512	-7.08	-5.3	-4.8
4K	-2.48	-2.7	-2.6
64K	-1.14	-1.4	-1.7

Table 5.7: Comparison of BGI's and XFI's overhead on the `kmdf` driver.

using an isolated 1Gb/s Ethernet driver. Nexus runs drivers in user space, which increases the cost of cross-domain switches. BGI's CPU overhead is similar to the CPU overhead reported for SafeDrive in the network benchmarks [174] but BGI provides stronger isolation.

Finally, the `kmdf` benchmark driver was used to compare the performance of BGI and XFI [159]. Table 5.7 shows the percentage change in transaction rate for different buffer sizes for BGI and for two XFI variants offering write protection. In each transaction, a user program stores and retrieves a buffer from the driver. BGI and XFI have similar performance in this benchmark but XFI cannot provide strong isolation guarantees for drivers that use the WDM, NDIS, or IFS interfaces (as discussed in Section 5.1 when examining the limitations of previous solutions). XFI was designed to isolate drivers that use the new KMDF interfaces. KMDF is a library that simplifies development of Windows drivers. KMDF drivers use the simpler interfaces provided by the library and the library uses WDM interfaces to communicate with the kernel. However, many KMDF drivers have code that uses WDM interfaces directly because KMDF does not implement all the functionality in the WDM interfaces. For example, the KMDF driver `serial` writes to some fields in IRPs directly and calls WDM functions that manipulate IRPs. Therefore, it is unclear whether XFI can provide strong isolation guarantees for real KMDF drivers.

5.6 Discussion

Faults in kernel extensions are the primary cause of unreliability in commodity operating systems. Previous fault-isolation techniques cannot be used with existing kernel extensions, or incur significant performance penalties. BGI's fine-grained isolation can support the existing Windows kernel extension API and the experimental results show that the overhead introduced by BGI is low enough to be used in practice to isolate Windows drivers in production systems, detecting internal kernel-errors before a system hang.

The Windows kernel API proved very amenable to fault isolation through API interposition, raising the question of whether such interposition can be successfully applied to other legacy operating systems with complicated interfaces. For this, a well-defined API with strict usage rules is key. On a practical note, support for Windows alone may have significant impact due to the popularity of this operating system. Nevertheless, it should be possible to use similar mechanisms for other operating systems, at least for some kernel extensions, and it will be worthwhile to

consider support for such interposition when designing future operating-system extension APIs.

Finally, it should be noted that BGI's trusted computing base includes the compiler as well as the interposition library, which may include bugs of their own. Future work could simplify the implementation of the interposition library by using API annotations to generate wrappers automatically.

Chapter 6

Related work

Mitigating the lack of memory safety in C programs is an old and important research problem that has generated a huge body of literature. Unfortunately, previous solutions have not been able to curb the problem in practice. This is because comprehensive solutions incur prohibitive performance overheads or break backwards compatibility at the source or binary level. Efficient and backwards-compatible solutions, on the other hand, have raised the bar, but remain vulnerable against advanced exploitation techniques.

6.1 Early solutions

Soon after buffer overflow attacks came on the scene, research focused on mitigating known exploitation techniques.

6.1.1 Call-stack integrity

Several techniques were inspired by StackGuard [48]. These techniques protect the return address from stack-based overflows using a "canary" placed next to the return address that is compared with its expected value before the function returns, or copy the return address to a safe area and use this copy when the function returns. These techniques have found their way into production use; for example, Microsoft Visual C provides a `/GS` option (turned on by default) and the GNU C compiler provides the `-fstack-protector` option.

These techniques have low overhead, but there are several attacks they cannot detect [166, 29, 127]. For example, targets other than the return address can be overwritten, including function pointers and `longjmp` buffers, or data pointers that can be used in a two-step attack. ProPolice [62] offered some additional protection by reordering stack variables to protect sensitive variables from overflows of adjacent arrays, and similar techniques are also used in production compilers. However, vulnerable buffers are not limited to the stack: heap-based and static buffers can overflow too.

6.1.2 Heap-metadata integrity

Heap canaries [128] have been used to detect heap-based overflows that tamper with in-band memory-management data structures. Heap memory allocators can also defend against such attacks by simply storing their metadata out-of-band. These solutions are efficient and have been used in production, but they are incomplete: memory errors on the heap may target data other than heap metadata, including function pointers in heap-allocated objects.

6.1.3 Static analysis

Static analysis [162, 91, 35, 10, 72] has been used to find memory errors in programs by analysing their source code. It is attractive because it does not introduce any runtime overhead. However, it is imprecise: it either misses many errors or raises many false alarms [176, 165]. Furthermore, the complexity of C makes static analysis hard: pointer arithmetic, equivalence of arrays and pointers, and alias analysis all require conservative assumptions further limiting precision.

Nevertheless, static analysis is important for practical solutions. When used in combination with dynamic techniques, such as those presented in this dissertation, false positives can be resolved at runtime and redundant checks can be eliminated at compile time.

6.2 Comprehensive solutions

The early solutions failed to mitigate the lack of memory safety because they offered limited protection. Several comprehensive solutions exist, but they typically either break backwards compatibility, or suffer from excessive overheads.

6.2.1 Clean slate

Using modern, memory-safe programming languages could address the problem. To ease the transition, languages like Java and C# were deliberately designed to resemble C in their syntax. While lots of software is now written in safe languages, C and C++ remain the most popular languages, mainly due to the abundance of legacy code and having an edge in performance. Porting requires significant human effort and safe languages lag in performance. This situation is unlikely to change in the near future.

6.2.2 Tweaking C

Rather than porting programs to a totally new language, programmers might find it easier to use a C dialect [81], a C subset amenable to analysis [60], or optional code annotations [107, 42]. Making annotations optional permits unmodified C programs to be accepted. Missing annotations can either fail safe at a performance cost [107], or fall back to unsafe execution [42]. Annotations may have to be trusted, introducing opportunities for errors, and may interact in complicated ways with the type system

(*e.g.* in C++). The approach of this dissertation, on the other hand, works on unmodified programs, does not require annotations, and the runtime mechanisms can be readily used with C++.

Cyclone [81, 71, 146] is a new dialect designed to minimise the changes required to port over C programs. But it still makes significant changes: C-style unions are replaced by ML-style sum types, and detailed pointer type declarations are required to support separate compilation. For common programs, about 10% of the program code must still be rewritten—an important barrier to adoption. Furthermore, Cyclone uses garbage collection, which can introduce unpredictable overheads. The performance problems of garbage collection were partially alleviated using region-based memory management [71], at the expense of further increasing the required porting effort. Cyclone's overhead is acceptable for I/O bound applications, but is considerable for computationally-intensive benchmarks (up to a factor of 3 according to [81]). An average slowdown by a factor of 1.4 is independently reported in [171].

Other approaches [88, 60] aim for memory safety without runtime checks or annotations for a subclass of type-safe C programs. They cannot, however, address large, general purpose programs, limiting their applicability to embedded and real-time control systems.

CCured [107, 43, 105] is a backwards-compatible dialect of C using static analysis with runtime checks for completeness. It uses type inference to conservatively separate pointers into several classes of decreasing safety, taking advantage of annotations when available. Most pointers are in fact found to be safe, and do not require bound checks. It uses fat pointers for the rest, and inserts any necessary runtime bounds checks. Unfortunately, fat pointers have backwards-compatibility problems. In follow-up work [43], metadata for some pointers is split into a separate data structure whose shape mirrors that of the original user data, enabling backwards compatibility with libraries, as long as changes to pointers in data do not invalidate metadata. CCured, like Cyclone, addresses temporal safety by changing the memory management model to garbage collection. Overhead is up to 150% [107] (26% average, 87% maximum, for the Olden benchmarks). Modifications to the source code are still required in some places, and unmodified programs may trigger false positives. With the Cyclone benchmarks minimally modified to eliminate compile-time errors and runtime false positives, CCured incurs an average slowdown of 4.7 according to [171].

Deputy [42] is a backwards-compatible dependent type system for C that lets programmers annotate bounded pointers and tagged unions. It has 20% average overhead for a number of annotated real-world C programs, but offers no temporal protection and, in its authors' tests, 3.4% of the lines of code had to be modified. A version of Deputy for device drivers is discussed in Section 6.4.

6.2.3 Legacy code

The sheer amount of legacy code means that even minor porting requirements can become a major obstacle. Some solutions avoid porting by accepting unmodified source code.

Early solutions had significant overhead. For example, BCC [85] was a source-to-source translator adding function calls to a run-time package. It increased each

pointer to three times its size to include upper and lower bounds. The BCC generated code was supposed to be about 30 times slower than normal, but the programs ran too slowly to be useful [141]. Rtcc [141] was a similar system adding run-time checking of array subscripts and pointer bounds to the Portable C Compiler (PCC), resulting in about ten-times slower program execution. Safe-C [11] was also based on fat pointers but could detect both spatial and temporal errors, with execution overheads ranging from 130% to 540%. Fail-Safe C [112] is a new, completely memory-safe compiler for the C language that is fully compatible with the ANSI C specification. It uses fat pointers, as well as fat integers to allow casts between pointers and integers. It reportedly slows down programs by a factor of 2–4, and it does not support programs using custom memory management.

These solutions have several problems caused by the use of fat pointers: assumptions about unions with integer and pointer members are often violated when using fat pointers, and memory accesses to fat pointers are not atomic. Programs relying on these assumptions, for example concurrent programs, may break unless modified. In some cases, in-band metadata like fat pointers can be compromised. Furthermore, memory overhead is proportional to the number of pointers, which can reach 200% in pointer intensive programs. The most important problem caused by fat pointers, however, is *binary incompatibility*: the resulting binaries cannot be linked with existing binary code such as libraries, because fat pointers change the pointer representation and the memory layout of data structures.

6.2.4 Binary compatibility

Patil and Fischer [118, 117] addressed the compatibility problems by storing metadata separately from the actual pointer, but their overhead was still high, hitting over 500% for the SPEC95 benchmarks. The bounds-checking solution by Jones and Kelly [82] specifically focused on backwards compatibility using a splay tree to map memory to its metadata. The generated code, however, is more than 10 times slower for pointer-intensive code. Moreover, it only supports programs with out-of-bounds pointers at most one element beyond the last element of an array. Memory overhead is proportional to the number of allocated objects, and can be high for programs that allocate many small objects (as demonstrated in Section 3.8).

CRED [130] extends the approach of Jones and Kelly [82] to allow the use of out-of-bounds addresses to compute in-bounds addresses. It achieves this using an auxiliary data structure to store information about out-of-bounds pointers. It also improves on performance at the cost of protection by limiting checking to string operations, which are the most vulnerable. It has less than 26% average overhead in applications without heavy string use, but 60–130% otherwise. The overhead for full checking, however, remains similar to [82]. CRED may fail if an out-of-bounds address is passed to an external library or if an out-of-bounds address is cast to an integer before subsequent use; these situations were found to be reasonably rare in practice. BBC has similar limitations, although WIT does not. Furthermore, tracking out-of-bounds pointers may cause leaks, since auxiliary data structure entries created for each out-of-bound pointer are only reclaimed when the intended referent object is deallocated, which may be too long (or never). Dealing with such cases may incur additional overhead

and weaken protection. The memory overhead of CRED was not studied, but should be similar to [82].

Xu *et al.* [170] describe a technique to improve backwards compatibility and reduce the overhead of previous techniques. Their solution provides spatial and temporal protection. It separates metadata from pointers using data structures that mirror those in the program. The metadata for each pointer `p` include bounds and a capability used to detect temporal errors, as well as a pointer to a structure that contains metadata for pointers that are stored within `*p`. This is one of the most comprehensive techniques, but its average runtime overhead when preventing only spatial errors is 63% for Olden and 97% for SPEC CINT—higher than BBC and WIT. Moreover, it has significant memory overhead for pointer intensive programs, up to 331% on average for the Olden benchmarks.

The system by Dhurjati *et al.* [57] partitions objects into pools at compile time and uses a splay tree for each pool. These splay trees can be looked up more efficiently than the single splay tree used by previous approaches, and each pool has a cache for even faster lookups. This technique has an average overhead of 12% and a maximum overhead of 69% for the Olden benchmarks. BBC's average overhead for the same benchmarks is 6% with a maximum of less than 20% (even if measured against the buddy system baseline), while WIT's average overhead for the same benchmarks is 4% with a maximum of 13%. Unfortunately, this system has not been evaluated for large CPU-bound programs beyond the Olden benchmarks and the memory overhead has not been studied. Interestingly, this technique can be combined with BBC easily.

SoftBound [104] is a recent proposal that records base and bound information for every pointer to a separate memory area, and updates these metadata when loading or storing pointer values. SoftBound's full-checking mode provides complete spatial protection with 67% average runtime overhead. To further reduce overheads, SoftBound has a store-only checking mode, similar to WIT, that incurs only 22% average runtime overhead. Similarly to the solutions presented in this dissertation, the key to the efficiency of this system is its very simple code sequence for runtime checks: a shift, a mask, an add, and two loads. This is still longer than either WIT (shift, load) or BBC (shift, load, shift). The big difference is that its simple code sequence comes at a high memory cost. Storing metadata per pointer can have memory overhead of up to 200% for the linear-table version (16 bytes per entry) and 300% for the hashtable version (24 bytes per entry). These theoretical worst-case overheads were encountered in practice for several pointer-intensive programs. SoftBounds deals with `memcpy` in a special way to ensure that metadata is propagated, but custom `memcpy`-like function will break metadata tracking. One advantage, however, is that it can protect sub-objects. Unfortunately, program-visible memory layout enables programs to legally navigate across sub-objects, causing false positives. Finally, as with some other solutions, library code modifying pointers in shared data structures will leave metadata in an inconsistent state.

SafeCode [59] maintains the soundness of the compile-time points-to graph, the call graph, and of available type information at runtime, despite memory errors. It requires no source-code changes, allows manual memory management, and does not use metadata on pointers or memory. The run-time overheads are less than 10% in nearly all cases and 30% in the worst case encountered, but it cannot prevent many

array bound-errors relevant in protecting programs (*e.g.* those accessing objects of the same type).

Similarly to WIT, DFI [32] combines static points-to analysis with runtime instrumentation to address non-control data attacks (these evade CFI, as discussed in Section 6.2.9). It computes a data-flow graph at compile time, and instruments the program to ensure that the flow of data at runtime follows the data-flow graph. To achieve this it maintains a table with the identifier of the last instruction to write to each memory location. The program is instrumented to update this table before every write with the identifier of the instruction performing the write, and to check before every read whether the identifier of the instruction that wrote the value being read matches one of those prescribed by the static data-flow graph. DFI can detect many out-of-bounds reads and reads-after-free, but it does not have guards to improve coverage when the analysis is imprecise and its average overhead for the SPEC benchmarks, where it overlaps with WIT, is 104%.

The technique described by Yong *et al.* [171] also has similarities with WIT: it assigns colours to objects and checks a colour table on writes. However, it has worse coverage than WIT because it uses only two colours, does not insert guards, and does not enforce control-flow integrity. The two colours distinguish between objects that can be written by an unsafe pointer and those that cannot. Yong *et al.* incur an average overhead ten times larger than WIT on the SPEC benchmarks where they overlap.

6.2.5 Working with binaries

Ideally, software should be protected without requiring access to the source code. Tools such as Purify [75], Annelid [108], and MEDS [78] can detect memory errors within binary executables. However, they slow down applications by a factor of 10 or more.

Preloaded shared libraries can be used to intercept and check vulnerable library functions, such as `strcpy`, in binary executables. For example, libsafe [14] uses the frame pointer to limit copy operations within the stack frame such that they do not overwrite the return address. It has less than 15% overhead, but offers limited protection.

The solutions examined in detail in Sections 6.1.1, 6.1.2, 6.2.8, and 6.2.9 can also usually be adapted to work with binaries, but they too suffer from either high costs or limited protection.

Working with binaries lets end users protect the programs they run, but is overlimiting: existing binary-only solutions have either low coverage or high overhead. Thus, it seems reasonable to require vendors and, in the case of open-source software, distributors, to apply the protections. In practice, applying intrusive changes to third-party software would also interfere with established software-support relationships.

6.2.6 Dangling pointers

Memory safety has both spatial and temporal dimensions, but many solutions described previously in this chapter offer only spatial protection. These can be used in combination with solutions that address temporal safety only.

Many debugging tools use a new virtual page for each allocation and rely on page-protecting freed objects to catch dangling pointer dereferences. Dhurjati and Adve [58] make this technique usable in practice by converting its memory overhead to address space overhead: separate virtual memory pages for each object are backed by the same physical memory. The run-time overhead for five Unix servers is less than 4%, and for other Unix utilities less than 15%. For allocation-intensive benchmarks, however, the time overhead increases up to a factor of 11.

HeapSafe [70] uses manual memory management with reference counting to detect dangling pointers. Its overhead over a number of CPU-bound benchmarks has a geometric mean of 11%, but requires source-code modifications and annotations, *e.g.* to deal with deallocation issues and `memcpy`-style functions that may transparently modify pointers.

Conservative garbage collection [24] can also be used to address temporal errors, but requires source-code tweaks, and has unpredictable overheads.

6.2.7 Format-string vulnerabilities

After the identification of the problem, several practical proposals have addressed format-string vulnerabilities. For example, FormatGuard [47] is a small patch to glibc that provides general protection against format bugs while imposing minimal compatibility and performance costs. Also, static analysis [136, 12, 126, 35] has been used successfully to weed out such bugs.

6.2.8 Taint checking

Dynamic taint tracking [50, 45, 36, 109, 46, 169, 121] is a powerful approach for detecting control-flow attacks resulting from a broad class of errors. It taints data derived from untrusted sources such as the network, by storing metadata to keep taint information about memory locations and registers, keeps track of taint propagation as data moves while the program executes, and raises an alarm if the program counter becomes tainted. Taint tracking is attractive because it can secure binary executables or even whole systems using binary rewriting, but can have enormous overheads, as high as 37×. More recent systems based on binary instrumentation have decreased overhead [124], but the average is still 260% for compute bound applications with a maximum slowdown by a factor of 7.9.

Higher performance has been achieved with compiler-based systems. [169] reduced the average overhead to 5% for server applications, but 61–106% (average 76%) for compute-bound applications. This approach uses a linear array as a tag map, similarly to my solutions, but the array occupies more address space: 1GB in a 32-bit machine. Some systems attempted to make performance tolerable using on-demand emulation [79, 120]. In addition, some more focused applications, such as detecting format-string attacks, can be made efficient [34] using static analysis to substantially reduce the number of checks.

Taint tracking has been extended to detect non-control-data attacks [36]. A pointer is tainted if the pointer value comes directly or indirectly from user input, and a security attack is detected whenever a tainted value is dereferenced during program

execution. This thwarts the attacker's ability to specify a malicious pointer value, which is crucial to the success of memory corruption attacks. The approach, however, has been criticised for producing excessive false positives [87, 138].

Clause *et al.* [41] describe a technique to detect illegal memory accesses using dynamic taint tracking with many colours. It assigns a random colour to memory objects when they are allocated and, when a pointer to an object is created, it assigns the colour of the object to the pointer. Then it propagates pointer colours on assignment and arithmetic. On reads and writes to memory, it checks if the colour of the pointer and the memory match. Their software-only version slows down SPEC CINT by a factor of 100 or more. With special hardware and 256 colours, their average overhead for SPEC CINT is 7%. This technique is similar to WIT, which has similar overhead without special hardware support.

6.2.9 Control-flow integrity

Enforcing control-flow integrity (CFI) [86, 2, 1] can prevent attacks from arbitrarily controlling program behaviour. The concept of CFI generalises the work of Wagner and Dean [161] that constrained the system-call trace of a program's execution to be consistent with its source code. CFI dictates that software execution must follow a path of a control-flow graph (CFG) determined ahead of time. The CFG in question can be determined by analysis—source-code analysis, binary analysis, or execution profiling. CFI can be efficiently implemented in software using binary rewriting.

Unfortunately, attackers can exploit memory errors to execute arbitrary code without violating CFI. Several attack examples are discussed in [37]. Nevertheless, CFI is a useful and cheap-to-enforce safety property. WIT enforces CFI as a second line of defence and to guarantee that its checks cannot be bypassed. The original CFI system shows an average overhead of 15% and a maximum overhead of 45% on the SPEC benchmarks that overlap with WIT. The average overhead grows to 24% with the version of CFI that uses a shadow call-stack to ensure that functions return to their caller. WIT has an average overhead of 10% and a maximum of 25% and it also ensures functions return to their caller. WIT has lower overhead because it operates on source code, and because it avoids CFI checks on function returns.

6.2.10 Privilege separation

Privilege separation [122] is a programming technique that isolates applications into fine-grained least-privilege compartments. Decomposition has been automated to a certain extent [22] using tools to help identify the sharing needs of compartments. It can address integrity as well as confidentiality, but requires more programmer effort and may incur significant overhead for frequently communicating compartments due to hardware paging (although using mechanisms such as those of BGI's fault domains may help alleviate this).

6.3 Boosting performance

Excessive performance degradation is widely recognised as a major limitation of many techniques, and several approaches have aimed to address it. Some of them are orthogonal to the work of this dissertation, and could potentially improve its performance, while others have limitations beyond performance.

6.3.1 Probabilistic defences

Various randomisation and obfuscation proposals perturb the execution environment to make attacks fail with high probability. Probabilistic protection can be cheaper to enforce than deterministic.

Address-space-layout randomisation (ASLR) [151] defends against memory error exploits by randomly distributing the positions of key data areas, including the base of the executable and the position of libraries, heap, and stack, in the process address-space. Simple address-space-layout randomisation is cheap; however, an attacker may succeed after multiple retries, or against a subset of the vulnerable population in the case of a large scale attack, and entropy is very limited in 32-bit systems [135]. Moreover, comprehensive techniques that randomise the position of every individual object have higher overhead (*e.g.* 17% for `gzip` [21] compared to WIT's 7%).

DieHard [18] randomises the size of heap allocations and the order in which they are reused, allowing programs to survive some memory errors with high probability. The resulting memory increase, however, can be impractical. Moreover, while DieHard's runtime overhead for SPEC CINT is 12% on average, the maximum overhead is 109% for the `twolf` benchmark, while WIT's overhead for `twolf` is only 3%. Archipelago [95] improves on DieHard's memory usage by trading address space; however, it still has high overhead for allocation-intensive workloads. In addition, like BBC, DieHard provides deterministic protection against buffer overflows that cross allocation boundaries, but, unlike BBC, its protection is limited to heap allocations and standard library functions like `strcpy`.

Another proposed technique is *instruction-set randomisation* (ISR) [84, 17, 16]. It uses process-specific randomised instruction sets so that attackers without knowledge of the key to the randomisation algorithm would inject code that is invalid for the given randomised processor, causing a runtime exception. Software ISR has higher runtime overhead than other randomisation approaches, but could benefit from hardware support. However, it does not prevent non-control data attacks. Moreover, a motivated attacker may be able to circumvent ISR by determining the randomisation key, and packaging his code to reduce the number of key bytes that must be determined [139, 164].

A third form of randomisation—*data space randomisation* (DSR) [20, 30]— randomises the representation of data stored in program memory. Unlike ISR, DSR is effective against non-control data attacks as well as code-injection attacks. Unlike ASLR, it can protect against corruption of non-pointer data as well as pointer-valued data. DSR has an average overhead of 15% (5–30%) for a mix of I/O-bound and CPU-bound programs. WIT has lower overhead for purely CPU-bound programs.

Moreover, sharing data with third-party libraries is challenging: randomisation for any data potentially accessed by third-party code has to be conservatively disabled.

Many probabilistic countermeasures rely on secret values which a lucky or determined attacker can guess. A critical assumption in many probabilistic countermeasures is that attackers cannot read the contents of memory. However, buffer overreads [142, 158] or reading contents of uninitialised buffers containing pointer values [20] can be exploited to read parts of the memory contents of a process running a vulnerable application. Thus, an application protected by ASLR, ISR, or canaries can be exploited if it contains both a buffer overread and a buffer overflow vulnerability.

6.3.2 Parallel checking

As multiprocessors are becoming increasingly common, various proposals [117, 118, 110] have sought to take advantage of spare parallelism by decoupling pointer- and array-access checking from the original computation. The basic idea is to obtain a reduced version of the user program by deleting computations not contributing to pointer and array accesses, and then inserting checks to this reduced version and running it in parallel, but in advance of the ordinary application. The main process only performs computations in the original program, occasionally communicating a few key values to the shadow process, which follows the main process checking pointer and array accesses. The overhead observed by the main process is very low—almost always less than 10% [117, 118]. This technique is orthogonal to my work, and is also not sufficient on its own because the burden on the shadow process remains significant, while tracking can become complicated due to concurrency and limitations on speculation.

Multi-variant execution [49] is another approach that can exploit abundant hardware resources. It works by executing two or more variants of the program in parallel to detect any deviation in their observable execution. An example of this approach is Orchestra [131], which runs two program variants differing in the direction their stack grows. Orchestra in particular provides limited protection, but in principle the approach can be extended with more variation among program versions. DieHard [18], for example, has a replicated mode where two copies of the program are executed using different heap randomisation. In general, non-determinism that can cause replicas to diverge (*e.g.* due to shared memory races) is hard to avoid efficiently [27], and the increased memory and energy use remains a problem.

6.3.3 Hardware support

AccMon [175] is a hardware proposal that is similar in spirit to WIT. It is based on the observation that, in most programs, a given memory location is typically accessed by only a few instructions. Therefore, by capturing the invariant set of program addresses that normally access a given variable, one can detect accesses by outlier instructions, which are often caused by memory errors. Its use of dynamic analysis, however, leads to false positives.

SafeMem [123] makes novel use of existing ECC memory technology to insert guards around objects with reasonable overhead (1.6%-14.4%). The necessary hardware, however, is not readily available on commodity systems.

HardBound [56] is a hardware-based bounds checking system. Moreover, many proposals described earlier, *e.g.* [50, 84, 41] can use hardware support to improve their performance. No comprehensive solutions, however, have been implemented in commodity hardware to date.

The only hardware-based solution that is seeing wide adoption is data execution prevention (DEP) [98] through the use of the NX bit in page table entries. DEP can efficiently prevent attacker code smuggled into the program as data. It does not protect, however, against attacks executing code already present in the program [53, 119, 134, 28, 89] and non-control-data attacks [37]. Moreover, the combination of DEP and ASLR is not immune to attacks either [129].

6.4 Kernel-level solutions

Many techniques to prevent or contain memory errors in kernel extensions have been proposed. These techniques, however, are not widely used because they cannot isolate existing kernel extensions with low overhead on standard hardware.

Some systems support kernel extensions written in a safe language [19, 25, 80]. These solutions, however, incur performance overheads associated with safe languages and cannot be applied to existing drivers.

Another approach is proof-carrying code [106] which allows running efficient code without runtime checks. A theorem prover is run offline to construct a proof that the code follows a security specification, and the kernel can rapidly verify this proof. Any code can be run, as long as a proof can be provided, but this is hard to achieve for complex extensions. Complex proofs require human intervention, and new effort is required if the code is updated.

Some systems isolate extensions by running them in user space (*e.g.* [66, 74, 77, 93]) but this introduces high overhead. In fact, both Linux and Windows support user-mode device drivers [40, 100] but only some types of devices are supported and there is a significant performance penalty. Microdrivers [69] improve performance by splitting a driver into a kernel component and a user component with help from programmer annotations. As much as 65% of driver code can be removed from the kernel, but the remaining kernel component is not isolated. Nexus [167] is a new operating system with comprehensive support for user-level drivers; however, it is not backwards-compatible with existing drivers and its performance overhead is high. Extensions can also be isolated on virtual machines (*e.g.* Xen [15]) by running a dedicated operating system with the extension in a separate virtual machine (*e.g.* [94, 144, 68]), but this approach also incurs high overhead.

Other systems isolate extensions while running them in the kernel address space. Nooks [149, 147, 148] was the first system to provide isolation of existing extensions for a commodity operating system, but it has high overhead because it uses hardware page protection. Mondrix [168] uses fine-grained hardware memory protection to isolate kernel extensions with low overhead, but requires special hardware, does not

check for incorrect uses of the extension interface, and it only supports 4-byte granularity (which is insufficient without major changes to commodity operating systems).

Software-based fault-isolation techniques, like SFI [163], PittSFIeld [97] and XFI [159], can isolate kernel extensions with low overhead but they do not deal with the complex extension interfaces of commodity operating systems (as discussed in Section 5.1). Other software-based fault-isolation techniques have similar problems.

SafeDrive [174] implements a bounds-checking mechanism for C that requires programmers to annotate extension source code. It has low overhead but it provides weak isolation guarantees. It does not provide protection from temporal errors, and, while it can prevent out-of-bounds reads, it does not distinguish between read and write accesses; for example, SafeDrive would allow the driver to write to fields in the IRP that can be legally read but not modified by extensions. Furthermore, it requires programmers to annotate the extension source code. The Ivy project aims to combine SafeDrive with Shoal [9] and HeapSafe [70], which should provide improved isolation when completed. However, this combination would still lack BGI's dynamic typestate checking, would require annotations, and it would be likely to perform worse than BGI because both Shoal and HeapSafe can introduce a large overhead.

Finally, SVA [51] provides a safe execution environment for an entire operating system, and, similarly to BGI, it can enforce some safety properties to prevent many memory errors. It does not isolate extensions, however, and incurs higher overhead than BGI.

6.5 Beyond memory-safety errors

More comprehensive and higher-level security guarantees can be enforced with the use of end-to-end security policies, albeit at the cost of increased programmer or user involvement. For example, decentralised information-flow control (DIFC) [103] can be used to enforce security policies on the flow of information between application components and the outside world. DIFC can protect against higher-level programmer errors, as well as against adversarial, untrusted code, and is applicable to legacy programs through implementations at the operating system level [61, 90, 173]. The programmer, however, must set up a security policy, and may have to restructure the program accordingly. My solutions, on the other hand, rely on implicit policies such as programming language rules forbidding the use of pointers obtained from the address of one memory allocation to access another memory allocation, or kernel API usage rules. Moreover, the protection granularity of these operating-system solutions is coarser compared to the language-level protection, permitting attacks conforming to the information-flow policy. For example, a compromised web server may be prevented from accessing the file system or other processes, but compromised code may still corrupt data, such as the web server responses it generates.

Finally, there are many important types of common vulnerabilities beyond memory-safety errors, such as cross-site scripting and SQL-injection vulnerabilities in web applications [154], to name a few, that also plague memory-safe languages. Unlike low-level memory errors targeted in this work, performance is less of a challenge when addressing such higher-level vulnerabilities.

Chapter 7

Conclusions

7.1 Summary and lessons learnt

The goal of this work was to mitigate the memory safety problems of C and C++ with solutions that can be used in the real world today. I observed that previous comprehensive solutions have failed on two grounds: lack of performance or backwards compatibility.

This work tried to explore new ways to address performance and backwards compatibility. BBC showed that dangerous memory-errors can be prevented at a lower cost, if memory errors that cannot be exploited by attackers are tolerated. WIT showed that a better balance between protection and performance is possible. Initial versions of WIT used fast runtime-checking mechanisms to detect simple sequential overflows, but I found it possible to increase protection with static analysis and checks on control-flow transfers without compromising performance. BGI further extended WIT with support for isolation of kernel extensions in separate protection domains to safeguard the availability of the main kernel, while maintaining good performance and backwards compatibility with the legacy Windows kernel API.

Table 7.1 compares the three solutions. Between the user-space solutions, BBC offers more complete spatial memory-safety protection by validating both reads and writes, but WIT is more efficient and consistent in performance, and offers control-flow integrity and indirect protection against temporal memory-safety violations and hijacked structure fields. BGI, in addition to the language-level rules of the user-space solutions, takes advantage of the kernel-API rules to enforce additional protection.

System	Address Space	Enforced Boundary	Valid Reads	Valid Writes	CFI	Temporal Protection	Overhead
BBC	user	allocation	yes	yes	no	no	30%
WIT	user	allocation	no	yes	yes	yes	7%
BGI	kernel	allocation, kernel extension	no	yes	yes	yes	7%

Table 7.1: Summary of the three proposals. All solutions are backwards compatible, but protection and performance vary.

Unlike many previous solutions, I focused more on making runtime checks efficient, rather than reducing their frequency through static analysis. Approaches relying on static analysis to eliminate redundant checks may have significant variation in performance, depending on the precision of the analysis. Lightweight checks, on the other hand, provide consistently good performance. This is especially true for WIT, that does not have BBC's slow path for handling out-of-bounds pointers.

At the implementation level, a simple but efficient data structure is used at the core of all solutions: a linear table mapping memory ranges to their metadata, which can be object sizes, colours, or rights. A key technique used in all solutions to improve performance without breaking binary compatibility was to modify the layout of objects in memory, for example, by padding objects to powers of two or organising memory in slots.

The three systems were implemented using the Microsoft Phoenix compiler framework [99], but it should be possible to re-implement them using the GNU GCC or LLVM [92] frameworks. Preliminary experiments with partial implementations of BBC and WIT using LLVM showed performance comparable to the Phoenix-based implementations.

7.2 Future directions

Several additional compile-time optimisations are possible. Since my work is orthogonal to existing approaches minimising the number of runtime checks through sophisticated program analysis, integrating state-of-the-art optimisations should further reduce the average runtime overhead. Moreover, there are some optimisations specific to my work that I have not yet explored. For example, WIT could unroll some loops containing overlapping write-integrity checks to make loop iterations access a single memory slot instead of 8 (assuming 8-byte memory slots).

Another practical improvement is implementing the described code transformations using binary rewriting. In this work, I instrumented code during code generation, thus relying on a specific compiler and requiring recompilation of existing software. This facilitated program analysis and adjusting memory layouts. It seems feasible, however, to implement much of this work using binary rewriting, especially if debugging information is available. This would enable the use of different compilers and remove the need for recompilation and source-code availability.

Finally, while I chose to materialise the ideas in this work using C, aiming at immediate usability and impact, it may be possible to apply these techniques to other languages or even hardware-based solutions. Memory safety incurs fundamental overheads that are also undesirable in other languages. Moreover, a hardware implementation could efficiently generalise the concept of baggy bounds beyond powers of two. For example, using a compact floating-point representation for bounds, the worst-case space overhead due to padding could be made less than 25% by using two extra bits in the bounds representation to represent allocation sizes with better than power-of-two precision. The bitwise operations required to align and check such pointers, while prohibitively expensive in software, may afford an efficient hardware implementation.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2005. Cited on pages 20, 27, 61, and 114.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *Proceedings of the 7th IEEE International Conference on Formal Engineering Methods (ICFEM)*, 2005. Cited on pages 20, 27, 61, and 114.

[3] Jonathan Afek and Adi Sharabani. Dangling pointer: Smashing the pointer for fun and profit. `http://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf`, 2007. Cited on page 24.

[4] Dave Ahmad. The rising threat of vulnerabilities due to integer errors. *IEEE Security and Privacy Magazine*, 1(4), July 2003. Cited on pages 23 and 24.

[5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2008. Cited on pages 19 and 45.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, 2009. Cited on page 19.

[7] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994. Cited on pages 20, 21, and 65.

[8] James P. Anderson. Computer security technology planning study. `http://csrc.nist.gov/publications/history/ande72.pdf`, 1972. Cited on page 23.

[9] Zachary R. Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009. Cited on page 118.

[10] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002. Cited on page 108.

[11] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1994. Cited on pages 27, 29, 58, and 110.

[12] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. ACM, 2005. Cited on pages 65, 72, and 113.

[13] Michael Bailey, Evan Cooke, Farnam Jahanian, David Watson, and Jose Nazario. The Blaster worm: Then and now. *IEEE Security and Privacy Magazine*, 3(4), 2005. Cited on page 23.

[14] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2000. Cited on page 112.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*. ACM, 2003. Cited on page 117.

[16] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1), 2005. Cited on page 115.

[17] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2003. Cited on page 115.

[18] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2006. Cited on pages 115 and 116.

[19] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*. ACM, 1995. Cited on page 117.

[20] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th international Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer-Verlag, 2008. Cited on pages 115 and 116.

[21] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 2005. Cited on page 115.

[22] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2008. Cited on page 114.

[23] blexim. Basic integer overflows. *Phrack*, 11(60), 2002. `http://phrack.com/issues.html?issue=60&id=11`. Cited on pages 23 and 24.

[24] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9), 1988. Cited on pages 27, 58, and 113.

[25] Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *Proceedings of the 5th IEEE Conference on Open Architectures and Network Programming (OPENARCH)*. IEEE Computer Society, 2002. Cited on page 117.

[26] Thomas Boutell and Tom Lane. Portable network graphics (PNG) specification and extensions. `http://www.libpng.org/pub/png/spec/`. Cited on pages 51 and 80.

[27] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicæ for defeating memory error exploits. In *3rd International Workshop on Information Assurance (WIA)*. IEEE Computer Society, 2007. Cited on page 116.

[28] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2008. Cited on pages 27 and 117.

[29] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10(56), 2000. `http://phrack.com/issues.html?issue=56&id=10`. Cited on page 107.

[30] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical Report TR-2008-120, Microsoft Research, 2008. Cited on page 115.

[31] Martin C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, 1996. Cited on pages 44 and 74.

[32] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006. Cited on pages 21, 28, 50, 65, and 112.

[33] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating System Principles (SOSP)*. ACM, 2009. Cited on pages 19 and 92.

[34] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2008. Cited on page 113.

[35] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities in Debian Linux. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 2007. Cited on pages 72, 108, and 113.

[36] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2005. Cited on page 113.

[37] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 2005. Cited on pages 23, 26, 30, 31, 79, 114, and 117.

[38] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 2005. Cited on page 41.

[39] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society, 1996. Cited on page 100.

[40] Peter Chubb. Get more device drivers out of the kernel! In *Proceedings of the 2002 Ottawa Linux Symposium (OLS)*, 2004. Cited on pages 16 and 117.

[41] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective memory protection using dynamic tainting. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2007. Cited on pages 114 and 117.

[42] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming (ESOP)*, 2007. Cited on pages 108 and 109.

[43] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2003. Cited on pages 16 and 109.

[44] Matt Conover and w00w00 Security Team. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, 1999. Cited on page 23.

[45] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain Internet worms? In *Proceedings of the 3rd ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, 2004. Cited on page 113.

[46] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP)*. ACM, 2005. Cited on page 113.

[47] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*. USENIX Association, 2001. Cited on pages 72 and 113.

[48] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, 1998. Cited on pages 17, 27, and 107.

[49] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*. USENIX Association, 2006. Cited on page 116.

[50] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2004. Cited on pages 113 and 117.

[51] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating System Principles (SOSP)*. ACM, 2007. Cited on pages 42, 84, and 118.

[52] DedaSys LLC. Programming language popularity. `http://www.langpop.com`, 2010. Cited on pages 15 and 23.

[53] Solar Designer. "return-to-libc" attack. Bugtraq security mailing list, 1997. Cited on pages 23 and 117.

[54] Solar Designer. JPEG COM marker processing vulnerability in Netscape browsers. `http://www.openwall.com/advisories/OW-002-netscape-jpeg`, 2000. Cited on page 23.

[55] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice and Experience*, 24(6), 1994. Cited on page 58.

[56] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2008. Cited on page 117.

[57] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 2006. Cited on pages 21, 33, 36, 38, 42, 44, 45, 46, 48, 50, 72, 75, and 111.

[58] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2006. Cited on page 113.

[59] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Notices*, 41(6), 2006. Cited on pages 27, 58, and 111.

[60] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003. Cited on pages 27, 58, 108, and 109.

[61] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP)*. ACM, 2005. Cited on page 118.

[62] Hiroaki Etoh and Kunikazu Yoda. ProPolice. In *IPSJ SIGNotes Computer Security (CSEC)*, volume 14, 2001. Cited on page 107.

[63] Security Focus. Ghttpd Log() function buffer overflow vulnerability. `http://www.securityfocus.com/bid/5960`. Cited on page 79.

[64] Security Focus. Null HTTPd remote heap overflow vulnerability. `http://www.securityfocus.com/bid/5774`. Cited on page 79.

[65] Security Focus. STunnel client negotiation protocol format string vulnerability. `http://www.securityfocus.com/bid/3748`. Cited on page 79.

[66] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *Proceedings of the USENIX Mach Symposium (MACHNIX)*. USENIX Association, 1991. Cited on page 117.

[67] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*. IEEE Computer Society, 1997. Cited on page 28.

[68] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS-1)*, 2004. Cited on page 117.

[69] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. *ACM SIGOPS Operating Systems Review*, 42(2), 2008. Cited on page 117.

[70] David Gay, Rob Ennals, and Eric Brewer. Safe manual memory management. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)*. ACM, 2007. Cited on pages 27, 58, 113, and 118.

[71] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2002. Cited on pages 27, 58, and 109.

[72] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Static Analysis Symposium (SAS)*. Springer-Verlag, 2003. Cited on page 108.

[73] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 2006. Cited on page 92.

[74] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*. ACM, 1997. Cited on page 117.

[75] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, pages 125–138, San Francisco, California, 1992. USENIX Association. Cited on page 112.

[76] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. *SIGPLAN Notices*, 36(5), 2001. Cited on pages 21, 61, and 62.

[77] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3), 2006. Cited on page 117.

[78] Jason D. Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson. MEDS: The memory error detection system. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems (ESSoS)*. Springer-Verlag, 2009. Cited on page 112.

[79] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. ACM, 2006. Cited on page 113.

[80] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2), 2007. Cited on page 117.

[81] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2002. Cited on pages 108 and 109.

[82] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, 1997. Cited on pages 22, 29, 30, 33, 34, 36, 38, 44, 45, 48, 50, 72, 110, and 111.

[83] Jeffrey Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance Inc., 1997. Cited on pages 101 and 102.

[84] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2003. Cited on pages 115 and 117.

[85] Samuel C. Kendall. BCC: Runtime checking for C programs. In *Proceedings of the USENIX Summer Technical Conference*. USENIX Association, 1983. Cited on page 109.

[86] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, 2002. Cited on pages 20, 27, and 114.

[87] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*. ACM, 2006. Cited on page 114.

[88] Sumant Kowshik, Dinakar Dhurjati, and Vikram 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)*. ACM, 2002. Cited on page 109.

[89] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. `www.suse.de/~krahmer/no-nx.pdf`, 2005. Cited on page 117.

[90] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of the 21st ACM Symposium on Operating System Principles (SOSP)*. ACM, 2007. Cited on page 118.

[91] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*. USENIX Association, 2001. Cited on page 108.

[92] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO)*. IEEE Computer Society, 2004. Cited on page 120.

[93] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yue-Ting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology (JCS&T)*, 20(5), 2005. Cited on page 117.

[94] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2004. Cited on page 117.

[95] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2008. Cited on page 115.

[96] Jean-Philippe Martin, Michael Hicks, Manuel Costa, Periklis Akritidis, and Miguel Castro. Dynamically checking ownership policies in concurrent C/C++ programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL)*. ACM, 2010. Cited on page 21.

[97] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*. USENIX Association, 2006. Cited on pages 16, 84, and 118.

[98] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. `http://support.microsoft.com/kb/875352/EN-US/`. Cited on pages 16 and 117.

[99] Microsoft. Phoenix compiler framework. `http://connect.microsoft.com/Phoenix`. Cited on pages 21, 65, 74, 100, and 120.

[100] Microsoft. User-mode driver framework (UMDF). `www.microsoft.com/whdc/driver/wdf/UMDF.mspx`. Cited on pages 16 and 117.

[101] Microsoft. Windows driver kit (WDK). `http://www.microsoft.com/wdk`. Cited on pages 92, 100, and 102.

[102] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy Magazine*, 1(4), 2003. Cited on pages 23 and 79.

[103] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9, 2000. Cited on page 118.

[104] Santosh Nagarakatte, Jianzhou Zhao, Milo Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009. Cited on pages 16, 37, 44, 50, 72, and 111.

[105] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), 2005. Cited on pages 16, 75, 76, and 109.

[106] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. *ACM SIGOPS Operating Systems Review*, 30(SI), 1996. Cited on page 117.

[107] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL)*. ACM, 2002. Cited on pages 16, 108, and 109.

[108] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2004. Cited on page 112.

[109] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2005. Cited on pages 28 and 113.

[110] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2008. Cited on page 116.

[111] NullLogic. Null HTTPd. `http://nullwebmail.sourceforge.net/httpd`. Cited on pages 30, 50, and 78.

[112] Yutaka Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009. Cited on page 110.

[113] Elias Levy (Aleph One). Smashing the stack for fun and profit. *Phrack*, 7(49), 1996. `http://phrack.com/issues.html?issue=49&id=14`. Cited on page 23.

[114] Walter Oney. Programming the Microsoft Windows Driver Model. Microsoft Press, second edition, 2002. Cited on pages 22 and 83.

[115] OpenSSL Toolkit. `http://www.openssl.org`. Cited on page 50.

[116] Hilarie Orman. The Morris worm: A fifteen-year perspective. *IEEE Security and Privacy Magazine*, 1(5), 2003. Cited on page 23.

[117] Harish Patil and Charles N. Fischer. Efficient run-time monitoring using shadow processing. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, 1995. Cited on pages 110 and 116.

[118] Harish Patil and Charles N. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software—Practice and Experience*, 27(1), 1997. Cited on pages 110 and 116.

[119] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy Magazine*, 2(4), 2004. Cited on pages 23 and 117.

[120] Georgios Portokalidis and Herbert Bos. Eudaemon: Involuntary and on-demand emulation against zero-day exploits. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. ACM, 2008. Cited on page 113.

[121] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4), 2006. Cited on page 113.

[122] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 2003. Cited on page 114.

[123] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2005. Cited on page 117.

[124] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006. Cited on page 113.

[125] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th USENIX Security Symposium*, 2009. Cited on page 26.

[126] Michael F. Ringenburg and Dan Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2005. Cited on pages 72 and 113.

[127] Rix. Smashing C++ vptrs. *Phrack*, 10(56), 2000. `http://phrack.com/issues.html?issue=56&id=10`. Cited on page 107.

[128] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX Large Installation Systems Administration Conference (LISA)*. USENIX Association, 2003. Cited on pages 27 and 108.

[129] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 2009. Cited on page 117.

[130] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2004. Cited on pages 16, 32, 33, 34, 36, 37, 38, 44, 45, 48, 50, 72, and 110.

[131] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. ACM, 2009. Cited on page 116.

[132] SecurityFocus. Vulnerabilities. `http://www.securityfocus.com/vulnerabilities/`. Cited on page 16.

[133] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996. Cited on page 101.

[134] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2007. Cited on pages 27 and 117.

[135] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2004. Cited on pages 28 and 115.

[136] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*. USENIX Association, 2001. Cited on pages 72 and 113.

[137] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. `http://skypher.com/wiki/index.php/Www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php`, 2004. Cited on page 23.

[138] Asia Slowinska and Herbert Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*. ACM, 2009. Cited on page 114.

[139] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, 2005. Cited on page 115.

[140] Manu Sridharan and Stephen J. Fink. The complexity of Andersen's analysis in practice. In *Static Analysis Symposium/Workshop on Static Analysis*, 2009. Cited on page 65.

[141] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software—Practice and Experience*, 22(4), 1992. Cited on page 110.

[142] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security (EuroSec)*. ACM, 2009. Cited on pages 28 and 116.

[143] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)*, 12(1), 1986. Cited on page 90.

[144] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2001. Cited on page 117.

[145] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability—a study of field failures in operating systems. In *Proceedings of the 21st Annual International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society, 1991. Cited on page 100.

[146] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming (SCP)*, 62(2), 2006. Cited on page 109.

[147] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2004. Cited on pages 84, 85, and 117.

[148] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4), 2006. Cited on page 117.

[149] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*. ACM, 2003. Cited on pages 84, 85, and 117.

[150] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1), 2005. Cited on page 103.

[151] PaX Team. Address space layout randomization. `http://pax.grsecurity.net/docs/aslr.txt`. Cited on pages 16 and 115.

[152] The Apache Software Foundation. The Apache HTTP Server Project. `http://httpd.apache.org`. Cited on page 50.

[153] The MITRE Corporation. Common vulnerabilities and exposures. `http://cve.mitre.org/data/downloads/`. Cited on page 16.

[154] The MITRE Corporation. Common weakness enumeration. `http://cwe.mitre.org/`. Cited on pages 24 and 118.

[155] The MITRE Corporation. Multiple buffer overflows in libpng 1.2.5. CVE-2004-0597, 2004. Cited on page 80.

[156] The Standard Performance Evaluation Corporation. SPEC benchmarks. `http://www.spec.org`. Cited on pages 44, 74, 78, and 80.

[157] TIOBE Software BV. Programming community index. `http://www.tiobe.com`, 2010. Cited on pages 15 and 23.

[158] Nathan Tuck, Brad Calder, and George Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2004. Cited on page 116.

[159] Úlfar Erlingsson, Silicon Valley, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006. Cited on pages 16, 84, 85, 104, and 118.

[160] US-CERT. Vulnerability notes. `http://www.kb.cert.org/vuls/`. Cited on page 16.

[161] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2001. Cited on page 114.

[162] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2000. Cited on page 108.

[163] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*. ACM, 1993. Cited on pages 16, 84, and 118.

[164] Yoav Weiss and Elena Gabriela Barrantes. Known/chosen key attacks against software instruction set randomization. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 2006. Cited on page 115.

[165] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NordSec)*, 2002. Cited on page 108.

[166] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2003. Cited on pages 50, 78, and 107.

[167] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2008. Cited on pages 101, 103, and 117.

[168] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP)*. ACM, 2005. Cited on page 117.

[169] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*. USENIX Association, 2006. Cited on page 113.

[170] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *ACM SIGSOFT Software Engineering Notes (SEN)*, 29(6), 2004. Cited on pages 16, 27, 44, 45, 48, 50, 58, 72, 76, and 111.

[171] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. *ACM SIGSOFT Software Engineering Notes (SEN)*, 28(5), 2003. Cited on pages 109 and 112.

[172] Suan Hsi Yong and Susan Horwitz. Pointer-range analysis. In *Proceedings of the 11th International Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004. Cited on page 38.

[173] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006. Cited on page 118.

[174] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the*

7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, 2006. Cited on pages 84, 92, 100, 103, 104, and 118.

[175] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2004. Cited on page 116.

[176] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2004. Cited on page 108.

[177] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7), 1993. Cited on page 58.

[178] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code Red worm propagation modeling and analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2002. Cited on page 23.