

# Inline and Sideline Approaches for Low-cost Memory Safety in C

**Myoung Jin Nam**

Selwyn College

November 2020

*This dissertation is submitted  
for the degree of  
Doctor of Philosophy*



**UNIVERSITY OF  
CAMBRIDGE**

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. This thesis does not exceed the regulation length of sixty thousand words.

# Inline and Sideline Approaches for Low-cost Memory Safety in C

Myoung Jin Nam

System languages such as C or C++ are widely used for their high performance, however the allowance of arbitrary pointer arithmetic and type-cast introduces a risk of memory corruptions. These memory errors cause unexpected termination of programs, or even worse, attackers can exploit them to alter the behavior of programs or leak crucial data.

Despite advances in memory safety solutions, high and unpredictable overhead remains a major challenge. Accepting that it is extremely difficult to achieve complete memory safety with the performance level suitable for production deployment, researchers attempt to strike a balance between performance, detection coverage, interoperability, precision, and detection timing. Some properties are much more desirable, e.g. the interoperability with pre-compiled libraries. Comparatively less critical properties are sacrificed for performance, for example, tolerating longer detection delay or narrowing down detection coverage by performing approximate or probabilistic checking or detecting only certain errors. Modern solutions compete for performance.

The performance matrix of memory safety solutions have two major assessment criteria – run-time and memory overheads. Researchers trade-off and balance performance metrics depending on its purpose or placement. Many of them tolerate the increase in memory use for better speed, since memory safety enforcement is more desirable for troubleshooting or testing during development, where a memory resource is not the main issue. Run-time overhead, considered more critical, is impacted by cache misses, dynamic instructions, DRAM row activations, branch predictions and other factors.

This research proposes, implements, and evaluates *MIU: Memory Integrity Utilities* containing three solutions – MemPatrol, FRAMER and spaceMiu. *MIU* suggests new techniques for practical deployment of memory safety by exploiting free resources with the following focuses: (1) achieving memory safety with overhead  $< 1\%$  by using concurrency and

trading off prompt detection and coverage; but yet providing eventual detection by a monitor isolation design of an in-register monitor process and the use of AES instructions (2) complete memory safety with near-zero false negatives focusing on eliminating overhead, that hardware support cannot resolve, by using a new tagged-pointer representation utilising the top unused bits of a pointer.





# Publications

---

- Myoung Jin Nam, Periklis Akritidis, and David J Greaves. FRAMER: Fast Per-object Metadata Management for Memory Safety. Paper presented at the Annual Arm Research Summit, 2018. Cited on pages 23, 53, and 58.
- Myoung Jin Nam, Periklis Akritidis, and David J Greaves. FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19*, page 612626, New York, NY, USA, 2019. Association for Computing Machinery. Cited on pages 23, 53, and 58.
- Myoung Jin Nam, Wonhong Nam, Jin-Young Choi, and Periklis Akritidis. MemPatrol: Reliable Sideline Integrity Monitoring for High-Performance Systems. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 48–69, Cham, 2017. Springer International Publishing. Cited on pages 23, 54, and 123.



# Acknowledgments

---

First of all, the completion of my PhD would not have been possible without all the support of my supervisor Dr. David J. Greaves. I would like to express my gratitude for his support, advice, and patience.

I would like to extend my gratitude to my examiners, Prof. Jon Crowcroft and Prof. Alastair F. Donaldson at Imperial College London, for sharing their valuable ideas and filling the viva with kindness.

My sincere thanks also go to Prof. Lawrence C. Paulson, Prof. Byron Cook at University College London, and Dr. Sean B. Holden for their constant encouragement and help.

I am indebted to Dr. Periklis Akritidis at Niometrics for his insightful guidance, support, and profound belief in my work and me, along with constructive criticism that pushed me to become a better version of myself.

My gratitude extends to Dr. Stephen Kell at University of Kent, Prof. Guy Lemieux at University of British Columbia, Prof. Jin-Young Choi at Korea University, Prof. Wonhong Nam at KonKuk University, Christos Rikoudis at Niometrics, and Dr. Matt Staats at Google for their support and helpful advice that improved my work.

I gratefully acknowledge the funding from the Research Foundation of Korea.

I would like to thank my former lab mates from the Networks and Operating Systems Group and Automated Reasoning Group and friends for their company and care that made my life in Cambridge a wonderful time, especially (I will omit titles of 'Dr' from this point. Sorry, Doctors!) – Julien Quintard, Martin A. Ruehl, Cecile Ritte, Bence Pasztor, Shu Yan Chan, Atif Alvi, Aisha Elsafty, Jordan Auge, Jean E. Martina, Damien Fay, Ilias Leontiadis, Charalampos Rotsos, Thomas Tuerk, Joon Woong Kim, Maria Rita Massaro, Foula Vagena, Mina Brimpari, Daniel Holland, Tony Coleby, Nicole Coleby, and Vassilis Laganakos.

I am also thankful to dear friends for their company during my time in Seoul and Singapore – Hyunyoung Kil, Jeongeun Suh, So Jin Ahn, Nickos Ventouras, Miyoung Kang, Chong Yi, and many more.

My deepest gratitude goes to my parents and two brothers for their support and everything.

And finally, this thesis is dedicated to my dearest Miu – my little girl, my fur angel – in heaven.



# Contents

---

<b>Abstract</b>	<b>3</b>
<b>Publications</b>	<b>7</b>
<b>Acknowledgements</b>	<b>9</b>
<b>Table of contents</b>	<b>14</b>
<b>List of figures</b>	<b>15</b>
<b>List of tables</b>	<b>17</b>
<b>1 Introduction</b>	<b>19</b>
1.1 Contributions . . . . .	22
<b>2 Background</b>	<b>25</b>
2.1 Memory Safety . . . . .	28
2.1.1 Spatial Memory Safety . . . . .	29
2.1.2 Temporal Memory Safety . . . . .	35
2.1.3 Architectural Support and Capability Model . . . . .	37
2.2 Type Safety . . . . .	41
2.3 Control-Flow Protection . . . . .	45
2.4 Concurrent Monitoring . . . . .	48
2.4.1 Monitor Isolation . . . . .	50
2.4.2 Kernel Integrity Monitors . . . . .	50
2.4.3 Cryptographic Key Protection . . . . .	50
<b>3 Overview</b>	<b>53</b>
3.1 FRAMER . . . . .	53
3.2 spaceMiu . . . . .	54
3.3 MemPatrol . . . . .	54

<b>4</b>	<b>FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications</b>	<b>57</b>
4.1	Overview . . . . .	57
4.2	FRAMER Approach . . . . .	60
4.2.1	Frame Definitions . . . . .	60
4.2.2	Frame Selection . . . . .	62
4.2.3	Metadata Storage Management . . . . .	63
4.3	FRAMER Implementation . . . . .	68
4.3.1	Overview . . . . .	68
4.3.2	Memory Allocation Transformations . . . . .	69
4.3.3	Memory Access . . . . .	71
4.3.4	Interoperability . . . . .	72
4.4	FRAMER Applications . . . . .	73
4.4.1	Spatial Memory Safety . . . . .	73
4.4.2	Temporal Memory Safety . . . . .	76
4.5	Optimisations . . . . .	77
4.6	Evaluation . . . . .	79
4.6.1	Memory Overhead . . . . .	79
4.6.2	Slowdown . . . . .	81
4.6.3	Data Cache Misses . . . . .	83
4.6.4	Instructions Executed . . . . .	85
4.6.5	Branch Misses . . . . .	86
4.7	Discussion . . . . .	87
4.7.1	Comparison with Other Approaches . . . . .	87
4.7.2	Hardware Implementation of FRAMER . . . . .	89
4.7.3	Additional Optimisations . . . . .	89
4.8	Conclusion . . . . .	90
<b>5</b>	<b>SpaceMiu: Practical Type Safety for C</b>	<b>91</b>
5.1	Overview . . . . .	91
5.2	Types and Type Relations . . . . .	93
5.2.1	Type Representation . . . . .	93
5.2.2	Physical Equality . . . . .	94
5.2.3	Physical Sub-typing and Type Relation . . . . .	95

---

5.3	Run-Time Typecast Checking . . . . .	96
5.3.1	Object-to-type Mapping . . . . .	98
5.3.2	Per-object Metadata Management and Pointer-to-Type Mapping . . . . .	100
5.3.3	Type Confusion Checking . . . . .	102
5.4	Union Type . . . . .	103
5.5	spaceMiu Implementation . . . . .	106
5.5.1	Creation of Type Descriptors . . . . .	107
5.5.2	Program Initialisation . . . . .	108
5.5.3	Memory Allocation . . . . .	108
5.5.4	Type Cast . . . . .	110
5.5.5	Memory Access . . . . .	111
5.5.6	String Functions . . . . .	111
5.6	Evaluation . . . . .	111
5.6.1	Memory Overheads . . . . .	112
5.6.2	Slowdown . . . . .	114
5.6.3	Executed Instructions . . . . .	115
5.6.4	L1 D-cache Misses . . . . .	116
5.6.5	Branch Prediction . . . . .	117
5.7	Discussion . . . . .	118
5.7.1	Effective Type Detection for Heap Objects . . . . .	118
5.7.2	String Functions and Effective Type . . . . .	118
5.7.3	Aliasing Rules . . . . .	119
5.7.4	Per-object Metadata Placement . . . . .	120
5.8	Conclusion . . . . .	120
<b>6</b>	<b>MemPatrol: Reliable Sideline Integrity Monitoring for High-Performance Systems</b>	<b>123</b>
6.1	Overview . . . . .	123
6.2	Threat Model . . . . .	125
6.2.1	Sideline Integrity Monitoring . . . . .	125
6.2.2	Heap Integrity . . . . .	126
6.3	Monitor Thread Isolation . . . . .	127
6.3.1	Protection of Data Structures in Memory . . . . .	128
6.3.2	Protection of Code . . . . .	131

---

6.3.3	Terminating or Tracing the Monitor Thread . . . . .	132
6.3.4	Faking Application Termination . . . . .	133
6.3.5	Detection of Normal and Abnormal Termination . .	133
6.3.6	Minimizing Performance Impact . . . . .	134
6.3.7	Limitations . . . . .	135
6.4	Case Study: Heap Integrity . . . . .	135
6.4.1	Memory Pools . . . . .	136
6.4.2	Integration with the Monitored Application . . . . .	137
6.4.3	Cryptographically Generated Canary Values . . . . .	138
6.4.4	Canary Recycling . . . . .	139
6.5	Evaluation . . . . .	139
6.5.1	Integration with NCORE . . . . .	139
6.5.2	Experimental Results . . . . .	140
6.6	Discussions . . . . .	144
6.6.1	Tunable Overhead . . . . .	144
6.6.2	Memory Safety . . . . .	145
6.7	Conclusion . . . . .	145
<b>7</b>	<b>Conclusions</b>	<b>147</b>
<b>8</b>	<b>Appendix</b>	<b>149</b>
8.1	Proofs . . . . .	149
8.1.1	Proof 1 . . . . .	149
8.1.2	Proof 2 . . . . .	149
	<b>Bibliography</b>	<b>151</b>

# List of figures

---

2.1	Embedded Metadata . . . . .	28
2.2	Off-by-one byte . . . . .	32
2.3	Disjoint metadata structures of Address Sanitizer and MPX . . . . .	33
4.1	Aligned frames in memory space . . . . .	61
4.2	Tagged pointer . . . . .	64
4.3	Access to division array . . . . .	66
4.4	Overall architecture of FRAMER . . . . .	68
4.5	In-frame checking at pointer arithmetic . . . . .	75
4.6	Normalised memory footprint (maximum resident set size) . . . . .	80
4.7	Normalised runtime overhead . . . . .	81
4.8	Runtime overheads for metadata management and retrieval . . . . .	82
4.9	Normalised L1 D-cache load misses per 1000 instructions (MPKI) . . . . .	84
4.10	Normalised L1 D-cache load miss count . . . . .	85
4.11	Normalised dynamic instruction count . . . . .	87
5.1	C example with typecasts . . . . .	92
5.2	Atomic Types . . . . .	94
5.3	Aggregated Types . . . . .	94
5.4	Physical Equality . . . . .	95
5.5	Metadata Storage and Type Descriptors . . . . .	98
5.6	Entries of Type Descriptors . . . . .	102
5.7	Union's safecast . . . . .	106
5.8	Overall architecture of spaceMiu . . . . .	107
5.9	Normalised Memory Footprint . . . . .	112
5.10	Normalised runtime overhead . . . . .	112
5.11	Normalised Dynamic Instruction Count Overhead . . . . .	113
5.12	Normalised L1-D Cache Miss Count Overhead . . . . .	113
5.13	Normalised Branch Miss Count Overhead . . . . .	114

5.14	String functions and effective type . . . . .	119
6.1	Protection of data in memory . . . . .	128
6.2	Untrusted-memory data type and access routines . . . . .	130
6.3	Canary-monitoring integration API . . . . .	137
6.4	Secure canary checking . . . . .	138
6.5	Relation between system read throughput and maximum detection latency . . . . .	142

# List of tables

---

4.1	Instrumented codes . . . . .	71
4.2	Summary averages over all benchmarks . . . . .	79
6.1	Cache hit rates . . . . .	141
6.2	NUMA effects . . . . .	143



# 1

## Introduction

---

C/C++ languages have *low-level* features such as providing a set of bit manipulators, allowing assumptions about the underlying hardware architecture to take advantages of hardware-specific behaviors/features, and compiler support for inline assembly language. Especially C/C++ has *pointers* which allow (and often require) direct manipulation of memory contents.

The visibility of memory layout in C or C++ has been two sides of the coin – it provides high performance however the allowance of arbitrary pointer arithmetic and type casting imposes the danger of memory corruption, which makes C/C++ languages *unsafe*. Those memory errors may cause unexpected termination of programs. Even worse, security exploits use memory safety vulnerabilities to corrupt or leak sensitive data, and hijack a vulnerable program’s control flow.

Despite advances in software defenses, exploitation of systems code written in C or C++ is still possible [82, 122, 129, 23]. In response, several defence techniques have been proposed to make software exploitation hard.

Current defenses fall in two basic categories: those that let memory corruption happen, but harden the program to prevent exploitation, and those that try to detect and block memory corruption in the first place. In the first category, for instance, Control-flow Integrity (CFI) [1, 75, 149, 150, 151] models all allowable control flows in a statically-computed Control-flow Graph (CFG), while Address Space Layout Randomization (ASLR) [102] hides the available CFG when the process executes. Both approaches can be bypassed [41, 120], since memory corruption can still occur, albeit exploitation is much harder.

The second category, providing fine-grained and strong memory safety enforcement, includes approaches that detect and block memory safety violations. The approaches instrument the program and maintain runtime metadata for access rights to block unintended accesses at runtime [6, 35, 53, 88, 36, 93, 114, 25]. Most of these systems are based on an *inline reference monitor* [112, 37] offering deterministic guarantees by preventing memory corruption in the first place. By embedding checks into the binary code during compilation or via binary rewriting, inline reference monitors can enforce integrity guarantees for the program's memory accesses or control-flow. Violations are detected promptly, with the instruction at fault identified, which greatly facilitates debugging. These memory safety solutions, based on inline monitors, are indispensable for finding memory errors in C/C++ programs during development and testing [94, 114].

Unfortunately for production deployment as an *always-on* solution, those approaches checking individual memory access are still heavy [122]. Their tracking of all objects (or pointers) incurs heavy performance overheads. Performance is critical for adoption since unsafe languages like C/C++ are employed for performance-sensitive applications.

Whilst pushing the limit of performance with novel techniques, researchers have made trade-offs among properties of memory safety enforcement: detection coverage [4, 146, 5], detection timing [141], compatibility [11, 93, 57] and performance. Some early techniques trade off *compatibility* for high locality of reference. One example is so-called *fat pointers* [11, 93, 57], a new pointer representation that stores extra metadata with the address of an object that the pointer points to. Fat pointers provide the best speed, but unfortunately impose binary incompatibility issues with external modules especially pre-compiled libraries. It is desirable to minimise the disruption owing to tacit assumptions by programmers and compatibility with existing code or libraries that cannot be recompiled.

To avoid breaking binary compatibility by changing object memory layout, more recent approaches inevitably chose to bear some performance degradation. Some of them decouple metadata from a pointer representation and store them in a *disjoint metadata*. The cost of such fine-grained memory safety storing per-object [33, 6, 114] (or per-pointer) [88, 143, 53,

---

87] metadata in a remote region is dominated by metadata updates and lookups, making efficient metadata management the key for minimizing performance impact. These solutions focus on reducing run-time overheads by (1) sacrificing detection coverage of memory errors [4, 146, 5] or precision [6, 9, 8]; or (2) wasting memory space with excessive alignment or large *shadow memory spaces* [6, 47, 89, 114] referring to a memory region as a mirror copy of an application space. Some other solutions trade accuracy for speed by allowing false negatives, and hence are more useful for troubleshooting than security. They still provide wide detection coverage but have evolved to keep the performance degradation as little as possible to reduce the time for software testing during development.

In most cases, it is reasonable to prioritise speed over efficiency in space amongst these two main performance assessment criteria, considering that memory safety solutions are normally used during development and it is more critical to reduce time scale than memory resources. However this perspective invites debates for production deployment. There are systems whose memory efficiency is as important as time, such as embedded systems with limited memory space or I/O server systems. In addition, some causes of run-time overheads can be easily resolved with hardware acceleration e.g customised instruction sets, while memory overhead cannot go away even with the hardware support.

Unfortunately inline reference monitors providing fine-grained memory protection have not fully resolved overheads caused by tracking individual memory allocations and accesses to them despite the advances in the techniques and trade-offs. For light-weight memory safety enforcement, researchers proposed replacing inline security enforcement with *concurrent* monitors [148, 125, 111, 109, 81]. In principle, such approaches can minimize the performance overhead on the protected application by offloading checks to the concurrent monitor. Detection, however, now happens asynchronously, introducing a detection delay. This weaker security guarantee is nevertheless still useful. For example, in the case of passive network monitoring systems, it helps validate the integrity of the system's past reports.

These proposals, however, face significant challenges. For some, the delay introduced before the detection of memory safety violations opens up

a vulnerability window during which the attackers have control of the program's execution and may attempt to disable the detection system. This undermines the guarantee of *eventual* detection, even worse, security that those protections aim to achieve by sacrificing spontaneous detection of memory errors. For others, attempts to isolate the monitor during the vulnerability window degrade performance. Finally, these solutions have been designed for general purpose systems, and their communication and synchronization overheads between the monitor and the application threads can be prohibitive for high-performance applications.

## 1.1 Contributions

This research demonstrates trade-offs between detection coverage, detection timing and performance; pushes the limit of performance metrics to extreme depending on the deployment; and improves memory safety. We propose and implement *Memory Integrity Utilities* (MIU), run-time verification systems for low-cost memory safety enforcement, that exploit free resources. Each approach sacrifices a subset of properties for others depending on the goal, and lowers overhead that is expensive or difficult to resolve with hardware support.

MIU proposes both:

- (1) an *inline* monitor prioritising near-complete memory safety with similar increase in run-time overhead to existing approaches but much lower space overhead.
- (2) a *sideline* monitor providing the minimal performance degradation by sacrificing timely and immediate detection.

Firstly, our inline monitor statically instruments an application and halts program execution at security violations. The goal of the monitor is to provide *fine-grained* and *deterministic* memory protection without relying on probability, so that it can also be used during development stage. For its deployment in practice, overhead must be kept low. Compared to existing approaches, our solution provides higher efficiency in data cache and memory footprint by utilising the top unused bits in 64-bit pointers. In our

experiments, Address Sanitizer [114] is faster, however both cause 2x slowdown, and the run-time overhead for our approach will be resolved with hardware acceleration (customised instruction sets). While keeping overheads for memory and data cache low, we remove false negatives that may incur in some previous approaches thus guarantee near-complete memory safety. Another advantage of this approach is its scalability: it can be used for memory safety, type safety, thread safety and garbage collection, or any solution that needs to map pointers to metadata.

This inline reference monitor is evaluated on two use cases with run-time verification systems for C programs detecting:

- (i) FRAMER: array-out-of bounds and some cases of dangling pointers
- (ii) spaceMiu: type confusions in C programs.

The first system, FRAMER [90, 91], illustrates the use of the capability framework on *spatial memory safety* that guarantees near-zero false negatives, allowing inexpensive validation of pointer dereferences by associating pointers to object metadata containing bounds information. The second system, spaceMiu, presents the application of the tagged pointer capability model to *type safety* detecting unsafe type casts, that also violate spatial memory safety. This work defines a type relation for the C language, that does not support type hierarchy, and *unsafe* type conversion in C programs, inspired by CCured [93]; and implements run-time type confusion verifier utilising efficient per-object type information.

Another contribution of this dissertation is a *sideline* monitoring system, *MemPatrol* [92], for practical deployment for high-performance systems. This system realises very low performance degradation by pushing other memory safety properties (immediate detection and error coverage) to extreme. *MemPatrol* does not detect errors in timely order, yet detects them *eventually*. The trade-off drops the overhead down to  $< 1\%$ , which is lower than  $5\%$ , that is commonly acceptable for production deployment. *MemPatrol* implements a *concurrent* monitor detecting memory errors, and using concurrency minimises the performance impact to a target program while allowing *configurable* overheads, that can be useful for any systems. This work addresses one of the challenges of concurrent monitors – a monitor

being compromised by attackers through memory corruptions during the detection delay caused by concurrency – by monitor isolation techniques that leverage CPU registers. It takes advantage of the AES instruction set of Intel processors [45] to implement CPU-only cryptographic message authentication codes (MACs), and stores critical information in regular registers of user-mode programs.

This dissertation proposes, designs and evaluates these three run-time verification approaches that improve memory safety.

# 2

## Background

---

There are two major categories in security enforcement depending on their goal. The first category is to detect and defend memory corruption in the first place by enforcing memory safety, offering stronger security guarantees. The second category aims at a favourable *security-to-overhead* ratio rather than complete memory safety. Those approaches in the second category let memory corruption occur in the first place but they harden a program execution, making it difficult to exploit memory corruptions for attackers, however exploitations are still possible with well-structured attacks.

*Probabilistic* solutions are usually based on *randomization* or *encryption* e.g. *Instruction Set Randomization*, *Address Space Randomization*, or *Data Space Randomization*. Randomization protections introduce entropy to prevent exploits of safety violations. For instance, Data Space Randomization makes it difficult for attackers to know how to replace values in code pointers by randomizing representation of all data. Address Space Layout Randomization (ASLR) [102, 40] hides the available Control-Flow Graph (CFG) when the process executes. It mitigates control-flow hijacking attacks by randomizing the location of code and data and thus the potential payload address. Both approaches can be bypassed [41, 120] e.g. initial information leakage of a code pointer and guessing attacks expose a program in memory and enable attackers to construct exploits to bypass ASLR however they can block the majority of attacks. The overhead of ASLR is negligible so it has been used in practice but the case to enforce full ASLR on Linux shows 10-25% overhead, which prevents deployment. The overhead comes from Position Independent Executables (relocatable executables) on 32-bit

machines and so ASLR should be enforced only for libraries by default on most distributions. Many protections with probability are in the second category that prevents attackers from exploiting memory corruptions rather than detecting vulnerabilities.

However some of protections built on a probabilistic model belong to the first category of preventing memory corruptions. Stack smashing protection, such as StackGuard [26], uses random values for stack cookies (or canaries) to detect memory overwrites to return addresses (§ 2.3). Although the detection coverage is not as wide as other memory protection solutions, e.g. it cannot detect data reads beyond boundaries, these solutions using encrypted cookies are widely deployed due to their very low overhead and great compatibility.

The other approaches enforce a *deterministic* safety policy by implementing a low-level *reference monitor* [37, 112]. A reference monitor observes the program execution and halts it whenever it is about to violate the given security policy, helping remove security vulnerabilities. While traditional reference monitors enforce higher-level policies, such as file system permissions, and are implemented in the kernel (e.g., system calls), more recent reference monitors enforce lower-level policies, e.g., memory safety or control-flow integrity. They can be implemented in two ways: (1) in hardware or (2) by embedding the reference monitor into the code through instrumentation. For instance, Code Integrity + Non-executable Data is enforced by the hardware, as modern processors support both non-writable and non-executable page permissions [79, 80]. Hardware support for protection with coarse granularity causes negligible overhead, however hardware acceleration may not resolve overhead of low-level monitors with fine granularity. We discuss hardware implementation especially architectural support for low-level monitoring in § 2.1.3.

The alternative to hardware support is adding the reference monitor dynamically or statically to the code. In this section, we focus only on solutions which transform existing programs to enforce various policies.

Firstly, *dynamic* (binary) instrumentation [94, 77, 16, 103] can be used to dynamically insert run-time checks into unsafe binaries at run-time. It supports arbitrary transformations but introduces some additional slowdown due to the dynamic translation process. Simple reference monitors, how-

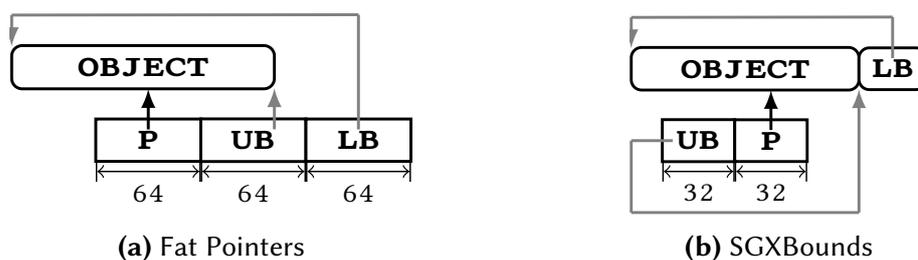
---

ever, can be implemented with low overhead: for instance, a *shadow stack* costs less than 6.5% performance for SPEC CPU2006 in [103]. More sophisticated reference monitors like *Taint Checking* [14] or *ROP detectors* [28] causes overheads that exceed 100% and are unlikely to be deployed in practice.

*Static* instrumentation inlines reference monitors at compile time. This can be done by the compiler or by static binary rewriting. Inline reference monitors can implement any safety policy and are usually more efficient than software dynamic solutions, since the instrumentation is not carried out at run-time. Those approaches provide deterministic and immediate detection of memory errors with fine granularity, however their high overhead is one of the biggest challenges to deploy them in practice. Other performance-optimised solutions for inline monitoring also incur high and unpredictable overheads [52].

The most widely used security protections implementing an inline reference monitor are static Control-Flow Integrity (CFI) and memory safety. CFI restricts the control-flow of an application to valid execution traces by monitoring the program at runtime and comparing its state to a set of pre-computed valid states. CFI is a defense that leverages run-time monitors to detect specific attack vectors (control-flow hijacks for CFI) and flags exploit attempts at run time. All modern compilers implement a form of CFI with low overhead but different security guarantees. If security properties are violated i.e. an invalid state is detected, an alert is raised; reference monitors usually terminate the application. Control-flow hijacking is usually the primary goal of attacks and many CFI techniques defend them at the cost acceptable to practical deployment (<5%). We discuss more about CFI in § 2.3.

On the other hand, memory safety monitors pointer dereferences. It usually tracks memory allocation (or pointers) and compares a pointer to be dereferenced with stored metadata such as bounds information or memory allocation status (alive or de-allocated). Unfortunately inline reference monitors enforcing memory safety suffer from high overheads (100%), since they check every memory allocation/release and access. So they often target development settings when testing a program. Memory corruption can be exploited to carry out other types of attacks as well and memory



**Figure 2.1:** Embedded Metadata: P, UB, and LB represent a pointer itself, upper bound, and lower bound, respectively.

safety provides a wider range of protection. However the overhead has not been fully resolved yet and performance is one of the main challenges of memory protection mechanisms.

In the following subsections, we discuss different software security vulnerabilities and identify the approaches to detecting the vulnerabilities and mitigating exploits.

## 2.1 Memory Safety

Enforcing memory safety stops all memory corruption exploits. Our focus is to enforce memory safety based on low-level, inline reference monitors embedding checks that prevent memory errors by transforming existing unsafe code. The instrumentation may be in the source code, intermediate representation, or binary level. For complete memory safety, both *spatial* and *temporal* errors must be prevented without false negatives. In addition, high-rate false alarms are critical for practical deployment, so they must be kept very low. Unfortunately it is extremely expensive to guarantee complete memory safety especially for system languages such as C/C++.

§ 2.1 reviews prior approaches for memory safety based on inline reference monitor that either track objects or pointers by instrumentation and discusses their trade-offs between detection coverage and performance. Another kind of software vulnerabilities, *type confusions*, will be discussed in § 2.2 separately.

### 2.1.1 Spatial Memory Safety

*Spatial* memory errors refer to *buffer overflows*. Buffer overwrites (out-of-bounds writes) can corrupt the content of adjacent objects, or internal data (like bookkeeping information for the heap) or return addresses. Similarly, buffer overreads may can reveal sensitive data or help attackers bypass address space layout randomization.

The only way to enforce complete *spatial memory safety* is to keep track of pointer bounds – the lowest and highest valid address it can point to. Many approaches have been proposed to enforce spatial memory safety in C/C++ programs. Some of these solutions offer extensive memory protection, but they slow down applications significantly.

Spatial memory safety solutions are divided into two categories depending on whether they associate bounds information with individual pointers or objects.

#### Pointer-based Tracking

Approaches tracking pointers associate each individual *pointer* with its metadata holding a valid address range that the pointer is allowed to point to [87]. Metadata is assigned to a pointer at pointer assignment and bounds checking is performed *only* at memory access unlike object-based approaches that may require optional extra-checks at pointer arithmetic operations (as discussed later in § 2.1.1). Holding an address range (the base address  $\sim$  upper bound) provides protection with *byte-granularity* and this permits creation of *out-of-bounds* pointers<sup>1</sup> and pointers to sub-objects that are allowed in C/C++. This makes it easier to detect *internal overflows* such as an array out-of-bounds inside a structure. As long as tracked pointers act inside instrumented codes, pointer-based approaches do not produce false violations. However, if instrumented pointers passed to un-instrumented external libraries are updated there and returned, they lose track of the pointers (this occurs when the non-instrumented code modifies the pointer and does not properly update the bounds metadata).

---

<sup>1</sup>Out-of-bounds pointers refer to pointers with a value that goes out of bounds of an object that they point to.

Pointer-based approaches are often implemented using *fat pointers* [11, 57, 93, 25]. They define a new pointer representation that embeds metadata (the base and upper bound) with itself as presented in Fig. 2.1a), thus increasing spatial locality of references by removing accesses to retrieve metadata in a remote memory region at run-time checking. Unfortunately the approaches sacrifice *binary compatibility*. Since fat pointers increase the number of bytes used to hold a pointer, they require modification of the memory layout and this damages compatibility with non-instrumented code.

CCured [93], which implements fat pointers, statically annotates a *pointer qualifier* (*safe*, *seq*, and *wild*) on pointers discovered by constraint rules and applies instrumentation depending on the pointer kind. Pointers involved with pointer arithmetic (*seq*) or typecast (*wild*) are equipped with two extra words holding the base and upper bound and especially *wild* pointers cause more overheads. All break binary compatibility. This requires wrapper annotations for calls to external libraries and imposes a conservative garbage collector. Cyclone [57] avoids using garbage collector in favour of region-based memory management, but also diverges more markedly from C. Moreover, updates to fat pointers spanning multiple words are not atomic, while some parallel programs rely on this.

Several pointer-based approaches [143, 89, 53, 87] choose memory layout compatibility over the speed with high locality. Using *disjoint metadata* achieves compatibility by decoupling metadata from a pointer representation and storing metadata in a remote memory region. SoftBound [89] implements both a hash table or shadow memory space to map pointers to the metadata. Unfortunately, the performance overhead of SoftBound is comparably high, 79% on average [89].

Hardware support [30, 53, 100, 135, 63] does not remove this overhead. Intel MPX [53, 87, 99] is an ISA extension that provides hardware-accelerated pointer-checking, using disjoint metadata in a *bounds table* holding per-pointer metadata as illustrated in Fig. 2.3b. Reportedly, MPX suffers due to lack of memory even with small working sets [66], and has turned out to be slow for pointer-intensive programs, owing to exhausting the limited number of special-purpose bounds registers (four registers), re-

quiring spill operations from regions of memory that themselves require management and consume D-cache bandwidth and capacity.

Pointer-tracking approaches provide strong memory protection with near-zero false positives/negatives, but it comes with the additional runtime overhead from metadata copy and update at pointer assignment, while object-based approaches update metadata only at memory allocation/release. In addition, the number of pointers is typically larger than that of allocated objects, so pointer-intensive programs may suffer from heavier runtime overheads. More importantly, it is difficult to achieve full compatibility with them – if a pointer created by the instrumented module is passed to and modified in an un-instrumented module, the corresponding metadata is not updated, causing false violations.

### Object-based Tracking

Due to the compatibility and cost of per-pointer metadata, most techniques track objects. *Object-based* approaches [6, 114, 33, 58, 9, 66, 35, 65, 147] store metadata *per object* and also make a trade-off against complete memory safety. They offer *compatibility* with current source and pre-compiled legacy libraries by not changing the memory layout of objects. In addition, per-object metadata is updated only at memory allocation/release so even if a pointer is updated in an un-instrumented module, the metadata does not go out-of-sync.

Per-object metadata management supports binary compatibility however it has some drawbacks. First of all, the approach does not enforce complete memory safety. For instance, it is more difficult to detect internal overflows compared to using per-pointer metadata management that can simply set up a pointer's metadata with the address range of a sub-object.

One of the disadvantages of object-tracking is that it may not detect memory access violation when pointers exceed the bounds of *right* object (*intended referent*) [58] by pointer arithmetic and then land in the valid range of *another* object. Memory access with these pointer can be seen *valid* in many object bounds-based approaches. Knowing only the bounds of objects is not enough to catch errors at pointer dereferences, because we do not know if the pointer points to the intended referent. To

keep track of them, object-tracking approaches may have to check bounds at pointer arithmetic [58]. However, performing bounds checks only at pointer arithmetic may therefore cause *false positives*, where a pointer going out-of-bounds by pointer arithmetic is not dereferenced as follows:

```
1 int *p;
2 int *a = malloc(100 * sizeof(int));
3 for (p = a; p < &a[100]; p++) *p = 0;
```

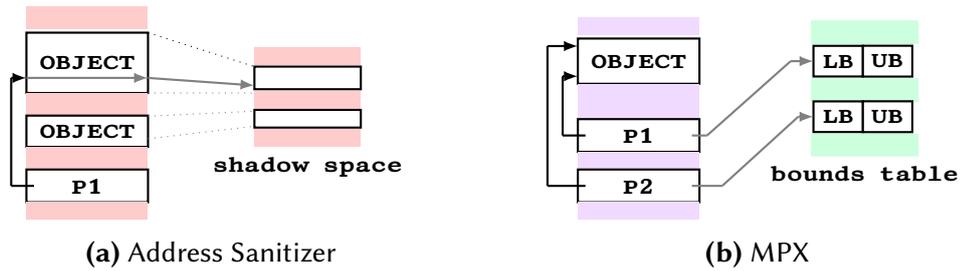
**Figure 2.2:** Off-by-one byte

On exiting the `for` loop, `p` goes out-of-bounds yet is not dereferenced – this is valid according to the C standard.

Therefore, object-based approaches should take a special care for out-of-bounds pointers. The early approach J&K [58] addressed violation of intended referents by padding objects with extra one byte (*off-by-one byte*). This still caused false positives when a pointer legitimately goes beyond more than one byte. A more generic solution to this problem was later provided by CRED [110]. Baggy Bounds Checking [6] instead performs bounds checking at pointer arithmetic, not pointer dereferences, and marks the out-of-bounds pointers so that errors are reported when those are dereferenced.

Object-tracking approaches store a valid range of allocation, so it requires *address range lookups*, while pointer-tracking allows access the corresponding entry using the address of a pointer as a *key* in the metadata table. Performing lookup on the address range of objects is more expensive than lookup by key, or different representations. J&K [58] used a splay tree to reduce the overhead of the range lookup but unfortunately the slowdown is still high (11x-12x). The approach [33] applied *automatic pool allocation* that partitions the memory space using static points-to analysis and stores metadata for each partition. This technique improved the performance up to 120% by reducing the number of bounds lookups.

Modern approaches avoid range lookups and reduce slowdown using a *shadow space* [6, 114, 147, 24, 47, 94, 5]. Shadow space allows single direct *array access* to metadata and this reduces the increase in executed instructions for metadata access, removing metadata lookup in a data structure. Necula



**Figure 2.3:** Disjoint metadata structures of Address Sanitizer and MPX

and Xu [93] creates a mirror copy of a data structure, i.e. *byte-to-byte* mapping, and SoftBound [89] used both a hashtable and shadow space, and showed that using shadow space reduces runtime overhead, on average, by 2/3 compared with using table lookup.

Beyond byte-to-byte mapping of the application space used by early techniques, recent techniques reduced the size of shadow space with compact encoding, at the cost of minimum allocation size or loss of some precision. An example is Baggy bounds checking (BBC) [6]. BBC divides the memory space into *fixed-sized* blocks and mandates object alignment to the base of a block, to prevent metadata conflicts caused by multiple objects in one block. Consequently it pads each object to the *next* power of two, so that each one-byte sized entry stores only  $\log_2(\text{padded object size})$ . This allows compact bounds information and fast lookup by sacrificing memory efficiency. BBC addresses violation of intended referents by checking bounds at pointer arithmetic, not at memory access, and marking pointers going out of *padded* bounds so that they are detected at dereference. This can cause false positives when an illegal pointer comes back within their valid range without being dereferenced. BBC performs *approximate* bounds checking which tolerates pointers going out-of-bounds yet within the padded bound. Memory access with those pointers violates spatial memory safety however it still enforces security, preventing exploits.

Address Sanitizer [114] (ASan) utilizes shadow space differently. Like BBC, it also re-aligns and pads each object but it pads an object with *red-zones* front and back as shown in Fig. 2.3a. While BBC stores  $\log_2$  (padded object size) in a corresponding entry in the shadow space and tolerates access to the pad, ASan considers access to redzones as out-of-bounds,

providing greater precision. The errors are identified by the value in the corresponding entry in the shadow. At memory access, ASan derives the address of its corresponding entry from a pointer, and the entry tells if the address is *addressable*. ASan also prevents some *dangling pointers* by forcing freed objects to stay in a so-called *quarantine zone* for a while. A disadvantage of ASan is that its error detection relies on spatial or temporal *distance*. It loses track of pointers going far beyond the redzone and reaching another object's valid range and it makes it tricky to address false negatives caused by violation of intended referents. ASan addresses this issue by enlarging the space between objects. The wider the redzone, the more errors ASan detects. In addition, *use-after-free* errors cannot be detected, in the cases where dangling pointers are used to access objects after the pointer is freed from the quarantine. ASan detects most errors, but it is less deterministic in theory and trades-off memory space for detection coverage. In our experiments comparing ASan and our prototype, ASan and FRAMER's normalised memory footprints are 8.84 and 1.23, respectively.

Rather than fat pointers or shadow space, *tagged pointers* [65, 66] can instead be used since there are unused bits in a pointer e.g. top 16 bits in a 64-bit pointer. SGXBounds [66] trades-off *address space* for speed and near-complete memory safety. SGXBounds makes objects carry their metadata in a *footer* as shown in Fig. 2.1b, and utilizes the higher 32 bits of a pointer to hold the metadata location. The location is the upper bound of its referent at the same time and object size information is stored in the footer. Storing the absolute address of bounds frees SGXBounds from violation of intended referents that challenge many object-tracking approaches. However this approach works when there are enough spare bits in pointers, which is the case with SGX enclaves, where only 36 bits of virtual address space are currently supported.

Hardware-accelerated tagged pointers are available without sacrificing address space. ARM v8.5 ISA [9, 8] introduces the Memory Tagging Extension (MTE). This assigns a 4-bit tag to each 16 bytes at memory allocation, and tapping memory accesses with incorrect tags in the pointer. However, this approach has 1/16 chance of false negatives at each memory access when the tags in the memory and a pointer match. In case of a real-world exploitation, the random 1/16th chance on an individual access rapidly

disappears to an acceptable level over the course of tens of operations but still provides a usable channel that can be exploited over multiple similar systems in a structured attack. However this solution still relies on probability for memory safety.

In the above subsections, we have discussed approaches to prevent spatial memory errors especially on buffer overflows. Similarly, uninitialised pointers containing garbage values may happen to point at a valid object, so we make sure that pointers are initialised. Lastly, it is difficult to track sub-objects such as an array in a structure or in an outer array, so it may require to manage additional information such as type information.

### 2.1.2 Temporal Memory Safety

Temporal memory safety violations (*dangling pointers*) include null pointer dereferences, *use-after-free*, or *double-free* errors. Dangling pointers arise when pointers are dereferenced (used) after the memory area they try to dereference has been deallocated and returned to the memory management system. Attacks exploiting vulnerable pointers after referent objects are released are as strong as spatial memory safety violation, letting the pointers dereference attacker-controlled data.

Dangling pointers often occur in attempts to access *freed* heap objects and tend to be exploited in conjunction with *type confusion* errors. Assume a dangling pointer pointing to a new object tries to read memory with the freed object's type. When a virtual function of the freed object is called and the virtual function pointer is looked up, the content of the new object will be interpreted as the vtable pointer of the old object. This allows the corruption of the fake vtable pointer, comparable to exploiting a spatial write error, but in this case the dangling pointer is only dereferenced for a read. An additional aspect of this attack is that the new object may contain sensitive information that can be leaked when read through the dangling pointer of the old objects type.

Temporal memory errors also occurs in *stack-allocated* objects. Pointers to a local variable, that are assigned to a global or heap pointers, become dangling when the function of the local variable returns while the pointer is still alive. In this case, dangling pointers are exploited to overwrite sen-

sitive data. Writing through a dangling pointer is similarly exploitable as an out-of-bounds pointer by corrupting other pointers or data inside the new object. When the dangling pointer is an escaped pointer to a local variable and points to the stack, it may be exploited to overwrite sensitive data, such as a return address.

Like spatial memory safety solutions, approaches to ensure temporal memory safety [4, 88, 86, 118, 2, 32] can be divided into two categories: (1) to block temporal memory errors in the first place and (2) to prevent exploitations of dangling pointers.

Like spatial memory safety enforcements (§ 2.1.1), the approaches in the first category also track live objects or pointers to detect dangling pointers. Valgrind’s Memcheck [94] and Address Sanitizer (ASan) [114] track *objects* and mark their status in corresponding entries in the shadow memory. These tools can detect dangling pointers attempting to access after their referent object is de-allocated, as long as new objects are not yet allocated in the locations. However they produce false negatives when the memory region is re-allocated; the area is registered again and the invalid access remains undetected. ASan removes some false negatives by keeping freed objects in the *quarantine* area to prevent use-after-free errors during limited period of time but still may miss some errors. Object-tracking approaches provides less complete temporal memory safety compared to pointer-based approaches like spatial memory safety. Unfortunately, their overheads are quite high. Valgrind, based on dynamic instrumentation, causes higher overhead (10x), while ASan built on LLVM causes around 2x.

Pointer-tracking approaches provide stronger protection. A pointer is associated with allocation/release status information along with bounds information for complete memory safety. A pointer needs to uniuquify live objects i.e. to distinguish not only two objects in different memory regions but also two *temporally-distinctive* objects allocated in the same memory area. CETS [88] assigns a unique ID to each live object and a pointer to the object is associated with the ID. The IDs are stored in a global dictionary. Together with SoftBound [89], a pointer-based spatial memory safety solution, CETS guarantees near complete memory safety. The average overhead is 48% solely and with SoftBound+CETS is 2x. As mentioned

in § 2.1.1, these tools have false violations when pointers are updated in external un-instrumented modules.

Approaches in both categories are still heavy for practical deployment and even for debugging for some benchmarks. Some approaches narrow detection coverage down for low run-time and memory overheads. Cling [4] suggests a customised dynamic memory allocator replacing `malloc` routines. It enforces type-safe memory re-use among only objects with same type and alignment. It does not target detection of all dangling pointers; instead it aims at preventing use-after-free attack exploiting combined vulnerabilities: dangling pointers and type confusion errors. Since it is embodied in the memory allocator, it detects only temporal memory errors of heap objects.

### 2.1.3 Architectural Support and Capability Model

Memory protection systems such as Mondrian Memory Protection [139], Hardbound [30], Capability Hardware Enhanced RISC Instructions (CHERI) [135], M-Machine [21], and industrial approaches such as Intel’s Memory Protection Extensions (iMPX) [53] and Arm’s Memory Tagging Extension (MTE) [8, 9] have been proposed for architectural support for *fine-grained* memory safety.

Mondrian [139] is a memory protection model layered atop page-based virtual memory, to facilitate multiple protection domains. The page table is supplemented by a Protection Look-aside Buffer (PLB) for managing permissions and a set of sidecar registers are paired with general-purpose registers to reduce PLB pressure. This removes requiring userspace ISA changes to support Mondrian, enhancing incremental deployment.

Mondrian provides *address validity* that associates protection properties with regions of *address space* (mentioned in § 2.1.1) rather than a per-pointer basis. It pads all *allocations* to introduce *guard regions* like MMU guard pages. Smaller pads are possible than with pages, while reducing the threshold at which most overflows can be detected. This however prevents the approach from providing protection for sub-allocations such as array entries or individual stack frames, and this may undermine finer-grained protection. This is particularly a concern today when many classes

of exploitable security vulnerabilities are premised on overflows with attacker control over inputs to arithmetic. In addition, Mondrian relies on supervisor mode to manage its protection table. This demands a domain switch for each allocation and free event so protection-domain scalability is limited – each domain requires its own complete protection table, each with substantial memory and initialization expense.

Hardbound [30] is a hardware-assisted *fat-pointer*. The approach provides pointer-based memory safety, not address validation or object-based safety, thus provides finer-grained protection than Mondrian. Hardbound utilises a *shadow space* to store the base and bounds for each pointer-aligned virtual memory location, and another metadata space of *tag bits* to identify pointers in order to reduce the overhead of non-pointers. Bounds information is initialized by the modified software – a memory allocator for heap objects and ideally also by a compiler for stack and global objects. The metadata are then propagated and validated by the hardware i.e. a simulated in-order processor propagates bounds into the shadow table via registers and verifies bounds, when pointers are dereferenced. Un-instrumented libraries and applications will experience less mitigation.

Hardbound provides compatibility: its executables can run on legacy hardware and ABIs are maintained by retaining native pointer size. However its fat pointers are *forgeable*: an instruction adding or modifying the bounds information allows arbitrary bounds, and the tables are accessible via virtual memory. As a result, Hardbound pointers do not constitute a protection domain. Hardbound is also a CISC design that proposes a microcode implementation, and requires transactional memory to write to three table entries atomically.

Intels Memory Protection Extensions (iMPX) [53] provides Instruction Set Architecture (ISA) for spatial memory safety. They describe additions to the x86 ISA to provide hardware-acceleration for compiler-based memory protection with *disjoint metadata*. As with Hardbound, bounds information *per pointer* is stored in architecturally-supported shadow tables or also in software-defined locations (adjacent to the pointer itself) and bounds checking is performed using explicit instructions.

MPX does not support pointer compression. Each 64-bit pointer consumes four metadata: base, upper bounds, the expected pointer value,

and 64 reserved bits. The expected pointer is used for comparison with a pointer value after a pointer returns back from external modules. If there is a mismatch, MPX drops tracking the pointer. MPX sacrifices memory efficiency for compatibility with legacy code which may not update bounds, unlike Hardbound. MPX does not address use-after-free errors but supports typecast checking, making it one of the strongest spatial memory safety enforcements.

Armv8.5-A [8, 9] introduced a new feature called *Memory Tagging*. ARMv8.5-MemTag (MTE) provides architectural support for memory protection using *lock and key* access to memory. Tagging memory implements the lock and pointers (virtual addresses) are modified to contain the key. Memory access is permitted only if the key matches the lock. Memory locations are tagged by adding *four* bits of metadata to each 16 bytes of physical memory (Tag Granule). MTE supports random tag generation and pseudo-random tag generation based on a seed. Due to the limited number of tag bits available (4 bits), the same tag may be allocated for different memory allocations for any specific execution, which causes false negatives. In the aspect of exploitations, the random 1/16th chance on an individual access rapidly disappears to an acceptable level over the course of tens of operations but still provides a usable channel that can be exploited over multiple similar systems in a structured attack. In order to implement the key bits without requiring larger pointers, MTE uses the Top Byte Ignore (TBI) feature of the Armv8-A Architecture. With TBI enabled, the top byte of a virtual address is ignored when using it as an input for address translation. This allows the top byte to store metadata and four bits of the top byte are used to provide the key.

The memory bandwidth impact will depend greatly on the underlying hardware architecture and could be close to zero if the tags are largely implemented in separate hardware resources and blocks are normally cleared on allocation. In addition, the code overheads for heap operations is small but users may prefer to avoid the run-time management overheads by disabling MTE for stack operations.

*Capability-based security* is a different concept of security models. First of all, a *capability* (known as a key) is a communicable and unforgeable token of authority and capability-based security refers to the principle of

designing user programs such that they directly share capabilities with each other according to the principle of least privilege, and to the operating system infrastructure necessary to make the transactions secure.

A capability defines a protected *object reference* which grants a user process *access rights* to interact with an object e.g. reading data associated with an object, modifying the object, and executing the data in the object as a process. The capability logically consists of a reference that uniquely identifies a particular object and a set of access rights and a user program must use the capability to access an object. This capability model can be implemented in a number of different ways: operating systems [29, 62, 137, 115], languages [3, 84], and hardware [140].

M-Machine [21], one of the early systems, is a 64-bit tagged-memory capability system implementing *guarded pointers* tracking pointers (§ 2.1.1). M-Machine pointers are unforgeable. They define a protection domain within a single address space, and support protection-domain switching. It compresses a fat pointer to 64 bits: only power-of-two aligned and sized segments are supported therefore padding is required for common structures that break binary layouts.

Capability Hardware Enhanced RISC Instructions (CHERI) [140] is a hybrid capability model that extends the 64-bit MIPS ISA with *byte-granularity* memory protection. The key features are a capability coprocessor and tagged memory. The coprocessor supports 32 compiler-managed capability registers, each 256-bit wide, holding capabilities. For memory safety, CHERI implements hardware fat pointers in the form of capabilities. Each memory capability holds the base and length fields, describing a segment of memory, and the permissions field indicating an allowed permission for the region such as load data, store data, execute, and load and store for capabilities. It avoids race conditions, which challenged fat pointers, by updating capability fields and tags atomically.

Capability models must preserve *capability integrity*, while allowing userspace management, i.e. capabilities in memory must not be corrupted by general-purpose stores. CHERI implements *tagged memory* to protect in-memory capabilities. Valid capabilities are identified by an extra tag bit associated with each 256-bit location. Any non-capability store clears this bit, protecting capabilities in memory without appealing to kernel mode.

Regional separation [137, 62] is another way to protect capabilities: defining memory regions that can store capabilities distinct from those that can store data. This has limitations, since most programming languages allow pointers and user data neighboring.

Incremental adoption of memory protection systems is critical but compatibility has challenged the deployment of capability systems. CHERI improves earlier capability systems with limited adoption, by hybridizing capability-based addressing with a RISC ISA and MMU-based virtual memory. It provides both *fine-grained* protection as well as *compatibility*.

## 2.2 Type Safety

A program is called *type-safe* when it never explicitly or implicitly converts values from one type to another. One way to ensure type safety of a program is to use type-safe languages. However due to the comparably poor performance of those languages, unsafe system languages such as C or C++ are still widely used for high-performance systems.

Type conversion in C/C++ is sometimes required and useful despite its risk to undermine the integrity/safety of a program. For example, implicit type conversion from an array to a pointer to the 1st element of it (array decay), the result of a floating pointer operations stored in an `int`-typed variable, or `unsigned int` value passed to a function taking a `signed int`. These unsafe type conversions may cause data loss or re-interpretation of a value, therefore we make sure that every memory object including a variable, function argument, and function return value hold an acceptable kind of data; and operations involving values of different types, informally speaking, *make sense* and do not cause data loss, incorrect interpretation of bit patterns, or memory corruption. Type confusions are often combined with dangling pointers for attacks – the memory area of the deallocated object (the old object) is reused by another object (new object). The type mismatch between the old and new object can allow the attacker to access unintended memory.

Several approaches [36, 46, 56, 60, 69] have been proposed to prevent violations of *spatial* memory safety through unsafe typecasting. Those can be

categorised into two depending on if they are based on per-object (pointer) metadata or vtable pointers.

One kind of approach [15, 31, 124, 149] is based on vtable pointers embedded in objects in C++. These approaches save run-time overhead by avoiding manipulation of per-object (or pointer) metadata, that significantly slows down many run-time verification systems. However they usually do not support type checking between non-polymorphic classes, not having a vtable, without breaking binary compatibility. Control-Flow Integrity (CFI) [128, 54, 130, 39] prevents some of these exploits by verifying all indirect control flow transfers within a program to detect control-flow hijacking. However, these techniques address the type confusion problem only partially if control flow is hijacked, i.e., they detect usage of the corrupted vtable pointer, ignoring any preceding data corruption.

The other approaches [46, 93, 56, 60] are based on tracking live objects or pointers. The solutions track individual objects or pointers and store/utilise per-object (per-pointer) type information. Most of them, except fat pointers, secure interoperability with un-instrumented modules and support non-polymorphic classes. However, they incur high run-time overhead to manage metadata. Like deterministic bounds checking, the approaches suggested their own metadata management mechanism to reduce performance loss. This research focuses on the second category of type confusion verification based on per-object (or pointer) metadata.

Type safety enforcements based on per-object/pointer type metadata focus type conversions taking advantage of features of type hierarchy, since the majority of typecasts in C/C++ programs are either *upcasts* (conversion from a descendant type to its ancestor type) or *downcasts* (in the opposite direction). Upcasts are considered safe, and this can be verified at compile time, since if a source type of upcasts is a descendant type, then the type of the allocated object at runtime is also a descendant type.

In contrast, the target type of a downcast may mismatch the run-time type. If a target type is a descendant type of the target type, an access to the object after downcasts may cause *type confusion*, a security vulnerability including internal overflows. Recognition of the run-time type is undecidable, so downcasts require run-time checking to prevent type confusion.

One of the challenges of run-time typecast verification is *pointer-to type mapping*. Typecast pointers to different types may have moved to one of the sub-fields of its referent object and run-time checkers should map the pointer to the corresponding type at the offset. This requires an efficient management of both (1) per-object type information and (2) per-type memory layout. We need to associate an individual object (or pointer) with its object type and map a pointer to the object's type using type information in the metadata storage, which unfortunately causes high overheads. A pointer then should be mapped to a type at the corresponding offset, and examined if the type conversion from the type at the offset to a target type is *safe*.

Another challenge is to determine what type conversion is *safe* e.g. to define what upcasts and downcasts are in C language. First of all, since C language does not support type hierarchy, unlike C++, we should define the hierarchy in C and their relation with other types. For example, arbitrary pointers are frequently converted to `void*` and passed inter-functionally as an argument, which is allowed and considered safe. In CCured [93], `void` is considered to be an *empty* structure and a *prefix* of any type, that is, any type is a *sub-type* of `void`. Under this definition of sub-typing, it is allowed to *upcast* from a pointer in any type into `void*`. The pointer is then typecast from `void*` to other types (desirably restoring its type) for access, requiring type confusion checking at runtime. It is more tricky to judge valid casts between non-`void` types. Strict rules on type conversion can cause false positives, while loosening them can bring false negatives.

CCured [93] ensures both memory and type safety enforcements (§ 2.1.1). This approach observed that most typecasts in real programs are *safe upcasts* (from a pointer to object to a pointer to the first sub-object), and the rest is mostly *downcasts* (in the opposite direction) in C programs written in the object-oriented style. CCured adopted *physical sub-typing* [22] defining type hierarchy in C by flattening aggregate types to *primitives (atomics)* to reduce `wild` pointers (§ 5.2).

CCured is based on *fat pointers* [11, 57, 93]. Approaches tracking an individual pointer [89, 53, 87] provide strong and precise memory/type safety allowing to associate a pointer with any corresponding type i.e. a higher composite type or sub-type. However it suffers heavier run-time overhead

to manipulate per-pointer metadata. In addition, embedding metadata inside objects opens a vulnerability of polluting metadata through memory writes following bad typecasts. CCured addresses this by updating/checking tags at memory access, causing further overhead. Although amongst pointer-tracking approaches, using fat pointers guarantees the high performance, the compatibility issue has not been resolved yet. More modern memory/type safety enforcements store per-object information in disjoint metadata storage to secure the compatibility.

TypeSan [46] is designed for an always-on solution for explicit type checks in C++. This approach, inspired by CaVer [69], focuses on conversion from an instance of a parent class to a descendant class. Downcasting is frequently used if the parent class lacks some of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions of the descendant class that do not exist for a given object, it may use data as a regular field in one context and as a virtual function table (vtable) pointer in another.

TypeSan uses the per-allocation (per-object) paradigm [18, 33, 6, 47, 114] and is composed of two services: type management and metadata storage. Firstly, type management service is responsible for associating type layout with each allocation site and validating downcast operations with these layouts. This service includes (1) a type layout table holding mappings of unique offsets to data fields corresponding to nested types and (2) a type relation table holding compatible types for each class. The second service is metadata storage mapping from object base addresses to type layout tables. Like many other modern run-time verification tracking objects or pointers, the approach creates shadow space [6, 47, 68, 89, 114] for *pointer-to-type* mapping. TypeSan is based on *variable* compression ratio memory shadowing [47]. It allows more detailed memory allocation than other shadow space but it also mandates a uniform alignment. Approaches using shadow space spend more memory space to reduce the run-time overheads with the small increase in dynamic instructions. The trade-off can be merely an issue for debugging/testing during development, but still makes it less useful as an always-on solution on memory-intensive systems such as on embedded systems or IO-servers.

Other approaches [60, 105] exploit debugging infrastructure and instrument the programs allocators. Their analysis is invoked from a debugger, rather than running continuously during execution. Unlike other approaches checking at typecasts followed by pointer dereferences, Libcrunch [60] checks pointer creations not uses, reducing the number of run-time checks, since pointer creations are much less frequent than dereferences. This imposes false negatives since a pointer can be typecast before use. In addition, it unwraps structure type only one level, unlike CCured or one of our prototypes (spaceMiu) unrolling down to primitive types as in physical sub-typing. Libcrunch's check is relatively strict, because there are hardly real code which requires the full permissiveness of physical typing. Some other approaches [19, 76] attach physical types to machine words, unfortunately causes high overheads ( $10x \sim 100x$ ) [145].

One of the challenges of type safety enforcement is to define type hierarchy for C programs and safe/unsafe typecast unlike array out-of-bounds checking which has more straightforward definitions of valid/invalid memory access. Judgements of typecasts in C programs may differ depending on C programmers' coding style and intention.

## 2.3 Control-Flow Protection

Attackers often exploit memory corruptions to take control over the program by diverting its control-flow. If this attack fails by *Code Integrity* enforcement, attackers attempt to use memory corruptions to corrupt a code pointer. *Code Pointer Integrity* aims at preventing the corruption of code pointers.

Code Pointer Integrity [67] in the first category of security enforcement, which detects and defends memory corruption in the first place, offers stronger security guarantees. The approach moves all code pointers to a safe place and prevents the process from corrupting the safe place. Memory corruption can still happen but all controlling data are protected. However, so far, these techniques can be either bypassed or incur a significant overhead to the running process. The probabilistic defense protecting the safe region can be used [42, 38]. For example, using Authenticating Page

Mapper (APM) [42], which builds on a user-level page-fault handler to authenticate arbitrary memory reads/writes in the virtual address space, hardens information hiding with negligible overhead on average  $<1\%$ .

While Code Pointer Integrity aims to prevent the corruption of code pointers, *Control-flow Integrity* detects it. Control-Flow Integrity (CFI) [1, 126, 75, 97, 106, 149, 150] preventing illegal control flow is one example in the second category. CFI mechanisms normally consist of two components: (1) a static analysis component to construct the Control-Flow Graph (CFG) of the application and (2) a dynamic enforcement to restrict control flows according to the generated CFG. In a nutshell, CFI restricts the set of possible target locations (indirect jump target including subroutine return) by executing a run-time monitor that validates the target according to the constructed set of allowed targets. If the observed target is not in that set, the program terminates. Most CFI approaches apply two different mechanism depending on indirect control-flow transfers: *forward-edge* and *backward-edge*. Forward-edge control-flow transfers direct code forward to a new location and are used in indirect jump and indirect call instructions. The backward-edge is used to return to a location that was used in a forward-edge earlier, e.g., when returning from a function call through a return instruction.

For forward-edge transfers, the code is usually instrumented with equivalence checks to ensure that the target observed at runtime is in the set of valid targets. This run-time check depends on the precision of CFI, for instance, a full set check or a simple type comparison.

Checking backward-edge transfers should be handled differently, since equivalence checking does not work when the control-flow is redirected to any *valid* call sites by attackers upon return from a callee. Strong backward-edge protections therefore leverage the context through the previously called functions on the stack. A mechanism that enforces stack integrity ensures that any backward-edge transfers can only return to the most recent prior caller. This property can be enforced by storing the prior call sites in a shadow stack or guaranteeing memory safety on the stack [5], i.e., if the return instructions cannot be modified then stack integrity trivially holds. CFI approximates the CFG for better performance and this allows attackers to leverage the approximation for delivering exploits [41].

On the other hand, in the presence of information leaks, randomization can be bypassed, since hidden code can be revealed [120].

Original CFI mechanism [1] reported overhead with 16% on average for an old version of the SPEC benchmarks, while the maximum measured is as high as 45%. The implementation which uses a shadow stack mechanisms for returns has an additional 10% overhead. Write Integrity Testing (WIT) [5] recorded less overhead (25%), while enforcing both weak memory safety and the same security policy for control-flow hijacking. The experiments presented in [17] showed that more recent, compiler-based CFI mechanisms (e.g. VTV [126], VTI [15], or LLVM-CFI 3.9 [75]) have low overheads (<5%) suitable for widespread adoption in production environments. VTV and VTI are limited to virtual method call, while LLVM-CFI 3.9 supports all indirect calls.

*Dynamic return integrity* is another approach to enforce control flow. Instead of preventing all indirect control transfers, it prevents only illegal returns, since stack-based buffer overflow exploits are the most common form of exploit for remotely taking over code execution. *Stack smashing* [7] is the most well-known control-flow hijacking attack, which exploits a buffer overflow in a local variable to overwrite the return address on the stack. StackGuard [26], the first proposed solution, addressed the attack by placing a secret value (called as cookies/canaries) between the return address and the local variables. Any changes to the cookies are regarded as overwrites to the return address by a buffer overflow and this is detected by the check placed before the return instruction. Stack cookies do not protect indirect calls and jumps, and they are also vulnerable to direct overwrite attacks and information leaks. The mechanism is still popular and widely deployed, since its cost is extremely low – the performance overhead is negligible (less than 1%) and no compatibility issues are introduced.

*Shadow stack* [131, 127] improves canary-based mechanism, extending it to solve information leaks and direct overwrites. Stack Shield [131] creates a shadow stack and stores the saved return address in the shadow. Upon function return, the shadow copy can be compared with the original return address. The shadow stack is not protected but checking if two return addresses match makes attacks much harder, because attackers must successfully corrupt both return addresses in a remote memory region in a

limited time (i.e. before the function returns). To protect the shadow stack itself, RAD [127] uses *guard pages* or switching *write permission* to protect the shadow stack area. While Stack Shield does not protect against direct overwrites with low cost, RAD with stronger protection causes 10x slow-down.

## 2.4 Concurrent Monitoring

Memory safety enforcement based on inline monitoring provides extensive detection coverage of memory errors. However, heavy and unpredictable run-time overheads of inline monitoring applied at so fine-a-grain makes it unattractive for production deployment in performance-critical applications, such as passive network monitoring systems. Even performance-optimized solutions for inline monitoring incur high and unpredictable overheads [52].

The unpredictability of the performance overhead incurred from inline monitoring is a problem by itself. Its runtime overhead is highly dependent on the code being instrumented. A few instructions inserted in a tight loop can translate to a large number of dynamic instructions at runtime, causing a significant performance impact. Moreover, additional memory accesses for security checks may increase memory bandwidth and cache misses. Inlined checks cannot abstain from using the cache hierarchy and slow down the application.

To avoid the costs of inline reference monitors, researchers proposed replacing inline security enforcement with concurrent monitors. [148, 125, 141, 111, 109, 81]. In principle, such approaches can minimise the performance overhead on the protected application by offloading checks to the concurrent monitor. Detection, however, now happens asynchronously, introducing a detection delay. This weaker security guarantee is nevertheless still useful. For example, in the case of passive network monitoring systems, it helps validate the integrity of the system's past reports.

With the advent of multicore machines, utilizing spare CPU cores for security became an attractive approach. Multi-Variant Execution Environments (MVEE) [111] uses spare core capacity to run several slightly dif-

ferent versions of the same program in lockstep, monitoring them for discrepancies. This approach, however, is impractical for high-performance multicore programs utilizing the majority of the CPU's cores, and there can still be a shared cache impact of concurrent monitoring.

Cruiser [148] is one of the original systems to decouple security checks from program execution, running them instead in a thread inside the address space of the monitored process. Cruiser's guarantees are *probabilistic*. Its low detection latency, for the programs it was evaluated on, helps defend against tampering with its data structures, but is not a reliable solution on its own. Moreover, for performance-critical applications using large amounts of heap memory, like network traffic identification systems, the detection latency would increase significantly due to the sheer number of canaries, and should not be relied upon for security. Cruiser also employs pseudo-isolation using ASLR and surrounding its data structures with inaccessible guard pages, in the hope that blind access will trigger segmentation faults. Recent studies [98], however, demonstrate that faith in ASLR-based information hiding is misplaced. Therefore, these systems do not offer a strong guarantee against tampering with the monitor's execution before a compromise is detected. Without reliable isolation, the risk of exploitation remains as long as there is detection delay.

Other software-based techniques utilizing spare CPU cores include ShadowReplica [55] and TaintPipe [81], which aim to improve the performance of dynamic information flow tracking (DIFT) for security. DIFT is a comprehensive protection mechanism, and these solutions demonstrated significant performance improvements over inline DIFT, but the remaining overhead due to the presence of inline stub code remains significant ( $> 2\times$  slowdown over native execution).

Some memory protections with limited detection coverage can be adopted for concurrent monitoring system due to their effectiveness and simplicity, unlike inline monitors with strong protection but high overhead from tracking objects/pointers (§ 2.1.1). Combining them can resolve runtime overheads of inline monitors to the level of practical deployment.

StackGuard [26] detected exploitations of stack-based buffer overflows by placing a *canary* word before the return address, and checking it when the stack frame is deactivated. Similar solutions are currently widely de-

ployed due to their light weight, and the idea has been extended to heap buffer overflow detection [95, 108] with canary checks triggered by memory management routines, library routines, or system calls.

### 2.4.1 Monitor Isolation

*Eventual* detection of errors in monitoring systems may not be guaranteed when they are compromised. Concurrent monitors are also vulnerable to the risk, unless the monitoring systems bear the heavy overheads to protect the monitor itself. The problem of the isolation of monitoring systems has been addressed by using OS kernel or VM hypervisor mechanisms [12, 95, 125]. These are comparatively safer, but come with additional overhead and engineering costs. Instead, *MemPatrol* presents a userspace-based solution, sharing the address space of the main application. Our solution minimises the overhead on the execution of the main application threads while at the same time it allows monitoring the entire memory of the application if so required. Finally, it avoids any engineering costs for the maintenance of custom kernel modifications.

Other software-based isolation mechanisms using Software Fault Isolation (SFI) [133] suffer from overheads because they inline checks to the code that needs to be contained. NativeClient [144] has an average overhead of 5% for CPU-bound benchmarks, which is acceptable for deployment in practice.

### 2.4.2 Kernel Integrity Monitors

Kernel integrity monitors (KIMs) periodically inspect critical kernel memory regions, such as syscall tables and persistent function pointers, isolating themselves from the kernel by residing in hypervisors [50] or hardware such as PCI cards [104]. Some KIMs tackle transient attacks by snooping the bus traffic using PCI cards [83] or even commodity GPGPUs [64].

### 2.4.3 Cryptographic Key Protection

TRESOR [85], a disk encryption system that defends against main memory attacks, uses a similar CPU-only cryptographic mechanism based on AES-

NI [45], but stores its secret key in special CPU debug registers in each core and is kernel-based.

Note that the availability of the AES-NI instruction set is not a hard requirement, but rather an optimization and implementation convenience. For example, Loop-Amnesia [117], a disk encryption system similar to TRESOR, does not rely on AES-NI. We could avoid the dependency on the special AES instruction set by using their CPU-only implementation of AES.

MACs have been previously used in Cryptographic Control Flow Integrity (CCFI) [78] to protect control flow elements such as return addresses, function pointers, and vtable pointers. CCFI also takes advantage of the AES-NI instruction set.



# 3

## Overview

---

### 3.1 FRAMER

FRAMER [90, 91], presented in Chapter 4, is an inline reference monitor that implements a *capability* framework with object granularity. Its sound and deterministic per-object metadata management mechanism enables direct access to metadata by calculating their location from a *tagged pointer* by exploiting unused top 16 bits of a 64-bit pointer.

FRAMER can be the base of a solution for both (1) practical deployment with customised ISA for its efficiency of memory footprint and cache memory and (2) sound runtime verification during development.

This may improve the performance of memory safety, type safety, thread safety and garbage collection, or any solution that needs to map pointers to metadata. FRAMER improves over previous solutions by simultaneously

1. providing a novel encoding that derives the location of per-object metadata with low memory overhead and without any assumption of objects' alignment or size,
2. offering flexibility in metadata placement and size,
3. saving space by removing any padding or re-alignment, and
4. avoiding internal object memory layout changes.

We evaluate FRAMER with a use case on memory safety.

## 3.2 spaceMiu

The spaceMiu design, presented in Chapter 5, a run-time type confusion checker for C programs based on *tagged pointers*-capability model. spaceMiu defines type hierarchy in C programs; validates type conversion using per-object type metadata; and authorises only safe memory access at run time that removes software vulnerabilities in the first place.

Type conversion in C with weak type rules is frequently used and sometimes required, however arbitrary typecasts impose the risk of memory corruption of programs, causing vulnerability exploitations to leak sensitive data and hijack a program's logic. Static analysis can validate type conversion in many cases however some violation should be examined at run-time. Run-time type confusion checkers track live objects (or pointers) and check typecasts using per-object/pointer type metadata. Unfortunately it causes high overheads, making efficient metadata management the key for performance.

Many large-sized C programs are written in an object-oriented style and the majority of typecasts in C (also in C++) programs [93, 116]. If an allocated objects type is a descendant type of the target type at downcast, access to an object after downcasts may cause memory corruptions including internal overflows, commonly known as *type confusion*.

Inspired by CCured's view on sub-typing, our type confusion checker, spaceMiu, defines up/downcast and provides relaxed validity of typecasts. It applies typecast rules conservatively with our efficient per-object type metadata management utilising *tagged pointers*. This work shows how we ensure type safety with tagged pointers-capability model that enables direct access to type metadata. This framework can be used to support multiple potential security enforcements including memory safety in parallel.

## 3.3 MemPatrol

*MemPatrol* [92], presented in Chapter 6, a "sideline" integrity monitor that allows us to minimise the amount of performance degradation at the expense of increased detection delay. Inspired by existing proposals, *MemPatrol* uses a dedicated monitor thread running in parallel with the other

---

threads of the protected application. Previous proposals, however, either rely on costly isolation mechanisms, or introduce a vulnerability window between the attack and its detection. During this vulnerability window, malicious code can cover up memory corruption, breaking the security guarantee of “eventual detection” that comes with strong isolation. The key contributions of this work are (i) a novel userspace-based isolation mechanism to address the vulnerability window, and (ii) to successfully reduce the overhead incurred by the application’s threads to a level acceptable for a performance-critical application. We evaluate *MemPatrol* on a high-performance passive network monitoring system, demonstrating its low overheads, as well as the operator’s control of the trade-off between performance degradation and detection delay.



# 4

## FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications

---

### 4.1 Overview

Security mechanisms for systems programming languages, such as fine-grained memory protection for C/C++ based on inline reference monitoring, authorize operations at runtime using access rights associated with objects and pointers. The cost of such fine-grained capability-based security models is dominated by metadata updates and lookups, making efficient metadata management the key for minimizing performance impact. Existing approaches reduce metadata management overheads by sacrificing precision, breaking binary compatibility by changing object memory layout, or wasting space with excessive alignment or large shadow memory spaces.

With these limitations in mind, *object-capability models* [29, 68, 134, 140], using hardware-supported tags, become very attractive, because they can manage compatibility and control run-time costs. However, they cannot entirely avoid undesirable overheads such as metadata management related memory accesses just by virtue of being hardware-based. In turn, some hardware-based solutions also trade accuracy for acceptable performance [70].

This chapter presents FRAMER [90, 91], a memory-efficient capability model using *tagged pointers* for fast and flexible metadata access. FRAMER provides efficient per-object metadata management that enables direct access to metadata by calculating their location using the (currently) unused top 16 bits of a 64-bit pointer to the object and a compact supplementary table. The key considerations behind FRAMER are as follows.

Firstly, our tagged pointer encoding can enable the memory manager freedom to place metadata in the associated header near the object to maximise spatial locality, which has positive effects at all levels of the memory hierarchy. Headers can vary in size, unlike approaches that store the header at a system-wide fixed offset from the object, which may be useful in some applications. Headers can also be shared over object instances. We do not develop that aspect in the current implementation and we leave for future work to manipulate a memory manager for full freedom to place a header or shared header.

Secondly, the address of the header holding metadata is derived from tagged pointers regardless of objects' alignment or size. FRAMER uses a novel technique to encode the *relative location* of the header in unused bits at the top of a pointer. Moreover, the encoding is such that, despite being relative to the address in the pointer, the tag does not require updating when the address in the pointer changes. A supplementary table is used only for cases where the location information cannot be directly addressed with the additional 16-bits in the pointer. The address of the corresponding entry in the table is also calculated from a tagged pointer. With the help of the tag, this table is significantly smaller compared to typical *shadow memory* implementations.

Thirdly, we avoid wasting memory from any padding and superfluous alignment, whereas existing approaches using shadow space [6, 47, 68, 89, 114] re-align or group objects to avoid conflicts in entries, FRAMER provides great flexibility in alignment, that completely removes constraining the objects or memory. The average space overhead of our approach is 20% for full checking despite the generous size of metadata and the supplementary table in our current design.

Fourthly, this approach facilitates *compatibility*. FRAMER's tag is encoded in otherwise unused bits at the top of a pointer, but the pointer size is unchanged and contiguity can be ensured.

In this study, to achieve *deterministic* memory protection with data memory efficiency, while preserving the full 48-bit address space available in contemporary CPUs, FRAMER sacrifices dynamic instruction count. FRAMER (1) reins in the increase in extra cache misses for metadata (owing to spatial locality compared with a total shadow memory approach) and (2) tolerates an increase in executed instructions for arithmetic operations. This may sound unfavourable since the increase in dynamic instructions is one of the major contributors to performance loss. However, note that we can move to an even sweeter spot in the future where the instruction overhead for calculation is reduced via customised ISA. In addition, the measured performance ends up being better than might be expected – the evaluation shows excellent D-cache performance where the performance impact of software checking is, to a fair extent, mitigated by improved instructions per cycle (IPC). FRAMER's framework provides a novel encoding that derives metadata pointer from an object pointer by exploiting the unused top 16 bits of a 64-bit pointer, lowering both memory footprint and cache misses. FRAMER's metadata management does not make any assumption of object alignment or size. This avoids wasting memory for padding or re-alignment. At the same time, it minimises additional cache misses that are consumed to access metadata in a remote region. In the experiments, normalised L1 D-cache miss counts for FRAMER and ASan on average are 1.40 and 2.31, respectively.

The contributions of FRAMER are the following:

- FRAMER presents an efficient encoding technique for relative offsets that is compact and avoids imposing object alignment or size constraints. Moreover, it is favourable for hardware implementation.
- Based on the proposed encoding, this chapter designs, implements and evaluates FRAMER, a generic framework for fast and practical object-metadata management with potential applications in memory safety, type safety and garbage collection.

This work demonstrates promising low memory overheads and high instruction-level parallelism.

## 4.2 FRAMER Approach

In a nutshell, FRAMER places per-object metadata close to their object and calculates the location of metadata from only (1) an *inbound* pointer<sup>1</sup> and (2) additional information tagged in the otherwise unused, top 16 bits of the pointer. We exploit the fact that *relative addresses* can be encoded in far fewer bits than *absolute* addresses with assistance from the memory manager to restrict the distance between the allocation for an object and a separate object for its metadata. In the current implementation, the metadata is stored in front of the object, essentially as a *header* that an object carries with itself, requiring only a single memory manager allocation. Our own memory manager can be implemented for the future design. For the remaining cases where the relative address cannot fit in a 16-bit tag, we use a compact supplementary table to locate the header. The tag encodes when this is the case, and also sufficient information to locate the supplementary entry.

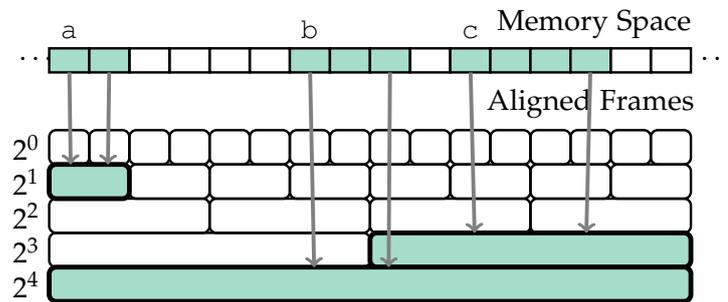
We are now going to introduce the concept of *frames* used to encode relative offsets. We first define frames in § 4.2.1 and show how to calculate an object's *wrapper frame* in § 4.2.2. In § 4.2.3 we explain how relative location can be encoded in a tagged pointer using these concepts, and how to exploit this encoding to reduce the supplementary table's size.

### 4.2.1 Frame Definitions

To record the relative location in the top 16 bits of a 64-bit pointer, which are spare in contemporary CPUs, we define a logical structure over the whole data space of a process, including statics, stack, and heap. The FRAMER structures are based on the concept of *frames*, defined as memory blocks that are  $2^n$ -sized and aligned by their size, where  $n$  is a non-negative

---

<sup>1</sup>An inbound pointer is a pointer whose value is within the valid range of an object it points to.



**Figure 4.1:** Aligned frames in memory space: a memory space can be divided into frames that are defined by memory blocks that are  $2^n$ -sized and aligned by their size. A memory object's wrapper frame is the smallest frame completely containing the object. For instance, the 2-byte sized object *a*'s wrapper frame size is  $2^1$  (called 1-frame). In the same way, objects *b* and *c*'s wrapper frames are 4-frame and 3-frame, respectively.

integer. A frame of size  $2^n$  is called *n-frame*. A memory object  $x$  will intrinsically lie inside at least one bounding frame, and  $x$ 's *wrapper frame* is defined as the smallest frame completely containing  $x$ , so there exists only one wrapper frame for  $x$ . For instance, in Fig. 4.1, each sharp-cornered box represents a byte, and contiguous coloured bytes are objects allocated in memory (e.g. object *a* has a size of 2 bytes). Memory space is divided to frames illustrated as round-cornered boxes. Objects *a*, *b* and *c*'s wrapper frames are ( $n = 1$ )-frame (or 1-frame), 4-frame, and 3-frame, respectively. For  $0 \leq m < n$ , we call *m*-frames placed inside an *n*-frame  $f$ ,  $f$ 's *subframes*.

Frames have several interesting properties. Firstly, an *n*-frame is aligned by  $2^m$  for all  $m < n$ . Secondly, an object's wrapper frame size is not proportional to the object's size. As shown in Fig. 4.1, the object *b* has a larger wrapper frame than *c*, even though *b*'s size is smaller. This is because the wrapper frame size for an object is determined by both the object's size and location. Thirdly, as discussed previously, an object's wrapper frame is defined as the smallest frame containing the object. Given an object  $x$ , its wrapper frame is obtained by finding a frame having  $x$ 's base (i.e. lower bound) and upper bound in its lower-addressed  $(n - 1)$ -subframe and higher-addressed  $(n - 1)$ -subframe, respectively. For example, in Fig. 4.1, object *b*'s lower and upper bound are placed in *b*'s wrapper frame (4-

frame)’s lower-addressed and higher-addressed 3-subframes, respectively. It is trivial to prove that an object’s wrapper frame is the frame having the object’s lower and upper bound in its biggest subframes, as presented in Appendix 8.1.1.

Following basic `malloc` semantics, FRAMER does not natively support object movement or growth (we reset its wrapper frame at `realloc`). Therefore, there exists a unique wrapper frame for each object, and it is determined at memory allocation. Since it does not change during the life time of an object, we can encode the metadata location using an offset relative to the wrapper frame. At memory allocation, we determine the wrapper frame for the allocated object and store the metadata offset in the pointer tag.

### 4.2.2 Frame Selection

We now show how to calculate the size of the wrapper frame, given an object. We call an object whose wrapper frame is an  $n$ -frame an  $n$ -object. For any  $k$ -object  $o$ , since its wrapper frame (i.e. a  $k$ -frame) is aligned by  $2^k$  by definition, the addresses of all bytes in the frame coincide in their most significant  $(64 - k)$  bits, and so do the addresses of all bytes in  $o$ . In addition, the base and upper bounds are located in the lower and higher-addressed  $(k - 1)$ -frame, respectively. This means that the  $(k - 1)^{\text{th}}$  least significant bit of the base and that of the upper bound are complementary to each other.

Based on these, we can calculate  $k$ , the binary logarithm ( $\log_2$ ) of  $o$ ’s wrapper frame’s size. Let  $(b_{63}, \dots, b_1, b_0)$  and  $(e_{63}, \dots, e_1, e_0)$  bit vectors of  $k$ -object  $o$ ’s base and upper bound respectively, and  $X$  a *don’t care* value. We derive  $\log_2(\text{wrapper frame size})$  by performing XOR (exclusive OR) and CLZL (count leading zeros) operations as follows ( $b_{63}$  is the most significant):

$$\begin{array}{r}
 (b_{63}, \dots, b_k, b_{(k-1)}, b_{(k-2)}, \dots, b_0) \\
 (e_{63}, \dots, e_k, e_{(k-1)}, e_{(k-2)}, \dots, e_0) \quad \text{XOR} \\
 \hline
 (0, \dots, 0, 1, X, \dots, X) \quad \text{CLZL} \\
 \hline
 (64 - k) \\
 \hline
 \end{array}$$

We then get  $k$  by subtracting the result of the `CLZL` operation from 64, since  $k = 64 - (64 - k)$ .

### 4.2.3 Metadata Storage Management

FRAMER's memory manager places metadata in a *header* before the object contents. For instance, in Fig. 4.3, *a*, *b* and *c* are all objects containing a header. Using any bounding frame as a frame of reference, we can encode the location of the object's metadata (i.e. header) relative to the base of this frame. We can then derive the metadata location given an inbound pointer using the following:

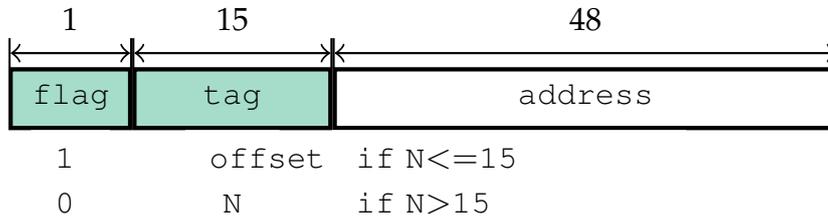
1. the binary logarithm of the bounding frame size ( $N = \log_2 2^N$ )
2. an offset to a header from the bounding frame base

Given an inbound pointer and a bounding  $N$ -frame, aligned by  $2^N$  by definition, we derive the bounding frame's base by clearing the pointer's  $N$  least significant bits. This means that once a bounding frame's  $N$  value is known to us, we can obtain the frame's base without any other information but the address in an inbound pointer's 48 lower bits.

Having the value of  $N$  at hand, we may tag pointers with the offset from the bounding  $N$ -frame's base to the header. However, even with the value of  $N$  provided, the 16 bits of the tag cannot hold the large offsets required for some combinations of wrapper frame size and header location. For instance, a ( $N = 20$ )-object's offset (20-frame's base  $\sim$  the header) may need up to 19 bits.

To encode within the limited space of unused 16 bits of a pointer with both an arbitrary offset and  $N$  value, FRAMER divides the virtual address space into *slots* with a fixed size of  $2^{15}$  bytes, aligned to their size, i.e., 15-frames. Slots are set to a size of  $2^{15}$  so that offsets to the header of objects can be encoded in the unused 15 bits of a pointer (one bit among 16 is reserved for a flag described subsequently). In Fig. 4.3,  $d_a$  is the offset to the header of the object *a*.

FRAMER then distinguishes between two kinds of objects, depending on their wrapper frame size, namely *small-framed* and *large-framed* objects. Small-framed objects are defined as ( $N \leq 15$ )-objects, i.e. objects whose



**Figure 4.2:** Tagged pointer: the tag depends on the value of  $N$  (binary logarithm of the wrapper frame size of a referent object).

wrapper frame size is less than/equal to  $2^{15}$ . Large-framed objects are defined as ( $N > 15$ )-objects. For example, in Fig. 4.3, object *a* is small-framed, whereas *b* and *c* are large-framed. One extra bit, in particular the most significant, is used for a *flag* indicating if the object is small-framed or large-framed as shown in Fig. 4.2. We handle objects differently depending on their kind.

### Small-framed Objects

Small-framed objects are completely contained in a single slot, so any pointer to them is derived to the slot base by zeroing the 15 least significant bits of the pointer. The offset of a small-framed object  $x$ 's header from the base of the slot containing  $x$  is stored in the 15-bit pointer tag. For instance, in Fig. 4.3 we tag pointers to the small-framed object *a* with  $d_a$  (slot0's base  $\sim$  *a*'s header).

We further turn on the most significant bit of the pointer to indicate that the particular object is small-framed. FRAMER then recognises a pointer to a small-framed object by the flag being on and takes the 15-bit tag as an offset to its header from the base of the slot containing the object. This way, we avoid storing the value of  $N$  for small-framed objects.

In summary, when we retrieve metadata from a header of a small-framed object (i.e., flag is on), inbound (in-slot) pointers are derived to the base of the slot by zeroing the 15 least significant bits ( $\log_2(\text{slot\_size}) = 15$ ), and then to the address of the header by adding the offset to the base address of the slot as follows:

```

1 // FLAG_MASK: ~(1ULL << 63)
2 // flag is on

```

```
3
4 offset = (tagged_ptr & FLAG_MASK) >> 48;
5 slot_base = untagged_ptr & (~0ULL << 15);
6 header_addr = slot_base + offset;
7 obj_base = header_addr + header_size;
```

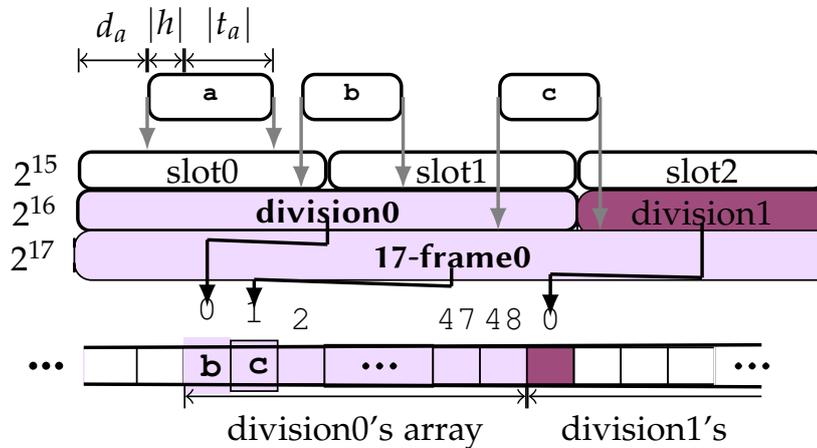
Small-framed objects are overwhelmingly common. Our experiments showed the number of large-framed objects is very low compared to small-framed ones: 1: > 200,000 on average and 1: millions in some benchmarks. This is fortunate, because the header location for small-framed objects is derived from tagged pointers alone, while large-framed objects require additional bits of information. We describe encoding for large-framed objects next.

### Large-framed Objects

Since large-framed objects span several slots, zeroing the 15 least significant bits ( $\log_2$  of slot size) of a pointer does not always lead to a unique slot base, thus the offset in the tag cannot be solely used to derive their relative location. In Fig. 4.3, a pointer to a 16-object `b` can derive two different slot bases (`slot0` and `slot1`) depending on the pointer's value, and that is the case for 17-object `c` (`slot1` and `slot2`). In addition, the offsets from the base of their wrapper frame ( $(N > 15)$ -frame) to an  $(N > 15)$ -object's header may not fit in spare bits. Hence, for large-framed objects, we need to store additional location information. We store these additional bits in a supplementary table, and use a different encoding in the pointer tag to derive the address of the corresponding entry from any pointer to the object. We stress here that the location of this entry is also derived using the tag in a way that enables much smaller tables than typical shadow memory implementations.

During program initialisation, we create a table holding an entry for each 16-frame. We call such a frame a *division*. Each entry contains one sub-array and the sub-array per division is called a *division array*. Each division array contains a fixed number of entries potentially pointing to metadata headers, in the current implementation as follows:

```
1 typedef struct ShadowTableEntryT {
```



**Figure 4.3:** Access to division array: the object *a* is small-framed, while *b* and *c* are large-framed.  $d_a$  is the offset to *a*.  $h$  denotes a header and  $|t_a|$  is the size of *a*. *b* and *c*'s entries are mapped to the same division array. The entries in the division arrays store their corresponding object's header location, while the small-framed object *a* does not have an entry. Only one entry of *division1*'s array is actually used, since the division is not aligned by  $2^{17}$ .

```

2  HeaderTy *division_array[48]; // 64-16
3  } DivisionT;

```

Contrary to small-framed objects, in the tag for large-framed objects we store the binary logarithm of their wrapper frame size (i.e.,  $N = \log_2 2^N$ ) as shown in Fig. 4.2. The address of an entry in a division array is then calculated from an inbound pointer and the  $N$  value, and the entry holds the address of a header. By definition, a wrapper frame of an ( $N \geq 16$ )-object is aligned by its size,  $2^N$ , therefore, the frame is also aligned by  $2^{16}$ . This implies that a ( $N \geq 16$ )-frame shares the base address with a certain division, and is mapped to that division.

Each ( $N \geq 16$ )-object maps to one division array, but that division array contains entries for multiple large-framed objects. In Fig. 4.3, both *division0* and *17-frame0* are mapped to *division0*. Their mapped division (*division0*) is aligned by  $2^{17}$  at minimum, while *division1* is aligned by  $2^{16}$  at max.

The tag  $N$  can be used as an index into the division array to associate a header pointer, stored in an entry in the division array, with each large-framed object mapped to the same division. For each  $N \geq 16$ , at most one  $N$ -object is mapped to one division array, and the proof is presented in Appendix 8.1.2. We use the value  $N$  as an index of a division array, and tag  $N$  in the pointer. Given a  $N$  value-tagged pointer (`flag==0`), we derive the address of an entry as follows:

```

1 // UBASE: division base of userspace's base
2 // SCALE: binary logarithm of division_size, i.e. 16
3 // TABLE: address of a supplementary table
4 // flag is off
5 // p is assumed tag-cleaned here
6
7 frame_base = p & (~0ULL << N);
8 table_index = (frame_base - UBASE) / (1ULL << SCALE);
9 DivisionT *M = TABLE + table_index;
10 header_addr = M->division_array[N - SCALE];

```

The base of the wrapper frame (i.e. the base of the division) is obtained by zeroing the least significant  $N$  bits of the pointer. The address of its division array is then derived from the distance from the base of virtual address space and  $\log_2(\text{division\_size})$  ( $2^{16}$ ). Finally we access the corresponding entry with the index  $N$  in the division array.

Entries in a division array may not always be used, since an entry corresponds to one large-framed object, which is not necessarily allocated at any given time, e.g. if object `b` is not allocated in the space in Fig. 4.3, 0th element of `division0`'s array would be empty. This feature is used for detecting some dangling pointers, and more details are explained in § 4.4.2.

Unlike existing approaches using shadow space, FRAMER does not re-align objects to avoid conflicts in entries. Our *wrapper frame-to-entry* mapping allows wrapper frames to be overlapped, that gives full flexibility to memory manager.

We could use different forms of a header such as a *remote* header or a *shared* header for multiple objects, with considering a cache line, stack frame, or page. In addition, although we fixed the division size ( $2^{16}$ ), future designs may offer better flexibility in size.

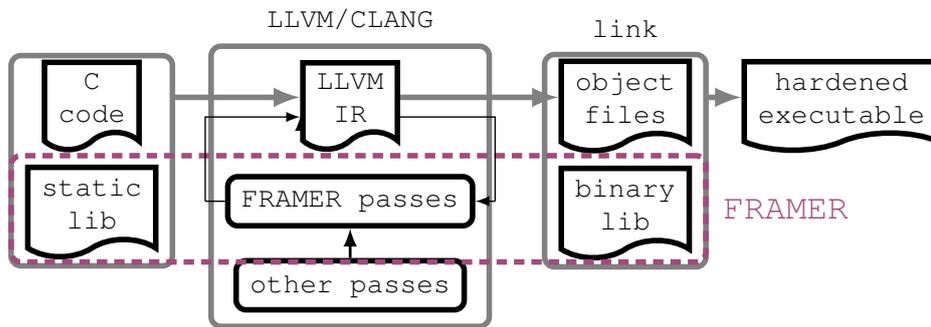


Figure 4.4: Overall architecture of FRAMER

We showed how to directly access per-object metadata only with a tagged pointer. Our approach gives great flexibility to associate metadata with each object; gives full freedom to arrange objects in memory space, that removes padding objects unlike existing approaches using shadow space. This mechanism can be exploited for many purposes: the metadata can hold any per-object data.

## 4.3 FRAMER Implementation

This section describes the current implementation of FRAMER which is largely built using LLVM. Additionally, we discuss how we offer compatibility with existing code.

### 4.3.1 Overview

There are three main parts to our implementation: FRAMER LLVM passes, and the static library (lib), and the binary library in the dashed-lined box in Fig. 5.8. The target C source code and the hooks for FRAMER’s functions in the static library are first compiled to LLVM intermediate representation (IR). Our main transformation pass instruments memory allocation/release, access, or optionally pointer arithmetic in the target code in IR. In general, instrumentation simply inserts a call to library functions, however, our use of header-attached objects and tagged pointers requires further transformations at compile-time. The third part is wrappers around `malloc` family (`malloc`, `calloc`, `realloc`, and `free`) routines and string

functions. Those function are interposed at link time in the current implementation but it is also reasonable to implement customised functions instead of wrapping them to reduce overheads instead of calling subroutines.

Our customised compiler optimisations are discussed in § 4.5.

We also had to modify the LLVM framework slightly. Our main transformation is implemented as a LLVM Link Time Optimisation (LTO) pass for whole program analysis, and runs as an LTO pass on gold linker [74], however, incremental compilation is also possible.

We also insert a prologue that is performed on program startup. The prologue reserves address space for the supplementary metadata table, but pages are only allocated on demand.

### 4.3.2 Memory Allocation Transformations

We instrument memory allocation and deallocation to prepend headers and update metadata by transforming the target IR code at compile time.

#### Stack-allocated Objects (address-taken locals)

For each local allocation of aggregate-type that needs a header, we create a new object with a structure type that contains two fields, one for the header and one for the original allocation as follows:

```
1 struct __attribute__((packed)) newTy {
2   HeaderTy hd;
3   Ty obj; // Ty is an original object's type
4 };
```

We insert a callsite to our hook function that decides if it is small or large-framed, updates metadata in the header, and also in the entry for large-framed objects. It then creates a flag and tag (offset or N value), and moves the pointer to the second field whose type is the actual allocated type by the target program. The hook returns a tagged pointer. The allocation of the original object is removed by FRAMER's pass, after the pass replaces all the pointers to the original object with the tagged pointer to the new object.

We instrument function epilogues to reset entries for large-framed non-static objects. Currently we instrument all the epilogues, and utilise the entry values for detecting some cases of dangling pointers, that will be discussed in § 4.4.2. This instrumentation can be removed for better performance – entries can be simply over-written at the next update to them.

### Statically-allocated objects (address-taken globals)

Creating a new global object with a header attached is straightforward, however, other parts of the implementation are more challenging.

For static/global objects, pointers to them cannot be replaced with a tagged one (i.e. the return value of the hook), since the return value of a function is non-constant, whereas the original pointer may be constant e.g. an initializer of other static/global objects or an operand of `constant expression` (LLVM `ConstExpr`) [71]. The initializer and operands must be constant, hence, replacing with a tagged pointer should be done statically for global objects.

While the tag should be generated at compile-time, the wrapper frame size is determined by their actual addresses in memory, that are known only at run-time. To implement a tagged pointer generated from run-time information at compile-time, FRAMER's transformation pass builds `ConstExpr` of (1) the wrapper frame size  $N$  (2) offset, (3) tag and flag selection depending on its wrapper frame size, (4) pointer arithmetic operation to move the pointer to the second field, and then finally (5) constructs a tagged pointer based on them. The original pointers are replaced with this constant tagged pointer. The concrete value of the tagged pointer is then propagated at run-time, when the memory addresses for the base and bound are assigned.

FRAMER inserts at the entry of the program's main function a call to an initialisation function for each object. This function updates metadata in the header and, for large-framed objects, the address in the table entry, during program initialisation.

**Table 4.1:** FRAMER inserts code, highlighted in gray, for creating a header-padded object, updating metadata and detecting memory corruption. Codes in line 2, 5, and 8 in the first column are transformed to codes in the second column.

	Original C	Instrumented C
1		struct HeaderTy {unsigned size; unsigned type_id;}; struct newTy{HeaderTy hd; <b>int A[10]</b> ;};
2	int A[10];	struct newTy new_A;
3		tagged = handle_alloc(&new_A, A_size); /* tagged = tag   &(new_A->A[0]), A_size = sizeof(int) * 10 */
4	int *p;	int *p;
5	p = A+idx;	p = tagged + idx;
6		check_inframe(tagged, p);
7		untagged_p = check_bounds(p, sizeof(int));
8	*p = val;	*untagged_p = val;

## Heap objects

We interpose calls to `malloc`, `realloc`, and `calloc` at link time with wrapper functions in our binary libraries. The wrappers increase the user-defined size by the header size, call the wrapped function, and perform the required updates and adjustments similar to the hook for stack objects. We also interpose `free` with a wrapper to reset table entries for large-framed objects.

### 4.3.3 Memory Access

FRAMER's transformation pass inserts a call to our bounds checking function right before each `store` and `load`, such that each pointer is examined and its tag stripped-off before being dereferenced. The hook extracts the tag from a pointer, gets the header location, performs the check using metadata in the header, and then returns an untagged pointer after cleaning the tag. The transformation pass replaces a tagged pointer operand of `store/load` with an untagged one to avoid segmentation fault caused by dereferencing it.

Bounds checking and untagging are also performed on `memcpy`, `memmove` and `memset` in similar way. (Note that LLVM overrides the C li-

rary functions to their intrinsic ones [73]). `memmove` and `memcpy` has two pointer operands, so we instrument each argument separately.

As for string functions, we interpose these at link time. Wrapper functions perform checks on their arguments, call wrapped functions with pointers cleared from tags, and then restore the tag for their return value.

### 4.3.4 Interoperability

FRAMER ensures *compatibility* between instrumented modules and regular pointer representation in pre-compiled non-instrumented libraries by stripping-off tagged pointers before passing them to non-instrumented functions. FRAMER adds a header to objects for tracking, but this does not introduce incompatibility, since it does not change the internal memory layout of objects or pointers.

There is another rare case of false positives (we did not encounter them), where library code uses a tagged pointer to read from memory, where our instrumentation did not have a chance to clear the tag as follows:

```
1 struct Node { // Linked list node
2   int data;
3   struct Node* next;
4 };
5
6 // The function length is externally-compiled
7 int length(struct Node * head) {
8   int result = 0;
9   struct Node* current = head;
10  while (current != NULL) {
11    result++;
12    current = current->next;
13  }
14  return result;
15 }
```

At the call to the function `length`, FRAMER's pass loses a chance to tag-clean pointers to instrumented nodes except the pointer `head` passed as an argument. One way to address this is to track memory allocation very

conservatively – performing points-to and the whole program analysis, and instrumenting only objects whose pointers stay inside the instrumented modules. Unfortunately despite the heavy static analysis, many objects will be dropped from tracking. The certain way is to force ignoring the top bits at memory access with hardware support or, with a performance overhead, by a segmentation fault handler.

## 4.4 FRAMER Applications

In this section we discuss how FRAMER can be used for building security applications. We explore mainly spatial safety, but we discuss additional case studies related to temporal safety.

### 4.4.1 Spatial Memory Safety

FRAMER can be used to track individual memory allocations, and store object bounds in the header associated with the object. These bounds can be used at runtime to check memory accesses. Unlike other object-tracking or relative location-based approaches, FRAMER can tackle legitimate pointers outside the object bounds without padding objects, or requiring metadata retrieval or bounds checking at pointer arithmetic operations.

This subsection describes how FRAMER performs bounds checking at run-time.

#### Memory allocation

As described in § 4.3.2, a header is prepended to memory objects (lines 1, 2 in Table 4.1). For spatial safety, this header must hold at least the raw object size, but can hold additional information such as type information. This could be used for additional checks for sub-object bounds violations or type confusion. Its potential in type confusion checking is presented in § 5, and we do not experiment with these in this chapter.

Once we get the header address from a tagged pointer, an object's base address is obtained by adding the header size to the header address. After a new object is allocated, a hook (`handle_alloc`) updates metadata, moves

the pointer to (`new_A->A`), and then tags it (line 3). The pointer to the removed original object is replaced with a tagged one (`A` to `tagged` in line 5).

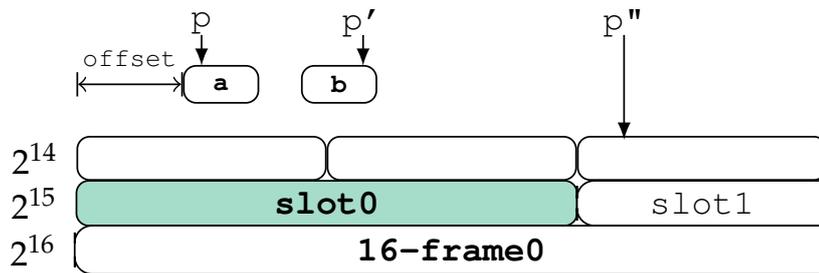
### Pointer arithmetic

As mentioned in § 2.1.1 and Figure 2.2, going out-of-bounds at pointer arithmetic does not violate memory safety, as long as the pointer is not dereferenced. However, skipping checks at pointer arithmetic can lose track of pointers' *intended referents*. Baggy Bounds Checking [6] handles this by marking such pointers during pointer arithmetic and reporting errors only when dereferenced, and J&K [58] pads an object by *off-by-one byte*.

Instead of padding, we include one *imaginary* off-by-one byte (or multiple bytes) when deciding the wrapper frame (see § 4.2.2) on memory allocation. The fake padding then is within the wrapper frame, and pointers to this are still derived to the header, even when they alias another object by pointer arithmetic. The biggest advantage of fake padding is that it is allowed to be overlapped with neighboring objects and thus saves memory. The fake padding does not cause conflicting supplementary table  $N$  values across objects possibly overlapping the bytes.

FRAMER tolerates pointers to the padding at pointer arithmetic, and reports errors on attempts to access them. FRAMER detects those pointers being dereferenced, since bounds checking at memory access retrieves the raw size of the object. Currently FRAMER adds fake padding only in the tail of objects, but it could be also attached at the front to track pointers going under lower bounds, even though such pointers are banned by the C standard.

Beyond utilising fake padding, to make a stronger guarantee for near-zero false negatives, we could perform *in-frame checking* at pointer arithmetic (line 6 in Table 4.1). We can derive the header address of an intended referent, as long as the pointer stays inside its wrapper frame (slot for small-framed), in any circumstance. In Fig. 4.5, consider a pointer (`p`), and its small-framed referent (`a`). Assuming `p` going out-of-bounds to `p'` by pointer arithmetic, `p'` even violates its intended referent, but `p'` is still within `slot0`. Hence, `p'` is derived to `a`'s header by zeroing lower



**Figure 4.5:** By pointer arithmetic, a pointer  $p$  goes out-of-bounds ( $p'$ ), and also violates its intended referent ( $a$  to  $b$ ). FRAMER still can keep track of its referent, since  $p'$  is *in-frame*.  $p''$  is *out-of-frame*, which we catch at pointer arithmetic.

$\log_2(\text{slot\_size})$  (15) bits and adding `offset`. This applies the same for large-framed objects.

Hence, we could check only *out-of-frame* ( $p''$  in Fig. 4.5) by performing simple bit-wise operations (no metadata retrieval) checking if  $p$  and  $p'$  are in its wrapper frame (or slot for small-framed):

```

1 // p: the source pointer of pointer arithmetic
2 // p': the result of pointer arithmetic
3 // N: log2 wrapper_frame_size (or slot_size)
4 is_inframe = (p' ^ p) & (~0ULL << N);
5 assert(is_inframe == 0);

```

FRAMER may report false positives for programs not conforming to the C standard with out-of-frame pointers getting back in-frame by pointer arithmetic without being dereferenced while they are out-of-frame. This is very rare, and those uses will be usually optimised away by the compiler above optimisation level `-O1`. Normally the distance between an object and its wrapper frame's bounds is large. We can also increase the wrapper frame size for all objects to enlarge this distance.

### Memory access

As mentioned in § 4.3.3, we instrument memory access by replacing pointer operands so that the pointers are verified and tag-stripped, before being dereferenced (line 7,8 in Table 4.1).

`check_bounds` first reads a tagged pointer's flag revealing whether the object is small or large-framed. As we described in § 4.2.3 and 4.2.3, we derive the header address from either an offset or an entry, and then get the object's size from the header and its base address as follows:

```
1 obj_base = header_addr + sizeof(HeaderTy);
2 obj_size = ((HeaderTy *)header_addr)->size;
```

We then check both under/overflows ((1) and (2) below, respectively). Detection of underflows is essential for FRAMER to prevent overwrites to the header.

```
assert(untagged_p >= obj_base); // (1)
assert(untagged_p + sizeof(T) - 1 <= upperbound); // (2)
// Where T is the type to be accessed
```

The assertion (2) aims to catch overflows and memory corruption caused by access after unsafe typecast such as the following example:

```
char *p = malloc(10);
int *q = p + 8;
*q = 10; // Memory corruption
```

In a similar fashion, we interpose string functions (`mem*` and `str*`) with our wrappers around them. Handling individual function depends on how each function works. For instance, `strcpy` copies a string `src` up to null-terminated byte, and `src`'s length may not be equal to the array size holding it. As long as the destination array is big enough to hold `src`, it is safe, even if the source array is bigger than the destination array. Hence, we check if the destination size is not smaller than `strlen(src)`, returning the length up to the null byte as follows:

```
assert(dest_array_size <= strlen(src));
```

#### 4.4.2 Temporal Memory Safety

FRAMER may need additional metadata and implementation to provide temporal memory safety enforcement [4, 32, 88, 118] but can still detect some forms of temporal memory errors that we now discuss briefly.

Each large-framed object is mapped to an entry in a division array in the supplementary table, and the entry is mapped to at most one large-framed object for each  $N$ . We make sure an entry is set to `zero` whenever a corresponding object is released. This way, we can detect an attempt to `free` an already deallocated object (i.e. a *double free*), by checking if the entry is `zero`. Access to a deallocated object (i.e. *use-after-free*) is detected in the same way during metadata retrieval for a large-framed object. Note that this cannot detect invalid *temporal* intended referents, i.e., an object is released, a new object mapped to the same entry is allocated, and then a pointer attempts to access the first object.

Detection of dangling pointers for small-framed objects is out-of-scope in the current implementation.

## 4.5 Optimisations

It is very expensive to check every memory access. Support of static analysis and program transformation can reduce overheads, for example, some operations might be redundant such as tag-cleaving on tag-free pointers. We applied both our customised and LLVM built-in optimisations. This subsection describes customised optimisations for FRAMER. Suggestion of further optimisations is provided later in § 4.7.3.

**Implementation Considerations** As described in § 4.3.2, we replaced all occurrences of an original pointer to a global object with a tagged one in constant expression (LLVM `ConstExpr`). Unfortunately, we experienced runtime hotspots due to the propagation of a constant (a global variable's address) to every large `ConstExpr`. To work around this issue we created a *helper* global variable for each global object; assigned the result of the constant propagation to the corresponding helper variable during program initialisation; and then replaced uses of an original pointer with `load` of the helper variable. This way, runtime overheads are reduced, for instance, benchmark `anagram`'s overhead decreased from 14 to 1.7 seconds.

**Non-array Objects** We do not track non-array objects that are not involved with pointer arithmetic, e.g., int-typed objects. It is redundant to perform bounds checking or untagging for pointers to them. We filter out simple cases, easily recognised, from being checked. In the general case, it is not trivial to determine if a pointer is untagged at compile time, since back-tracing the assignment for the pointer requires whole-program static analysis.

**Safe Pointer Arithmetic** Instead of full bounds checks, we only strip off tags for pointers involved in pointer arithmetic and statically proven in-bound for simple cases. For pointers where the bounds can be determined statically, we check if the index is smaller than the number of elements.

In some SPEC benchmarks, there are statically proven out-of-bound accesses, but we do not report memory errors since they may be unreachable. We inserted a termination instruction for this case so that it can report errors at runtime, when the execution reaches the point.

**Hoist Run-time Checks Outside Loops** *Loop-invariant* expressions can be hoisted out of loops, thus improving run-time performance by executing the expression only once rather than at each iteration. We modified SAFE-Code's [34, 119] loop optimisation passes, and added the modified pass to the LLVM LTO pass pipeline, so that it can be run after FRAMER's main transformation pass. We apply hoisting checks to monotonic loops, and pull loop invariants that do not change throughout the loop, and scalars to the pre-header of each loop. This pass works on each loop and if there are inner loops, it handles them first. While iterating our run-time checks inside each loop including inner loops, we determine if the pointer is hoistable. If a pointer is hoistable, we place its scalar evolution expression along with its run-time checks outside the loop, and delete the checks inside loop.

**Inlining Function Calls in the Loop** Inlining functions can improve performance, however it can bring more performance degradation due to the bigger size of the code (runtime checks are called basically at every mem-

**Table 4.2:** Summary averages over all benchmarks (first three columns normalised)

	Memory footprint	Runtime (cycles)	Dynamic instructions	IPC	Load density	D-cache MPKI	Branch density	B-cache MPKI
<b>Baseline</b>	1.00	1.00	1.00	1.70	0.28	24.85	0.19	2.85
<b>Store-only</b>	1.22	1.70	2.24	2.17	0.20	12.27	0.15	1.34
<b>Full check</b>	1.23	3.23	5.25	2.54	0.14	5.28	0.17	0.86

ory access). Currently, we only inline bounds checks that are inside loops to reduce the growth in code size that can be caused by inlining.

## 4.6 Evaluation

We measured the performance of FRAMER on C benchmarks from Olden [20], Ptrdist [10], and SPEC CPU 2006 [49], that are commonly used to evaluate run-time verification for memory safety and compare with other implementations with many object allocations or intensive pointer operations. For each benchmark we measured four binary versions: uninstrumented, only store-checked and full (both load and store checking enabled) on FRAMER, and ASan – one of the most widely used sanitizers. We disabled ASan’s memory leak detection at run-time and halt-on-error to measure overheads in the same setting as FRAMER. In-frame checking 4.5 was not included for evaluation. Binaries were compiled with the regular LLVM-clang version 4.0 at optimisation level `-O2`. Measurements were taken on an Intel<sup>®</sup> Xeon<sup>®</sup> E5-2687W v3 CPU with 132 GB of RAM. Results were gathered using `perf`. Table 4.2 summarises the average of metrics of the baseline and the two instrumented tests.

In this text, cache and branch misses refer to L1 D-cache misses and branch prediction misses both per 1000 instructions (MPKI), respectively.

### 4.6.1 Memory Overhead

Our metadata header was a generous 16 bytes per object. The large-frame array had 48 elements for each 16-frame (division) in use where the element size was 8 bytes to hold full address of the header. The header size

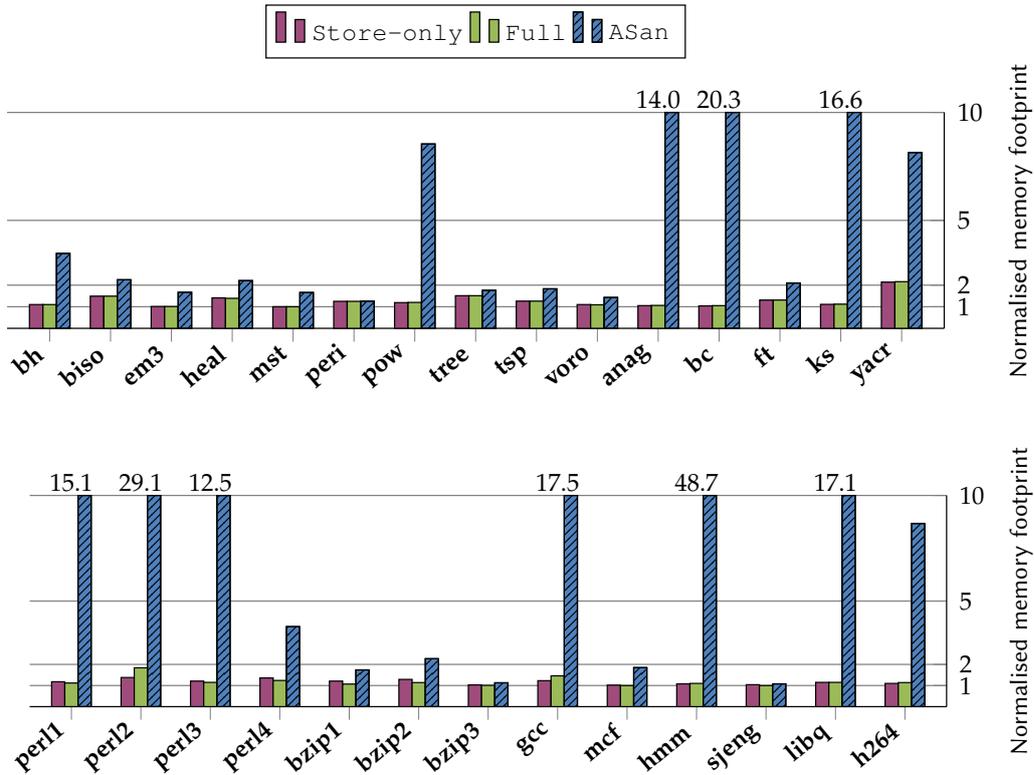


Figure 4.6: Normalised memory footprint (maximum resident set size)

and the number of elements of each division array can be reduced. Currently we mandate 16 alignment for compatibility with the `llvm.memset` intrinsic function that sometimes assumes this alignment. Despite inflation of space using larger-than-needed headers and division array entries and some changes of alignment, we see FRAMER’s space overheads are very low at 1.22 and 1.23 as shown in Fig. 4.6. These measurements reflect code inflation for instrumenting both loads and stores.

The memory overheads of FRAMER are low and stable ASan’s average normalised overheads are 8.84 for the same working set in our experiments, and the highest overhead is 4766% for `hmm`. The average memory overhead of FRAMER is 22% ~ 23% for both store-only and full checking, and only two tests, `perlbench.2` (84%) and `yacr2` (116%) recorded comparably higher growth than other tests. The two tests produce many small-sized objects, for example, `perlbench` allocates many 1-byte-sized heap objects. Currently FRAMER instruments every heap object, so attach-

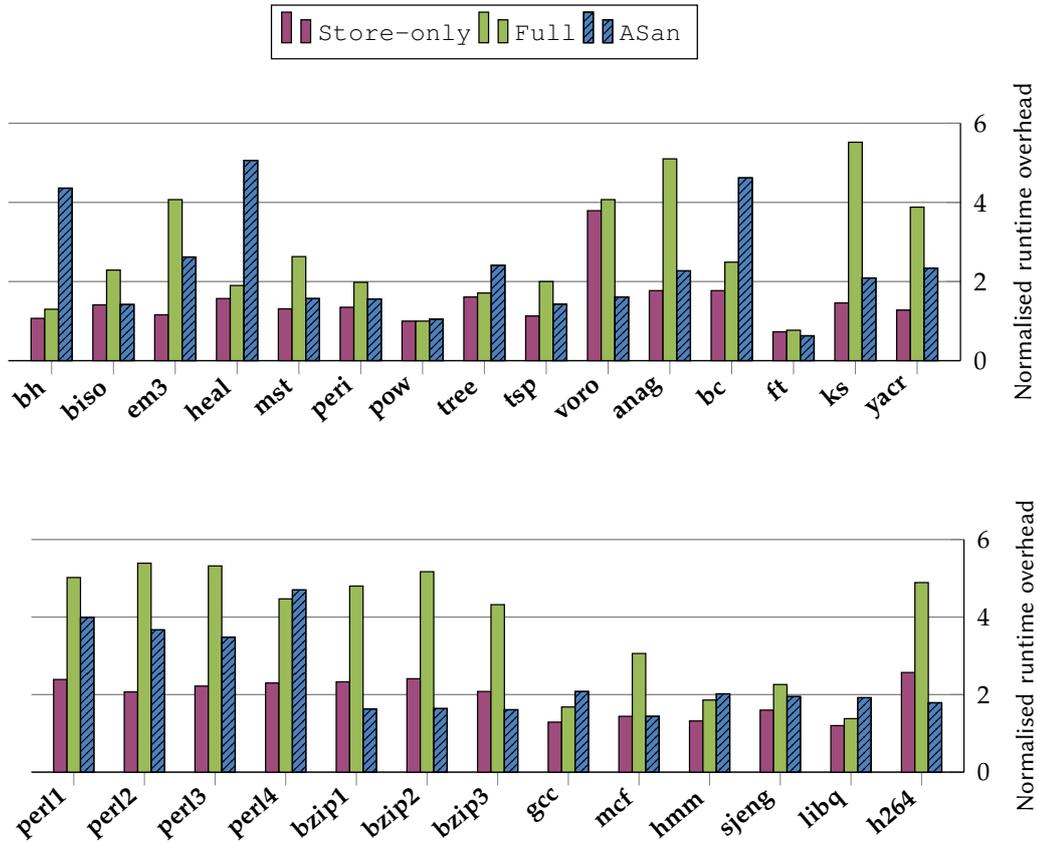
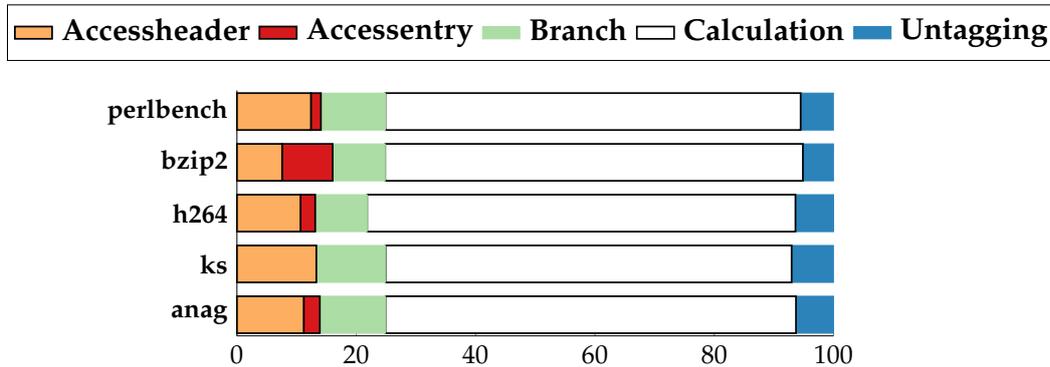


Figure 4.7: Normalised runtime overhead

ing a 16-byte-sized header to all the 1-byte-sized objects made the increase higher. FRAMER’s overheads for those benchmarks are still much lower than ASan’s: 2808% for `perlbench.2` and 714% for `yacr2`.

### 4.6.2 Slowdown

Fig. 4.7 reports the slowdown per benchmark (relative number of additional cycles). The average is 70% for store-only and 223% for full checking. For full-checking, `anagram` (410%) and `ks` (452%) stand out for high overheads despite its smaller program size, mostly due to heavy recursion and excessive allocations causing big growth in executed instructions (674% for `anagram`, 812% for `ks`) as shown in Fig. 4.11, but decreases in cache misses are moderate (76% for `anagram`, 81% for `ks`) compared to average (decreased by 63%). On contrast, `mcf` recorded the highest instruction overheads (1097%), but cache (91%) and branch misses (92%) are



**Figure 4.8:** Runtime overheads for metadata management and retrieval (the overhead for bounds checking excluded)

dropped the biggest among all the tests, so run-time overhead did not grow in proportion to increased instruction count. `perlbench` and `bzip` sets' overheads are high in both FRAMER and ASan. Both tests produce many objects, and especially `bzip` recorded much higher growth in executed instructions than `perlbench` and others.

Performance was impacted far less than would naively be expected from the additional dynamic instruction count (metric columns 2 and 3 in Table 4.2). The rise in IPC (column 4) is quite considerable on average, although the figure varies greatly by benchmark. The original IPC ranged from 0.22 to 3.20 but after instrumentation there was half as much variation.

Our slowdown is mainly due to increased dynamic instructions to calculate metadata location. We measured runtime overheads for metadata management/retrieval of benchmarks with the highest runtime overheads by forcing or preventing inlining of our hooks. As shown in Fig. 4.8, the fluctuations in proportion is negligible. Benchmarks with low runtime overheads showed a similar pattern.

We break down the run-time cost of metadata management/retrieval. The overhead consists of three main contributors:

1. Calculation
2. Memory access
3. Branch

Slowdown is dominated by Calculation (69.66%) – ALU operations to (1) generate a tag at memory allocation and (2) derive the header address from the tag and pointer value at memory access. Hardware acceleration in a future ISA would largely resolve this overhead. We isolated tag-cleaning from Calculation to show the cost of using tagged pointers without hardware support. Its cost (6.07%) would be removed on current ARM that ignores top spare bits. The cost of generating tags was negligible, since it is performed only at allocation. The overhead for bounds checking *after* metadata is retrieved is excluded in the measurement, since the overhead is ALU operations. Including it will increase the proportion of calculation more.

The remaining three components cannot be resolved with simple ISA changes. Branches arise in the following cases:

1. if a pointer is tagged or tag-free at memory access
2. if an object is a small or big-framed (i.e. a flag is true/false)

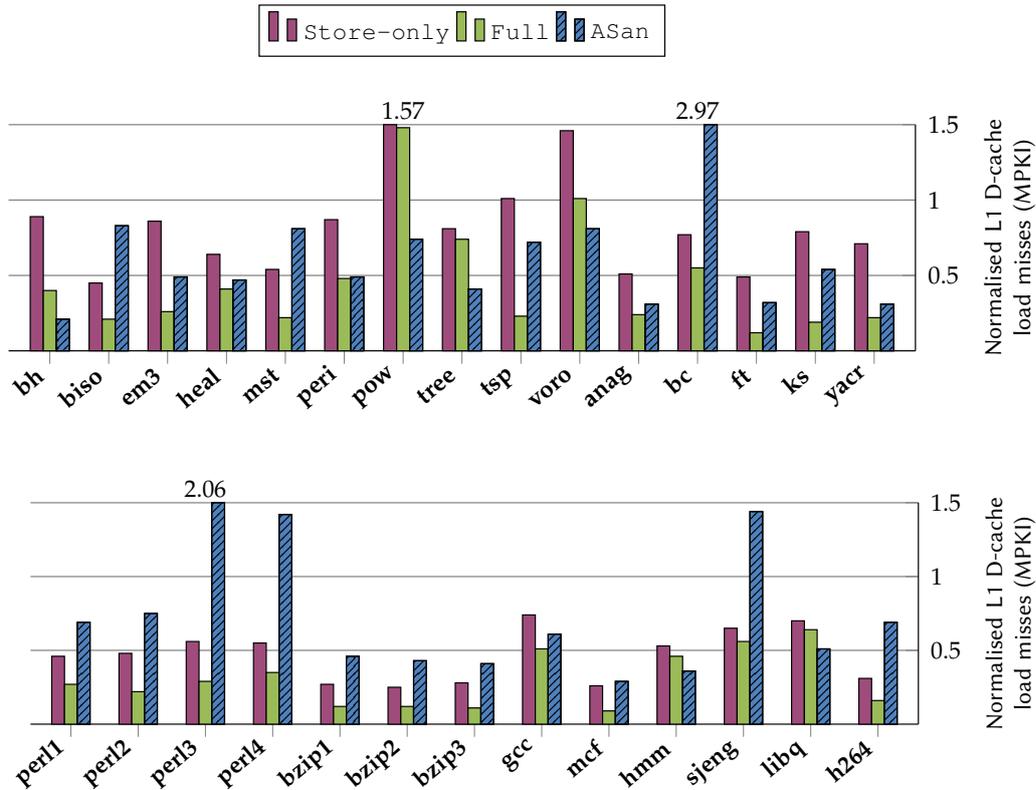
Branch checking for tagged/untagged and small/large-framed contributes 10.19% of the total overheads of metadata management.

Current FRAMER encoding avoided any restriction on object alignment, however, we are open to manipulate memory manager to remove large-framed objects for the future design. `Accessheader` and `Accessentry` represent ratios of overheads to access a header and entry, once their addresses are calculated from tagged pointers. Accessing a header takes more time than accessing an entry, since it is performed on both types of objects. Excluding the overhead of arithmetic operations, the cost is around 25% of that of metadata management and retrieval.

The remaining part of the total runtime overhead that is not included in the measurement shown in Fig. 4.8 is bounds checking performing arithmetic operations with loaded metadata, which can be resolved by ISA.

### 4.6.3 Data Cache Misses

One of the goals of FRAMER is to allow flexible relationships between object and header locality to minimise additional cache misses from metadata

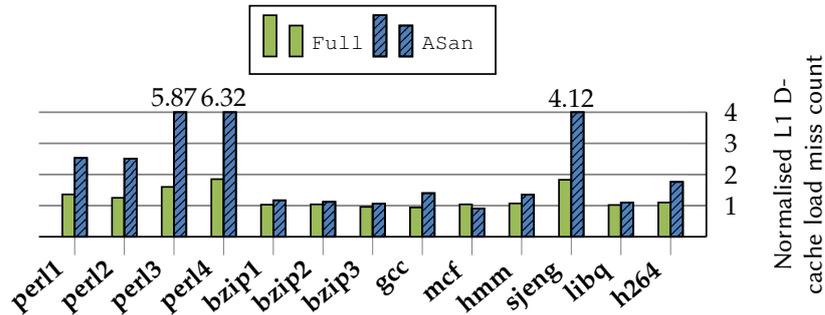


**Figure 4.9:** Normalised L1 D-cache load misses per 1000 instructions (MPKI)

access. We do not analyse L1 instruction cache miss rate since this generally has negligible performance effect on modern processors, despite our slightly inflated code. To explain the measured increase in IPC we analyse L1 D-cache misses MPKI (cache misses) and branch prediction misses MPKI. The baseline D-cache miss rate was 2.48% (Table 4.2) but this improves with FRAMER enabled owing to repeated access to the same cache data.

In Fig. 4.9, we normalise cache misses to the uninstrumented figure. The average normalised cache misses is 0.66 and 0.38 for store-only and full-checking, respectively. The miss rate is reduced since the additional operations we add have high cache affinity which dilutes the underlying miss rate of the application.

While ASan showed increase for four tests. ASan’s normalised misses on average is 0.73, which is higher than FRAMER’s 0.38. ASan’s highest



**Figure 4.10:** Normalised L1 D-cache load miss count

overhead is 197% for `bc`, and two tests reached increase more than by 100%. On FRAMER, `power`'s overhead by 48% is mainly caused by the very low increase in instruction executed in producing MPKI. The misses for the rest of benchmarks decreased, and normalised misses in full-checking mode were below 0.5 for 21 tests among 28 working set, whereas only 13 tests on ASan were lower than 0.5. The overall cache miss rate showed FRAMER is cache-efficient and stable.

Cache misses (MPKI) appear decreased with bloated instruction counts, so we also present the increase in total numbers of cache misses. Fig. 4.10 shows the normalised counts of cache misses for programs in SPEC that are comparably bigger sized than tests in Olden or PtrDist. The averages of shown tests for FRAMER (Full) and ASan are 1.24 and 2.40, and the averages for the whole set are 1.40 and 2.31, respectively. This shows the increase in cache miss count to access metadata in FRAMER is minimal. On FRAMER, the increase rate of all the tests except one (277% for `voronoi`) are below 100%. On ASan, the increases for 7 tests are above 100%, and `bc`'s increase rate is 1160%.

#### 4.6.4 Instructions Executed

Fig. 4.11 reports normalised overheads per benchmark. FRAMER increases dynamic instruction count by 124% for store-only, and 425% for full checking. This increase is the main contributor to slowdown. Dynamic instruction penalty arises from setting up and using tagged pointers. The major source of the growth is arithmetic operations. As shown in Fig. 4.8, 75%

of runtime overhead of metadata management/retrieval is dominated by calculation of (1) the header address at memory access and (2) the tag at allocation. This cost can be resolved with hardware acceleration in the future ISAs.

The penalty of utilising top bits is *over-instrumentation* – unless individual memory access is proven tag-free statically, we have to instrument it (i.e. tag-cleaning) to avoid segmentation fault in all major architectures, requiring the top bits to be zero (or special pointer authentication code in ARM8). This results in stripping the tag field for untagged pointers.

The average overhead for ASan is 226%, which is lower than FRAMER. The average excluding the highest test (1336% for `bh`) is 184%, while FRAMER’s average excluding the highest (1098% for `mcf`) is 400%. The difference in slowdown on average (FRAMER: 213%, ASan: 139%) was not big as the difference of instruction executed due to FRAMER’s cache efficiency. ASan consumes fewer dynamic instructions, since shadow space-only metadata storage requires simpler derivation of metadata location, taking advantage of re-alignment of objects, as trade-off of space and high locality.

Future implementations can optimise the case where conservative analysis reveals the tag never needs to be added. More discussion on optimisation is described in § 4.7.3.

#### 4.6.5 Branch Misses

Additional conditional branches arise in FRAMER from checking whether small or large frame is used and in the pointer validity checks themselves. Many approaches using shadow space are relieved from these branches at metadata retrieval.

As shown in Table 4.2 col 7, the dynamic branch density decreases slightly under FRAMER instrumentation, but the branch mis-prediction rate greatly decreases (col 8). The averages of normalised branch misses for store-only and full-checking are 0.62 and 0.42, respectively. This shows the additional branches achieve highly accurate branch prediction and that branch predictors are not being overloaded. Of the new branches added, the ones checking small/large frame size are completely statically pre-

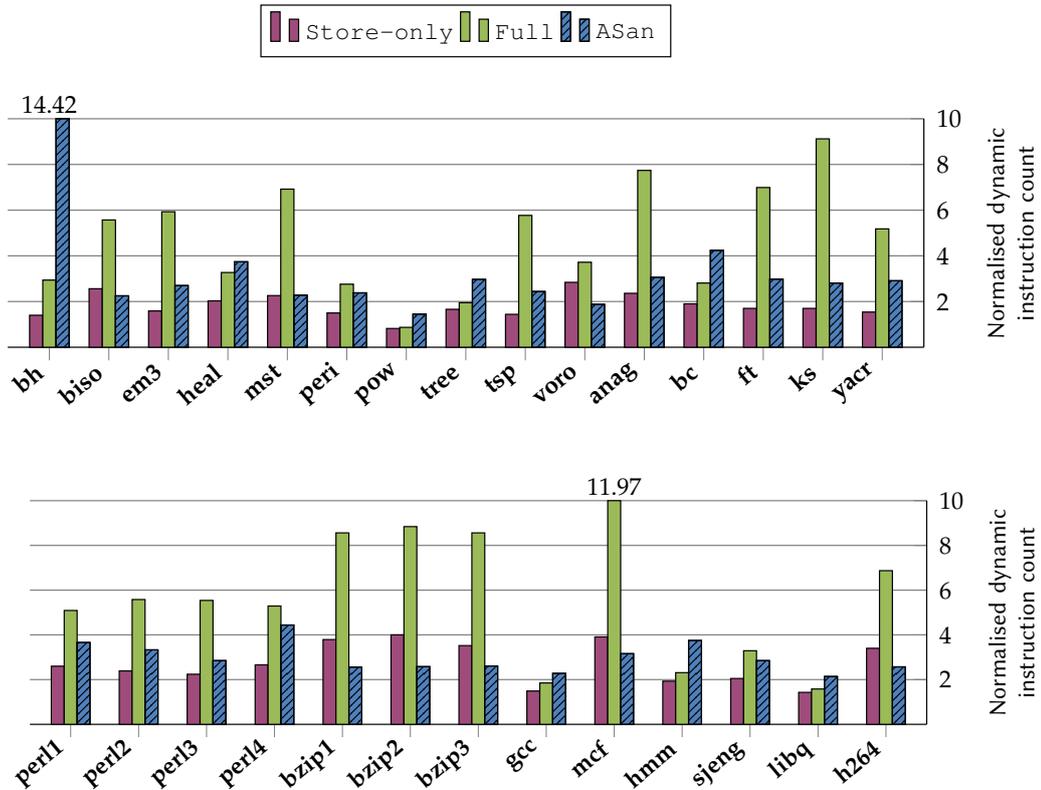


Figure 4.11: Normalised dynamic instruction count

dictable owing to the checking code instances being associated with a given object. And the ones checking pointer validity also predict perfectly since no out-of-bounds errors are detected in normal operation.

## 4.7 Discussion

### 4.7.1 Comparison with Other Approaches

#### Shadow space-based approaches

Shadow space-based approaches reduce slowdown by lowering executed instructions. Trade-off of data memory is tolerable in most systems during development. For practical deployment, however, their slowdown is still high and memory footprint is critical in some systems, e.g. small computer running in an embedded system or I/O-heavy server-side loads. In using shadow space, it is inevitable to pad and re-align objects to avoid

conflicts in entries [114, 6, 68, 47]. ASan pads each object for wider detection coverage and more padding for alignment, which burdens space, whereas FRAMER's fake padding and wrapper frames do not consume any space. Furthermore, their higher cache misses to access metadata in a remote memory region (including ASan's resetting entries at deallocation), making its runtime overheads unpredictable.

In comparison with ASan, FRAMER showed better efficiency both in memory footprint (FRAMER: 23%, ASan: 784%) and cache miss counts overhead (FRAMER: 40%, ASan: 131%). ASan showed lower increase in runtime overheads (FRAMER: 223%, ASan: 139%), however, 75% of FRAMER's overhead of metadata management and retrieval is consumed for calculation, that can be largely resolved with new ISA. The rest of overhead comes from bounds checking using loaded metadata, that can be also implemented as ISA.

### **SGXBounds**

SGXBounds spares 32 bits for a tag among 64 bits, while FRAMER tags only upper spare 16 bits. SGXBounds's retrieving an upper bound first, not the base like FRAMER, may save some overheads if we perform overflow-only checking. However, using a footer makes systems slightly more vulnerable to metadata pollution without complete memory safety. For both over/under underflow checking, we do not consider our derivation of the base, not the upper bound, as a weakness. In addition, frame encoding can be easily integrated to SGXBounds' design.

### **MPX**

Intel MPX [53, 87, 99] provides a hardware-accelerated pointer-based checking instrumented by compiler. In principle, FRAMER could utilize MPX extensions for performance when used for spatial safety. We showed FRAMER is more cache-friendly, but it could be made even faster if a single instruction implemented the complete tag decode operation, splitting apart the tagged pointer into an untagged object pointer and separate header pointer in another register. This would be a fairly simple, register-to-register instruction, operating on general purpose registers. Since this

has not used the D-cache, an enhancement would be to compare the pointer against a bounds limit at hardcoded offset loaded from the header, but the best design requires further study.

### 4.7.2 Hardware Implementation of FRAMER

We believe FRAMER's encoding is at its best when it is implemented as instruction set extensions. As mentioned in §4.7.1, the increase in the number of executed instructions for calculation, the main contributor to slowdown of FRAMER, can be resolved with new instructions. Tag-cleaning can be supported by hardware [9]. Moreover, generating a tag and deriving a metadata address can be implemented as a single operation, respectively.

### 4.7.3 Additional Optimisations

#### Utilising More Spare Bits

Currently, we mandate 16-alignment due to `llvm.memset` intrinsic function. On this alignment, we have spare 4 bits at the end of offset for small-framed and another 4 bits in the pointer. (We already have spare bits for large-framed ones.) Using the bits, we can perform bounds checking only at pointer arithmetic and mark out-of-bounds pointers, so that we can report errors when they are dereferenced. This way, we expect to remove duplicated runtime checks, since the pointer may be used for memory access multiple times. Above this, we can utilise them to encode more information for better performance.

#### Compiler Optimisation

Redundant runtime checks can be eliminated using *dominator trees*. Soft-Bound [89] reported that their simple dominator-based redundant check elimination improved performance by 13% and claimed more advanced elimination [13, 142] can reduce more overheads.

The penalty of using tagged pointers is that unless individual memory access is proven safe at compile time, we may have to *over-instrument* memory access to avoid segmentations faults. Some approaches can save ex-

pensive runtime checks to reduce performance degradation, bearing false negatives, but it is difficult in approaches using tagged pointer. We did not run dedicated pointer-analysis for this version but it can remove over-instrumentation. Loop optimisation did not show a large impact on reducing overheads, even for some SPEC benchmarks whose number of hoisted run-time checks reached hundreds at static time. Our naive optimisation skipping untagging improved performance more than state-of-the-art loop hoist pass. *Static points-to analysis* [121, 123], as long as it does not assume the absence of memory errors, potentially enables many tags and bounds checks to be removed at compile time.

## 4.8 Conclusion

This chapter presented FRAMER, a per-object capability system utilising the currently unused significant bits of pointers to store a tag. A key insight is that this tag can be bifurcated using a flag bit so that the overwhelmingly common case of *small-framed* objects can be dealt with efficiently in terms of both time and space. This ultimately benefits the performance of exceptional *large-framed* objects too, because the design can special-case them as well.

FRAMER is evaluated with a case study on spatial memory safety in C programs. However, we believe its capability design could benefit the performance of other programming language security mechanisms as well. Compared to existing approaches, *frame*-based offset encoding is more flexible both in metadata association and memory management, while still offering a fairly simple calculation to map from arbitrary pointers to metadata locations. In addition, its intrinsic memory and cache-efficiency make it potentially attractive for direct hardware support.

# 5

## SpaceMiu: Practical Type Safety for C

---

### 5.1 Overview

There are two major types of *spatial* memory errors. The first type of spatial memory errors is widely known as buffer overflows (§ 2.1.1). Another spatial memory error is *type confusion* errors where the program accesses a memory location using an *incompatible* type (§ 2.2). This chapter focuses on type confusion vulnerabilities in C.

Type conversions in C are frequently used. Attempts to access a memory location using a type violating the contract on the memory resource can cause memory corruption. Figure 5.1 shows one example of unsafe but widely used type conversion. A pointer `p` (line 6) is assumed to point to `SubTy`-typed data in memory, but it is risky: memory in the location may hold data in a different type. For instance, if `Ty`-typed data (line 2) is stored in the location, `sub` goes out-of-bounds (line 7). To prevent this, we need to find the data type, that `p` points to, in memory, and then check if casting from the data type to `SubTy` (line 6) is *safe*.

In addition, it is challenging to formulate type hierarchy in C language that heavily weakens and restores types on objects. In Figure 5.1, most approaches would treat `SubTy` as `Ty`'s child (i.e. subtype), and consider typecasting from `SubTy` into `Ty` *safe*. As for casting from `Ty` to `Y`, the policies differ depending on their view. `Ty` can be treated as `Y`'s subtype, since `Ty` has the same memory layout as `{int; char[5]; char[5];}`, which

```
1 struct Y {int fval; char ch[5];};
2 struct Ty {int ival; char name[10];};
3 struct SubTy {struct Ty x; struct Y * yp;};
4
5 struct Ty * foo (void * p){
6     struct SubTy* obj= (struct SubTy*)p;
7     struct Y* sub= obj->yp;
8     // do something with sub
9     ...
10    return (struct Ty*)sub;
11 }
```

**Figure 5.1:** C example with typecasts

can be interpreted as `{struct Y; char[5];}`. Our approach allows this cast, while some other approaches [60] do not. There is a need for balancing tightness of type rules to minimise false positives/negatives, but still detect memory corruptions via type errors without restricting programmers' freedom to manipulate pointers.

These memory corruptions are also frequently associated with *unions* when parsing data with many different embedded object types in C. This can trigger security consequences such as out-of-bounds read, code execution caused by size inconsistency or improperly-parsed file containing records of different types.

Several approaches [46, 60, 93, 34, 19, 35, 116] have been proposed to prevent memory corruptions through unsafe typecast. Some of them [15, 31, 124, 149] are based on vtable pointers, avoiding high overhead of tracking per-object (or pointer) that has been the bottleneck of practical runtime verification systems. However the approaches usually perform only type checking between polymorphic classes so cannot be directly used for type correctness for C programs.

Some other approaches [60, 46, 93, 57, 35] are based on per-object/pointer metadata. They can handle a wider range of type errors by tracking live objects or pointers without breaking binary compatibility (except fat pointers [11, 57, 93]) however their disadvantage is performance degradation to manage per-pointer/object type information at run time, making efficient metadata management for *pointer-to-type* mapping the key

to make those solutions practical. In addition, some of them handle only C++ with class hierarchy.

In this chapter, we present `spaceMiu`, a run-time type confusion checker for C programs. Inspired by `CCured` [93], `spaceMiu` formulates types including *unions* and type hierarchy in C mimicking up/downcast. Based on the type relation, we implement a run-time type confusion checker using our per-object type metadata management and two type descriptors holding per-type physical memory layout and type relation, respectively.

Our key contributions are as follows:

1. Formulation of type relation and the validity of typecast in C
2. Efficient and scalable per-object information management
3. Evaluation on practical tests

## 5.2 Types and Type Relations

The first challenge we address is to define what type conversion is *safe* in C programs. Many C programs heavily weaken types e.g. at function call, pointers typecast to a different type (often to `void*`) from the contracted type (*upcast*) and accessed with their original type (desirably) (*downcast*), which mimics type hierarchy. We define sub-typing in C based on physical memory layout in this section, and discuss run-time type confusion checking in the next section (§5.3).

Technically, C does not support type hierarchy unlike higher-level languages like C++, C# or Java. Even so, C programmers use structure types that mimic upcasts and downcasts. Inspired by `CCured` [93]’s *physical equality* and *physical sub-typing* [116, 22], we define concrete types and their relation, and adapt them to types supported by LLVM/clang for the C language.

### 5.2.1 Type Representation

Firstly, *atomic types* are a subset of *first-class* types, whose values are the only ones which can be produced by LLVM IR instructions [72], and that can show up anywhere themselves, not just as instances of types. That

include integers, floating points, pointers, vectors, and register types such as `x86_mmx`. We define atomic types as follows, where  $n$  and  $m$  are non-negative integers and `atomic*` represents a pointer type:

$$\begin{aligned} n &:= 1 \mid 8 \mid 16 \mid 32 \mid 48 \mid 64 \mid 128 \\ m &:= 16 \mid 32 \mid 64 \mid 128 \mid 80 \\ \text{atomic} &:= \text{int}_n \mid \text{fp}_m \mid \text{vector}_k \mid \text{x86\_mmx} \mid \text{atomic}^* \end{aligned}$$

**Figure 5.2:** Atomic Types

A vector with an element count  $k$ <sup>1</sup> e.g. `<4 x int32>` is treated as a first class type since it is commonly used when multiple other primitive data (atomic-typed data) are operated in parallel using a single instruction, for instance, SIMD.

An *aggregated type* normally refers to an ordered collection of other types – arrays or structures. We represent it as a list of atomic types and the list is constructed by using a function  $\pi$  that flattens a type using a constructor of an empty list (`nil`) and of non-empty list (`::`), and append operator  $\sqcup$ . An atomic type is a list of itself and an aggregated type is a list of unrolled fields/elements as follows:

$$\begin{aligned} \pi(\text{void}) &= \text{nil} \\ \pi(\text{atomic}) &= \text{Atomic} :: \text{nil} \\ \pi(\text{struct}\{\tau_1 f_1; \dots; \tau_n f_n\}) &= \pi(\tau_1) \sqcup \dots \sqcup \pi(\tau_n) \\ \pi(\tau[n]) &= \pi(\tau) \sqcup \dots \sqcup \pi(\tau) \text{ (} n \text{ times)} \end{aligned}$$

**Figure 5.3:** Aggregated Types

## 5.2.2 Physical Equality

We define type hierarchy using *physical equality* relation between types in a list form. Informally speaking, two types  $\tau$  and  $\tau'$  are *physically equal*, denoted by  $\tau \approx \tau'$ , when their memory layouts are identical. More precisely, two non-pointer atomic types are *physically equal* if both sizes and alignments are identical as presented in rules (1) ~ (4) in Figure 5.4. This work assumes that an atomic type's size and alignment are the same and the assumption holds in most C standards. Physical equality of aggregated types

<sup>1</sup> $k$  is a non-negative constant integer.

is defined based on lists of atomic types: two aggregate types are physically equal if their  $i_{th}$  atomic types are physically equal for all  $i$ , as shown in the rule (5) in Figure 5.4. Pointer types are equal if their reference types are equal as shown in the rule (6).

$$\begin{array}{c}
 \frac{}{nil \approx nil} \quad (1) \\
 \\
 \frac{m = n}{int_m \approx int_n} \quad (2) \qquad \frac{m = n}{fp_m \approx fp_n} \quad (3) \\
 \\
 \frac{m = n}{fp_m \approx int_n} \quad (4) \qquad \frac{\alpha \approx \alpha' \quad \tau \approx \tau'}{\alpha :: \tau \approx \alpha' :: \tau'} \quad (5) \\
 \\
 \frac{\tau_1 \approx \tau_2}{\tau_1^* \approx \tau_2^*} \quad (6)
 \end{array}$$

**Figure 5.4:** Physical Equality

Currently we tolerate type conversion between  $int_{64}$  and a pointer type in 64-bit machines, since the conversion is frequently used, and applying strict rules to them generated too many errors in our experiments e.g. `perlbench`.

### 5.2.3 Physical Sub-typing and Type Relation

Type hierarchy is defined based on our type relation between two types, called *Physical sub-typing* denoted by  $\preceq$ . We show  $\tau \preceq \tau'$  by proving that  $\tau$  is physically equal to  $\tau'$  appended with  $\tau''$ , that can be either an empty or non-empty list of atomic types as follows:

$$\tau' \preceq \tau \iff \exists \tau''. \tau' \approx \tau \sqcup \tau''$$

When  $\tau \preceq \tau'$  holds, typecast from  $\tau'$  to  $\tau$  is called *upcast*, which is considered safe, and denoted by  $\tau \longmapsto \tau'$ . Their pointer cast  $\tau^* \longmapsto \tau'^*$  is also safe upcast as presented in the following:

$$\frac{\tau' \preceq \tau}{\tau'^* \longmapsto \tau^*}$$

On contract, downcast is typecast from  $\tau$  to  $\tau'$  that may cause memory corruptions.

A common use of up/downcast is temporarily weakening an arbitrary pointer to `void*` given to a callback or function argument, and then casting back to its original type. A `void` pointer holds an arbitrary address and any pointer can be upcast to a `void` pointer without loss of information [61]. `void` is represented as an empty list, making it an ancestor of any types. The validity of typecast from an arbitrary pointer ( $\tau^*$  p) to `void*` can be derived with rules as shown below:

$$\frac{\frac{\frac{\tau \approx \tau}{\tau \approx nil :: \tau}}{\tau \preceq void}}{\tau^* \mapsto void^*}$$

Upcast pointers to `void*` passed inter-procedurally are normally converted back to their original type i.e. *downcast* via explicit cast. Unfortunately they may downcast to another type, which requires run-time checking.

In this notion of flattening types to the level of atomics, we may lose structural information of aggregate types e.g. nested structures or arrays in structure type. CCured interprets them as arrays of atomic types and make the type compatibility more relaxed than libcrunch [59] that breaks an aggregate type down to its immediate sub-types, not to the level of atomics. In many cases, libcrunch’s sub-typing makes more sense but we noticed LLVM converts contiguous atomics in an aggregate type to a vector type, so in this approach we follow CCured’s sub-typing.

### 5.3 Run-Time Typecast Checking

Upcast is considered safe and this can be verified at compile time: if a target type of typecast is an ancestor type of the source type, the target type at runtime is also an ancestor type of the source type.

In contrast, a target type of downcast may mismatch its data type of a pointer: if a pointer type is a sub-type of a target type at typecast, access with the pointer after downcast may cause overflows including internal overflows, hence downcasts require run-time checking to prevent this type confusion.

Downcast checking is more challenging than upcast checking, since the run-time type of a pointer is unknown at compile-time. It requires *pointer-to-type* mapping. First of all, we need to track individual objects (or pointers) and store per-object (pointer) type information in the database. We then should map a pointer at a unique offset in the object to a sub-object (field) corresponding to a nested type.

In summary, we need the following three pieces of information to map a pointer to its corresponding type at runtime:

1. a pointer's referent object type
2. an offset that the pointer references in the object
3. type information at the offset

First of all, to manage per-allocation types, spaceMiu leverages FRAMER's *per-object metadata* storage (§ 4.2): each memory object holds its object type information in a header attached to itself, and an object pointer is derived to the header location, as described in § 4.2.3. spaceMiu collects all the types used in a program and assigns a unique ID to each type at compile time. The current implementation of our main transformation as a Link Time Optimization (LTO) pass makes it easier to collect all used types of the whole program. Each header holds an object's type ID. In addition to the object type, FRAMER's tagged-pointer encoding allows us to get an object's base address at the same time.

Once obtaining both the type ID and base address of a referent object, we calculate an offset by subtracting the base address from an object pointer, and then get a type at the offset using *type layout descriptor*. Each entry of the type layout descriptor holds memory layout information of each type in the form of a list of types at each offset. We retrieve the object type's memory layout with the type ID as an index of the descriptor, and get a nested type with the offset. Pointers are usually mapped to a structure's field or array's element; otherwise we report them as an error.

spaceMiu then examines the validity of type conversion from a type at the offset ( $\tau$ ) to a target type using a *type relation descriptor*. If a target type is  $\tau$ 's descendent type (physically not equal to and downcast of  $\tau$ ) or totally incompatible, spaceMiu reports a type confusion error.

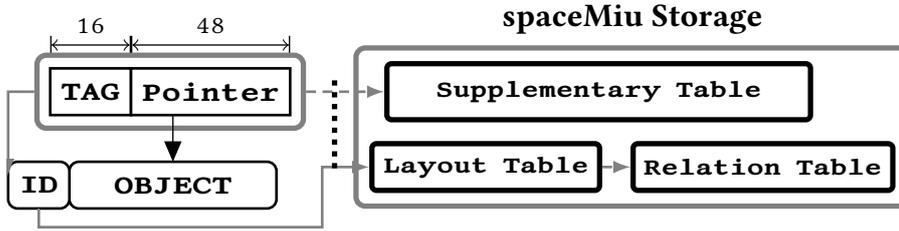


Figure 5.5: Metadata Storage and Type Descriptors

This type confusion checker can be run with other security enforcements on the shared metadata management such as spatial memory safety (§4.4), although we did not evaluate in that aspect in this study.

In the following subsections, we describe (1) object-to-type mapping using per-object metadata management and (2) pointer-to-type mapping and typecast checking using two type descriptors.

### 5.3.1 Object-to-type Mapping

Allocated objects in memory are accessed with a known *type*. The common three cases are (1) where an object’s type is determined at allocation and an object is written and read with their declared type. In contrast, there are cases (2) where an object may be associated with its type *after* allocation with delay or (3) where an object is accessed with a different type from its initial type. We discuss *what type* is associated with an object and *when* to map an object to its type in each case.

(1) The first case is where a memory block is associated with its type at allocation and holds data in the type. Most static/stack/global objects fall in this category. We call this type an object’s *declared type*. Those objects may be accessed in either their declared type or different type. Different types are either compatible or incompatible with the declared type.

(2) The second case is heap allocation via `malloc` family. A heap object does not have a declared type initially unlike static/global/stack objects. Instead, it is given its *effective type* when its pointer is typecast to a known type at memory write. We categorise heap allocations using `malloc` into three as follows:

- (a)  $\tau * p = \text{malloc}(\text{sizeof}(\tau))$

(b)  `$\tau^*$  p = malloc (sizeof( $\tau$ )  $\times$  n)`

(c) `void * p= malloc (n)`

Heap allocations (a) and (b) present common forms of allocation with `malloc` or customised wrapper functions whose effective type is specified by programmers. `spaceMiu` associates these objects with its effective type at the *first typecast* from `void*` to  $\tau^*$ . Objects of the case (a) as a  $\tau$ -typed singleton and those with (b) as a  $\tau$ -typed array. Although dynamically-allocated objects are allowed to change their effective type, we assume that objects in the cases (a) and (b) have one effective type during their lifetime, and this agrees with most programmers' habit.

Heap allocation (c) represents a *byte array* whose effective type is determined at memory write through string functions such as `memcpy`. Their effective type is usually `string` i.e. a `char`-typed array. Some byte arrays may keep being overwritten, being used as a buffer. We treat them as a buffer that is not associated with any particular effective type, instead of updating the type. More discussion on this implementation will be presented in § 5.5.6.

(3) The last case is where objects are given either their declared type at allocation or effective type at the first typecast, but may hold data in another type. Unions fall in this category. Stored data is usually in one of the field types of a union and it is read in the same type as the written type. However the data can be also read with another type that is not one of the fields of unions, that requires type checking at run time.

Unions may seem similar to heap objects in the sense that the data type is determined *after* allocation, however our approach will treat them differently. They may not hold data in the same type as its declared (or effective) type, which is union, and the memory may keep being overwritten with one of the field types, whereas most heap objects except byte arrays are assumed to hold data with only one effective type while they are alive.

We separate a union's stored data type from its first-associated type (declared or effective), which is a union, and call the stored data type a *run-time type* (RTT). `spaceMiu`'s per-object metadata holds both type information as shown below:

```
struct HeaderT {
```

```

int TID;          /* Type ID of an element */
unsigned size;   /* the raw obj size */
unsigned elemsize; /* an element size for an array
                  otherwise equal to size */
short isUnionTy; /* if yes 1, otherwise 0 */
short RTTunion;  /* Type ID of stored data in union */
};

```

For unions, spaceMiu stores the type ID of union itself in `TID` and the stored data (field) type ID in `RTTunion`. `isUnionTy` indicates if the object is union or not. For non-unions, `RTTunion` field is not used. Currently spaceMiu does not handle arrays, so the header form should be modified to cover arrays. More details on handling unions will be presented in § 5.4.

In summary, we store an object's declared (or effective) type ID and RTT ID for unions in an object's metadata.

### 5.3.2 Per-object Metadata Management and Pointer-to-Type Mapping

We leverage per-object metadata management of FRAMER presented in § 4.2. The metadata in the header holds an object's type information – an object type ID and RTT ID.

In addition to per-object type information, run-time type confusion checking requires an offset due to operations on pointers such as pointer arithmetic prior to typecasts to be verified. spaceMiu now performs *pointer-to-type mapping* that obtains an offset from the base of an object to a pointer reference and then maps a pointer at a unique offset to a sub-object's type at the offset by utilising a *type layout table*.

It is straightforward to get an offset in a structure-typed object is in this approach – simply subtracting an object base from a tag-cleaned pointer.

Now we utilise the type layout descriptor to map the offset in the object to its corresponding type. Once the object type ID is retrieved from a header, we directly access a corresponding entry holding *per-type physical layout* information in the descriptor with the object type ID as an index.

Each per-type layout information is in the array form, and each element of the array holds a type ID at a unique offset. Per-type layout contains a

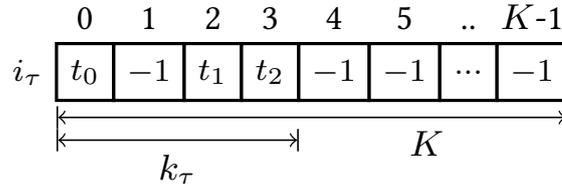
*flattened* list of type IDs at each offset as shown in Fig. 5.6a. All the entries (i.e. arrays) in the layout descriptor are fixed-sized to the last meaningful offset of a type having the maximum offset, so the element holding the ID of the type at the offset is also directly accessed with the calculated offset (we filled a negative integer (-1) in the elements that are not mapped to any type). For instance, the following structure `VERTEX` has a sub-structure `VEC` as a field:

```
struct VEC {
    double a;
    double b;
};
struct VERTEX {
    struct VEC a;
    struct VERTEX* b;
    struct VERTEX* c;
};
```

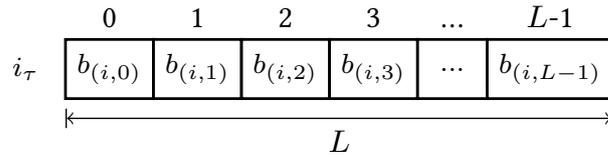
A list for `struct VERTEX` is obtained by flattening a sub-structure (`struct VEC`) as follows, where `max` is the maximum offset amongst all the layouts in a program:

```
{VECid, -1, -1, -1, -1, -1 -1, -1,
doubleid, -1, -1, -1, -1, -1 -1, -1,
VERTEX*id, -1, -1, -1, -1, -1 -1, -1,
VERTEX*id, -1, -1, -1, -1, -1 -1, -1,
... up to max}
```

Here, one offset may be mapped to multiple types in the flattened layout e.g. a composite-typed field (`VEC`) and its first field `double` at the offset 0 in the flattened layout of `struct VERTEX`. For those cases, we choose the highest type over sub-objects' type at the offset, since the type holds its sub-object information. For instance, at an offset 0 of `struct VERTEX`, a type ID of `struct VEC` is stored, not the ID of `double`.



(a) Entry of Layout Table:  $K$  is the maximum *meaningful* offset among all identified types in a target program, and  $k_\tau$  is the last meaningful offset of a type  $\tau$  with the type ID  $i_\tau$ .



(b) Entry of Relation Table: the table  $R$  is implemented as an  $L \times L$  matrix, where  $L$  is the number of all identified types in a target program. Each entry ( $R[i][j]$ ) holds a boolean value ( $b_{(i,j)}$ ) representing if typecast from type  $\tau$  with ID  $i$  to  $\tau'$  with ID  $j$  is safe or not.

**Figure 5.6:** Entries of Type Descriptors

### 5.3.3 Type Confusion Checking

Once a pointer is mapped to a type ( $\tau$ ), we check if type conversion from  $\tau$  to a target type ( $\tau'$ ) is safe i.e. if  $\tau'$  is physically equal to or upcast of  $\tau$ .

Now we utilise another type descriptor – *type relation table*. The relation table  $R$  is implemented as an  $L \times L$  boolean matrix, where  $L$  is the number of all used types in a target program. Each entry (row) corresponding to one type  $\tau$  holds relations between  $\tau$  and all types. Given two type IDs,  $i$  (source type  $\tau$ ) and  $j$  (target type  $\tau'$ ), we directly access the corresponding element in the table with them. A boolean value ( $R[i][j]$ ) represents if typecast from  $\tau$  to  $\tau'$  is safe, as shown in Fig. 5.6b. If  $R[i][j]$  is equals to false, spaceMiu reports a type confusion error.

For instance, spaceMiu found the following error at typecast from struct op\* to struct binop\* at offset 48 in the benchmark perl-bench:

```
// The syntax, iN, specifies an N-bit integer.
```

```
struct op      {struct.op*; struct.op*; struct.op* (*)*;
                i64; i16; i16; i8; i8};
```

```
struct.binop {struct.op*; struct.op*; struct.op* (*)*;
             i64; i16; i16; i8; i8;
             struct.op*; struct.op*};
```

## 5.4 Union Type

As briefly mentioned in § 5.3.1, unions are handled differently from other types, since their declared (or effective) type differs from its RTT. In this section, we discuss type safety for unions.

LLVM-clang does not support unions. What happens in practice is that clang converts a union into a structure. In this text, we call a structure, generated from a union by LLVM-clang, a *union-structure*. LLVM-clang still lets the cases recognisable by naming them with a prefix "union.". Consider the following union:

```
union Data {
    int a;
    int * b;
    char c[20];
};
```

LLVM-clang converts the union into the following union-structure:

```
// We present the code in C, not in LLVM IR, for explanation.
struct union.Data {
    int * b;
    char c[20];
};
```

During the process, clang may drop some fields (`int a` in this example) or re-order fields. At memory access to the dropped field, clang generates typecast of a pointer to the union-structure into the field type.

In LLVM-clang, a union is treated as a piece of memory that is accessed using *implicit pointer casts*. It seems similar to a byte array, however there is a difference: LLVM-clang does not simply create a union-structure having a byte array which is big enough to contain any user-specified union field. It creates a union-structure with the right size that meets *alignment requirement*. In the example above, LLVM-clang pulls the most aligned field (`int`

\* b) to the front, so that the union-structure is aligned by it. It then appends `char c[20]` to make the union-structure large enough to hold the member with the largest size, and discards the redundant member (`int a`). It considers both size and alignment to determine *what fields* are to be kept and *in what order* chosen fields should be. At memory access to the field `int a`, LLVM-clang performs a cast operation which converts a union-structure pointer (`struct union.Data`) into `int` pointer type.

Sometimes LLVM-Clang modifies fields.

```
union Data2 {
    char ch[41];
    int b[10];
};
```

The union (`Data2`) is compiled to the following LLVM-IR code:

```
%union.Data2 = type { [10 x i32], [4 x i8] }
```

`int b[10]` becomes the first member, since it is the most aligned member of the union, and the shortened `i8`-typed array (the `char` array) is appended to make it the right size.

That is, a union-structure does not hold information of fields that a union initially has. Hence, given a cast operation from a union-structure into a certain type in LLVM IR, we cannot distinguish which case the typecast is:

1. generated by LLVM-clang to access its (possibly dropped) field
2. explicitly specified by a programmer.

The validity of static cast from a union-structure into a field type with the case (1) is examined by the compiler, saving us from typecast checking at run-time. In contrast, the case (2) needs to be verified by `spaceMiu`, however we cannot distinguish the two cases, as mentioned.

We could simply check the validity of any typecast from the union-structure type into the target type, but there is another point to consider: there is a mismatch between the declared/effective type (union-structure) and the type of data (RTT) that is actually stored in the memory location, as we pointed in § 5.3.1. The declared/effective type (union-structure) and RTT differ in most cases. The declared/effective type does not change,

while its RTT can be overwritten multiple times. We could treat the last-written RTT as their object type, however, we may still need the declared/-effective type information. It is because typecast from a union-structure into another incompatible type may have been specified a programmer, which requires run-time checking. Therefore we keep both type information (union-structure and RTT). We update the ID of the object type (union-structure) at memory allocation, and we keep the RTT ID updated at memory write to it.

At memory write, a union-structure pointer is first typecast into a certain type ( $\mathbb{T}$ ), and then writes data in  $\mathbb{T}$ . At memory read, it is typecast back to the written data type  $\mathbb{T}$  (desirably), before reading in  $\mathbb{T}$ . Both write and read are through typecast, so run-time operations should be applied differently depending on the *use* of the cast – write or read. Instead of analysing uses of typecast at compile time and hooking differently, for union-structures, spaceMiu performs run-time checking at memory access, not at typecast.

Firstly, at data write to a union-structure, we update its RTT ID in the header. The RTT is either (a) one of the field types of an original union or (b) another type that is not a member of the union. The case (a) will be then proven safe by spaceMiu’s run-time check at memory write. The case (b), where a typecast to an incompatible type of a non-field, will be proven unsafe at run time.

At memory read, we check if it is safe to typecast from RTT, updated at the last memory write, into the type to be read. To do so, we first retrieve RTT information from the header, and then prove the compatibility between the RTT and target type.

Now we define safecast for union-structures. We treat a union-structure as a special byte array holding any data type, so we perform *bounds checking*. Here, we also check if alignment of the target type is *not larger* than the union-structure, otherwise the alignment difference may cause memory corruptions. This view is reasonable since memory access to any field is at the offset 0 by definition of unions.

Given  $\tau$  (a source type) and  $\tau'$  (a target type), where any of them is a union-structure, safecast for union-structures is defined as presented in Figure. 5.7:

$$\frac{\text{Align}(\tau) \geq \text{Align}(\tau') \quad |\tau| \geq |\tau'|}{\tau \mapsto \tau'} \text{ (Union's safecast)}$$

**Figure 5.7:** Union's safecast

For union-structures, we instrument memory access to them. Unfortunately, in practice, this causes high performance degradation from metadata management and type checking at every memory access. The overhead can be even larger than spatial memory safety enforcement, whose cost has been a major challenge.

To avoid runtime overhead, we perform typecasts for unions conservatively – hooking only memory access statically proven to be alias with union-structures by running *points-to analysis*. If a pointer operand of `store` instruction is an alias, then we hook the instruction to update the RTT ID for the object at runtime. As for `load`, we check if the alignment of the RTT is not smaller than the alignment of the type to be read with.

This instrumentation can cause false negatives or positives. We sacrifice accurate checking for unions to minimise performance degradation in the current implementation aiming at an always-on solution. For comprehensive memory/type safety, spaceMiu can be merged with FRAMER's spatial memory safety enforcement (§ 4.4), that instruments all memory accesses, on the same metadata management (§ 4.2). We then can perform operations for union-structures at memory access by recognising them with metadata in the header: `isUnionTy` indicating if the object is a union-structure, as described in § 5.3.1.

Our current header representation has limitations: it cannot handle other uses of unions such as union-structure arrays or aggregate types containing union-structures as fields. More generic and compact representation is needed to handle various uses of unions.

## 5.5 spaceMiu Implementation

There are three main parts to our implementation: spaceMiu LLVM passes, the static library (lib), and the binary lib in the dashed-lined box in Fig. 5.8.

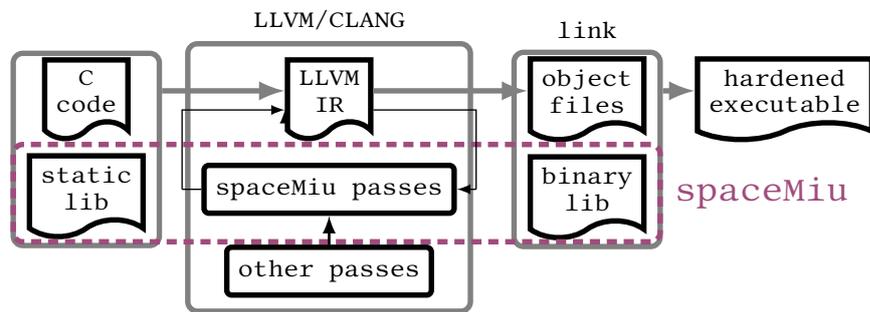


Figure 5.8: Overall architecture of spaceMiu

The source codes of a target program and our hook functions in the static lib are first compiled to LLVM intermediate representation (IR). Our main transformation pass works on the LLVM IR and instruments: (1) memory allocation/release, (2) pointer typecast, and (3) memory access. spaceMiu’s main transformation is implemented as an LLVM Link Time Optimisation (LTO) pass for whole program analysis, and runs as an LTO pass on gold linker [74], however, incremental compilation is also possible. Transformation of programs to implement tagged pointers and attaching a header to objects is the same as FRAMER described in § 4. Customised compiler optimisations are also previously discussed in § 4.5.

`malloc` family routines (`malloc`, `realloc`) are interposed with our wrappers defined in the static lib at compile time. `calloc` is not interposed at the moment, since we treat allocation with it as an array that we do not track. The third part is string functions and `free` interposed at link time. spaceMiu performs only tag-cleaning for string functions to prevent segmentation faults. Some of them such as `memcpy` or `strcpy` could be instrumented to update effective types, but we do not implement it in this implementation. Further discussion will be presented in § 5.5.6.

### 5.5.1 Creation of Type Descriptors

Before the main transformation pass traverses a program, it statically builds two type descriptors as presented in § 5.3.2 and § 5.3.3. It is straightforward to collect all used types of the whole program in our main as an LTO pass. It then creates the descriptors, layout and relation tables (Fig. 5.6), using the type IDs statically. The pass first collects all the identified types in

the target program; assigns each type a unique type ID; and then builds descriptors with the type IDs. It also assigns unique type IDs to pointer types of operands of *bitcast* operations <sup>2</sup>, so that we can remove overhead caused by redirecting entries to pointer types from their non-pointer types in the type descriptors at run time.

## 5.5.2 Program Initialisation

We insert a prologue that is performed on program startup. The prologue reserves address space for the supplementary metadata table and pages are only allocated on demand.

## 5.5.3 Memory Allocation

The current version tracks only structures including union-structures (§ 5.4) - atomics or arrays are not instrumented. Arrays can be handled by merging spaceMiu and FRAMER, and type information of atomics can be encoded in the spare bits in a 64-bit pointer, that spaceMiu and FRAMER do not currently use (discussed in § 4.7.3), without attaching a header to them for the future design.

### Stack/Global/Static Objects

Our main transformation pass performs similar instrumentation to FRAMER, presented in § 4.3.2, but updates additional per-object metadata – type IDs.

For union-typed objects, it turns on the `isUnionTy` field indicating that it is an union. The hook then creates a flag and tag (offset or N value), and moves the pointer to the second field (`Ty obj`) whose type is the actual allocated type by the target program.

We optionally instrument function epilogues to reset entries for large-framed non-static objects. The current implementation does not instrument epilogues, whereas FRAMER inserts epilogues to detect some dangling pointers.

---

<sup>2</sup>The `bitcast` instruction in LLVM IR converts a value to a given type without changing any bits.

## Heap Objects

spaceMiu's pass interposes calls to `malloc` and `realloc` with our wrappers around them at compile time, while FRAMER interposes them at link time (§ 4.3.2). Our wrappers call `malloc` and `realloc` with a user-defined size added by the header size; and the rest of operations are the same as the hook for non-heap objects except mapping objects to their type. Whereas non-heap objects are mapped to their declared type at allocation, heap objects are assigned their effective type at pointer casting during pointer assignment. Assuming that heap allocations with `malloc` meet the following signatures (§ 5.3.1):

```
Ty * p= malloc (sizeof(Ty));
```

our main pass searches the first pointer typecasting from `void *`, returned from `malloc`, to its target pointer type (`Ty *`), and passes the target type ID as an argument to our wrappers around `malloc`. The pass may fail to capture the effective type, if a `malloc` call does not meet the signatures or LLVM-clang optimises codes. It skips instrumenting the heap object, when our heuristics cannot detect the effective type.

As for heap allocations with user-customised wrappers around `malloc` family, their effective types are determined at call sites to the custom-wrappers, not to `malloc` inside them, so the pass *lazily* updates type metadata at the calls to the custom-wrappers. This delays object-to-type mapping longer at run time, and makes it more difficult to handle effective types, especially to decide whether to instrument `malloc` called by custom-wrappers. If our pass captures an effective type at one call to a user-customised wrapper, it should instrument all the call sites to `malloc`, that the custom-wrappers call. In addition, a user-customised wrapper usually has multiple call sites. Unfortunately this caused more overhead to track all heap objects including ones not mapped to any meaningful effective type in our previous experiment. To avoid the overhead, we forced inlining all custom-wrappers and then performed detecting an effective type at allocation; and tracking only objects in a type of our interest otherwise we skipped instrumenting it at inline `malloc`. The evaluation in § 5.6 is measured on inline custom-wrappers.

Handling type assignment has been a challenge for run-time type confusion checkers. As for programs in C++, UBSan [124] based on vtable pointers cannot capture type information of non-polymorphic objects. Approaches using per-object metadata such as CaVer [69], TypeSan [46], and HexType [56] instrument the *new* operator. HexType also detects types of hard-copying objects that have already been constructed. Libcrunch [60] handling C assumes a list of signatures like spaceMiu, but captures more effective types by performing source-level analysis and using debug information. One could also utilise a tool such as Coccinelle (semantic patching) [43] to statically extract effective types or the number of element (for arrays) by tokenising `malloc`'s arguments (`sizeof( $\tau$ ) * n`) and extracting two parts of a multiplication.

The current version does not interpose `calloc` that is assumed to allocate an array in our approach, that we do not track. Therefore we may have chances to miss a non-array allocation via `calloc(sizeof( $\tau$ ), 1)`.

We also interpose `free` with our wrapper but at link time. This performs resetting an entry for a large-framed object, and releasing the object with the hidden base (i.e. the address of the header).

#### 5.5.4 Type Cast

spaceMiu's transformation pass inserts a call to typecast checks right before each `bitcast` operation. It statically examines if the target type of typecast is upcast or downcast of the source type, and instruments only downcast sites. The pass passes a pointer and the target type ID to a hook along with addresses to type descriptors and other information needed to calculate the address of corresponding entries in the tables.

The hook starts with extracting a tag from a pointer: if the pointer is tagged (i.e. its referent object is instrumented), it performs checking, otherwise it returns. It then gets the header location; retrieves an object's type ID; and then performs pointer-to-type and type confusion checking.

### 5.5.5 Memory Access

Instrumentation of memory access is similar to FRAMER presented in § 4.3.3. The transformation pass instruments basically every memory access with a hook function just to clean tag, unlike FRAMER.

As mentioned in § 5.4, updating type information or verifying typecast of unions are performed at memory access. The main pass runs points-to analysis for each function and determines if an individual memory access is alias with any union. If so, the pass hooks memory writes with a function that updates the type ID for RTT (i.e. the type of `store`). For memory reads to unions, it inserts a hook checking if the typecast from the union's RTT to the type of `load` instruction is safe.

### 5.5.6 String Functions

Tag-cleaning should be also performed on string functions such as `memcpy`, `memmove` or `memset` in a similar way. We interpose them at link time with our wrappers, that call wrapped functions with tag-cleaned pointers and then restore the tag for their return value.

Some string functions such as `memcpy` and `strcpy` may be involved with deciding effective types. We do not implement in that aspect. Further extension to handle the issue is discussed in § 5.7.2.

## 5.6 Evaluation

The performance of `spaceMiu` is measured and evaluated on C benchmarks from SPEC CPU 2006 [49]. For the benchmarks having multiple tests (`perlbench`, `bzip2` and `gobmk`), the average of the tests is presented.

For each benchmark, two binary versions are measured: uninstrumented and instrumented by `spaceMiu`. Binaries were compiled with the regular `LLVM-clang` version 4.0 at optimisation level `-O2`. The same compiler optimisations in the same order are applied to two versions. Measurements were taken on an Intel<sup>®</sup> Xeon<sup>®</sup> E5-2687W v3 CPU with 132 GB of RAM. Results were gathered using `perf`.

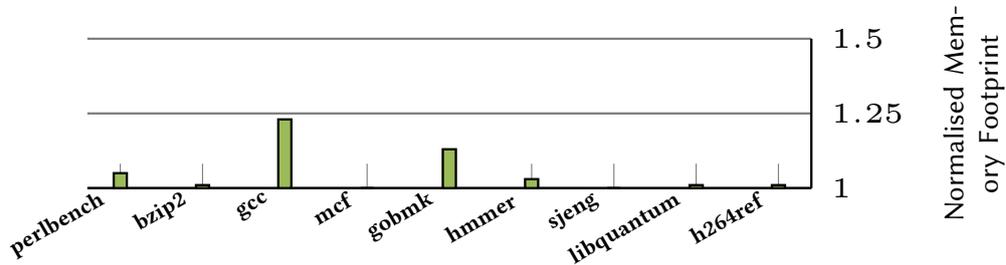


Figure 5.9: Normalised Memory Footprint

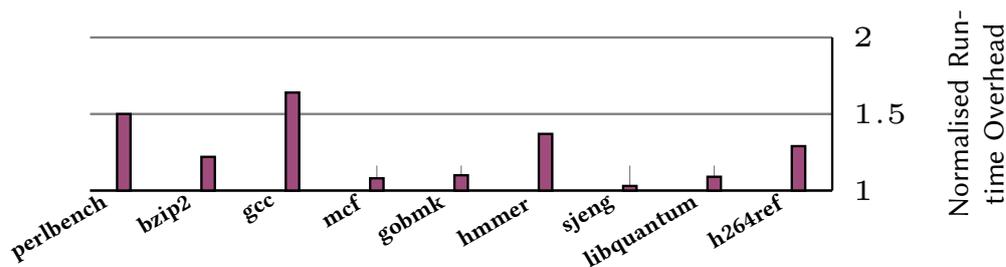
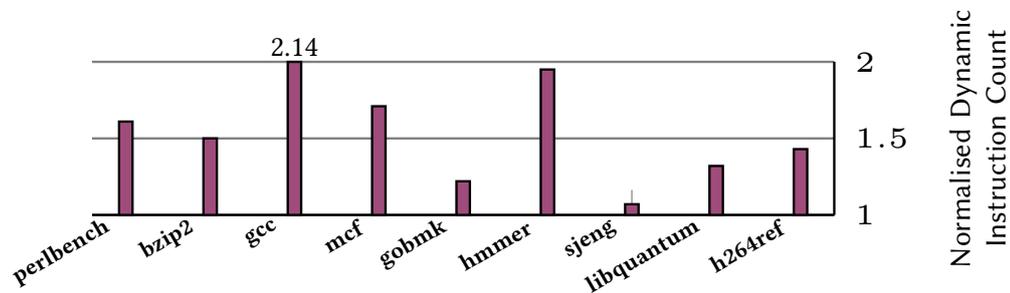


Figure 5.10: Normalised runtime overhead

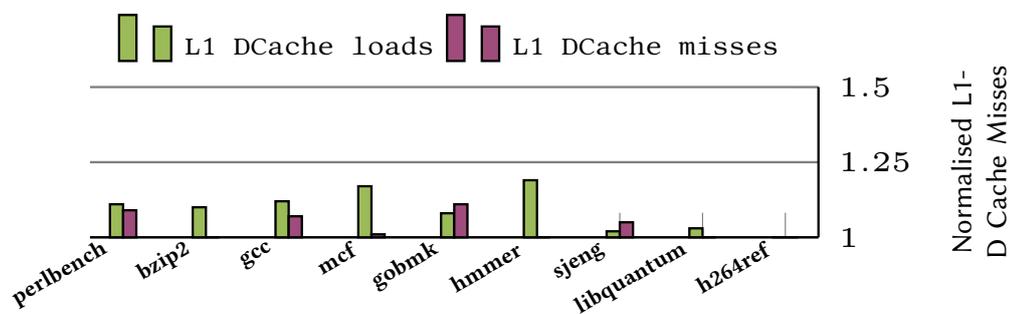
In this text, cache and branch misses refer to L1 D-cache miss counts and branch prediction misses, respectively.

### 5.6.1 Memory Overheads

Our metadata header was 16 bytes per structure-typed object. The large-frame array had 48 elements for each 16-frame (division) in use where the element size was 8 bytes to hold full address of the header. The header size and the number of elements of each division array can be reduced. Currently we mandate 16 alignment for compatibility with the `llvm.memset` intrinsic function that sometimes assumes this alignment. Despite inflation of space using larger-than-needed headers and division array entries and some changes of alignment, we see spaceMiu's space overhead are low at



**Figure 5.11:** Normalised Dynamic Instruction Count Overhead

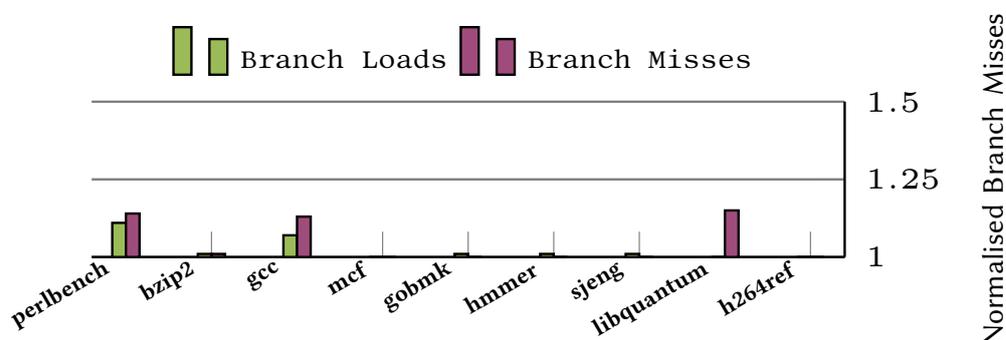


**Figure 5.12:** Normalised L1-D Cache Miss Count Overhead

1.05 as shown in Fig. 5.9. These measurements reflect code inflation for instrumenting metadata manipulation and typecast checking.

Despite overhead for type descriptors and padding comparably a smaller object than arrays with a fixed-sized header, the overall overhead is mainly influenced by increase in number of instrumented objects. One of `perlbench` tests with average overhead (5%) reaches 30% with instrumentation of all heap objects excluding space for descriptors. Two tests, `gcc` (23%) and `gobmk` (13%) recorded higher growth than other tests however it is still low.

The average memory overhead of `spaceMiu` (5%) instrumenting only structure-typed objects is lower than tracking global/static/local arrays and all heap objects for bounds checking (22 ~ 23% on average). `spaceMiu` does not track heap objects whose effective type is not recognised by the



**Figure 5.13:** Normalised Branch Miss Count Overhead

main pass (i.e. not following signatures) so many heap objects are dropped from instrumentation. Amongst tests with heap allocation via customised wrappers (e.g. `perlbench`, `gcc`, and `gobmk`), forcing inline those wrappers before instrumentation helped detect more effective types in the test `gcc`, while inlining was not of much help for `perlbench`. Many heap objects were discarded in the test `perlbench` and tracking all heap allocation increases the overhead to 23% from 5%. The increase comes from both object padding and a supplementary table.

## 5.6.2 Slowdown

spaceMiu's runtime overhead on average is 26%. We track only structures that are involved in up/downcast, so tests with a small number of instruments recorded comparably low increase both in memory footprint and cycles. The slowdown is mainly caused by the increase in executed instruction to set up tags, clean tags, and perform checks at typecast including calculation of type information location and retrieval of information.

Slowdown from tag-cleaning and branches is one of the downsides of using tagged pointers, but much of the overhead will be resolved with customised instructions. spaceMiu manipulates tags both at typecast for tag extraction and memory access for tag-cleaning to avoid segmentation fault. Tag-cleaning at memory access solely costs around 10% (> 10% in some tests) which is heavy for typecast checkers having a better chance for practical deployment however this can be resolved with hardware acceleration.

At typecast, spaceMiu extracts a tag from a pointer and checks if the tag is not zero, that results in additional branches. Removing duplicate operations to clean tags combined with points-to analysis can resolve more overhead.

`gcc` (64%) stands out the highest overhead among all the tests, and `perlbench` family test (50%) comes second as shown 5.10. Two tests are the main contributor to raise the average and excluding the two tests, the average is 17%.

The main contributor in the tests with the highest overhead varies. The test `perlbench` heavily typecasts pointers. Checking dominates the overhead of dynamic instruction count up to 48% excluding the cost of tag-cleaning (60% including tag-cleaning). In addition, a number of typecast errors are detected by spaceMiu, so extra branch failures at typecast checking causes more overhead (§ 5.6.5). The test `gcc` consumes more cycles on non-heap objects and spaceMiu's statically removing duplicate runtime checks using dominator tree and points-to analysis works well on `gcc` (up to 60,000 at static time) and this resolved overhead for type checking at run-time.

Compared to `gcc` or `perlbench`, the overhead for the test `hammer` is not high (37%) despite the highest increase in dynamic instruction count (95%) next to `gcc` (114%) but higher than `perlbench` (61%). `hammer`'s increases in L1-Dcache loads and branch loads are 10% and 2% respectively, while L1-Dcache miss counts and branch misses slightly decreased by 1% and 3%.

### 5.6.3 Executed Instructions

Fig. 5.11 reports normalised executed instruction overheads per benchmark. The increase in dynamic instruction counts is 55% on average.

The increase in executed instructions is the main source of slowdown. spaceMiu consumes majority of dynamic instructions on (1) setting up tags at allocation, (2) retrieval of per-object type information; pointer-to-type mapping; and then type confusion checking, and (3) tag-cleaning at memory access. Metadata update and typecast checking is also performed at memory access to unions: updating metadata at `store` and checking at

load whose pointer operand is statically identified as an alias with unions. The cost to check unions was very small on the tests spaceMiu was evaluated on.

The dominant source of the growth is arithmetic operations. The overall instruction overhead is still not dramatically high, since the occurrences of pointer typecasts is much lower than that of memory accesses in most programs, and spaceMiu skips run-time checks on pointers to referent objects that are not instrumented (i.e. tag-free pointers).

To save untagging operations for the same pointer without using hardware acceleration, we could run static analysis but it was not widely applied in that aspect in this version. Currently spaceMiu does not perform inter-procedural analysis for this, so skipping tag-cleaning is applied conservatively during optimisation.

As previously mentioned, there are heap objects whose effective type is not recognised by our compiler pass. It may be more thorough to interpose at allocation at link time and catch it at the first typecast at run time, assuming that a heap memory object's effective type does not change. Unfortunately we experienced much higher runtime overhead on this implementation. However it still can be combined with static analysis to find the first typecast operation in charge of effective type decision.

Future implementation can optimise some cases as discussed in § 5.7.1.

#### 5.6.4 L1 D-cache Misses

L1 D-cache misses arise when accessing an object and its header that do not fit in one cache line that is normally 64 byte-sized, and for big-framed objects, an indirect access to the supplementary table. In addition, more cache misses at access to type descriptors,

Figure 5.12 shows normalised L1 D-cache loads and misses per benchmark. The cache misses increased by 4% on average after instrumentation, while cache loads increased by 9%, so the miss rate went down overall. The baseline L1 D-cache miss rate was 10.25% and it slightly improves to 9.32% with spaceMiu enabled owing to repeated access to the same cache data.

Two tests recorded the highest increase in miss count – gobmk (11%) and perlbench (9%). As for perlbench, the low growth in memory footprint

(5%), high in executed instructions (61%), high in branch loads and misses (14% and 11%) tells that the growth in number of instrumented objects are comparably not high but `perlbench` is typecast-intensive, requiring frequent access to type descriptors in a remote region. In contrast, the memory overhead in the test `gobmk` is higher than average (13% > 5%) but the increase in branch loads, cache loads, and executed instructions are all below average. We can conclude that the higher increase in cache misses in the test `gobmk` is dominated by (1) bigger growth in the number of memory allocation or in bad alignment, and (2) updating metadata of objects that are not involved in typecasts at allocation.

### 5.6.5 Branch Prediction

Branches arise to decide whether an object is large-framed or small-framed at memory allocation of any kind of objects (static, stack, and heap). `spaceMiu` also checks if a pointer is tagged to decide if a pointer to be released at heap memory release via `free`. In addition, `spaceMiu` generates branches at the typecast checking in the followings cases:

1. whether a pointer is tagged
2. whether a pointer is small or large-framed
3. whether a pointer references a *meaningful* offset in a structure
4. whether the pointer typecast is safe or not.

Checking the validity of pointer typecast generates branch mis-prediction. As described, we conservatively report unsafe typecasts at run time e.g. tolerating 64-bit integer to double *safe*, that can be considered *unsafe* from the traditional and strict type safety policies.

As shown in Fig 5.13, branch misses increased slightly after instrumentation by 4% and branch loads increased by 9%, so the branch mis-prediction rate slightly decreased. This shows the additional branches achieve highly accurate branch prediction and that branch predictors are not being overloaded. Three tests `libquantum` (15%), `perlbench` (14%), and `gcc` (13%) recorded the highest overhead. While `perlbench` and `gcc` are comparably big programs instrumenting many objects and cast-intensive, `libquantum`

recorded lower overhead in other criteria (memory, executed instructions, and cache misses) below average. The high branch misses are dominated by mis-prediction of tagged/untagged pointers and small/large framed objects.

## 5.7 Discussion

### 5.7.1 Effective Type Detection for Heap Objects

To find an effective type of heap allocation with `malloc`, spaceMiu's LLVM pass searches the first typecast of a `void` pointer to the object into non-`void` pointer type as described in § 5.5.3. Under the assumption that the first typecast follows a call to `malloc` in the same function, their type information can be easily captured, but unfortunately other cases are dropped from tracking. In addition, instrumenting all heap allocations via customised `malloc` wrappers and *lazily* updating type metadata can waste performance to track heap objects whose types are captured. The whole program analysis using call graphs, dominator tree, and alias analysis can improve this.

A more accurate way to find effective types of heap allocations is to capture them at the first typecast at run time, however unfortunately this causes heavy run-time overhead especially for `malloc`-intensive programs.

Another way to detect effective types with less overhead but with more accuracy is to use a preprocessor. We can tokenise `malloc`'s arguments (`sizeof( $\tau$ ) * n`) by extracting two parts of a multiplication and then associate a heap object with its type information, object kind, and the number of element (for arrays) during the compile time. This is possible to transform the source code using a tool such as Coccinelle (semantic patching) [43] that can take those rules.

### 5.7.2 String Functions and Effective Type

The current implementation to capture effective types may miss some cases, and one example is buffers whose data is hard-copied with `string` functions such as `memcpy` or `strcpy`. The effective type is determined at

```

1 // Ty is structure type.
2 Ty obj = 55;
3 void* vp = malloc(sizeof Ty);
4 memcpy(vp, &Ty, sizeof Ty);
  /* *vp now is Ty typed */
5 Ty* gp = malloc(sizeof *gp);
6 memcpy(gp, &obj, sizeof *gp);
  /* *gp now is Ty typed */

```

**Figure 5.14:** String functions and effective type

memory copy. In Figure 5.14, `memcpy` not only changes the values of each byte of the object, but also determines the object's effective type. Our pass can detect the effective type of the object `gp`, allocated with `malloc` in the line 5, but fails for the object `vp`, since `vp`'s effective type is determined by hardcopying (line 4). We can handle this by adding instrumentation of string functions and modifying our instrumentation of `malloc` not to skip instrumenting the heap allocations whose effective types are not captured.

### 5.7.3 Aliasing Rules

A structure may have `void` pointers that are involved in `typedef` as follows:

```

1 struct A {void * pa1; T1 * pa2;};
2 struct B {T2 * pb1};
3
4 T2 myt2= {...};
5 struct A a;
6 a.pa1= &myt2;
7 struct B b= (T2*)p->pa1; /* run-time check here */

```

The contract type of the pointer `pa1` is `void`, but its storage content is `T2` (line 6). With aliasing rules, this cast with `void*` is illegal in C (except when the other type is `char`). `spaceMiu` reports this cast as an error (line 7) although the cast from the storage content type to a target type is safe upcast.

### 5.7.4 Per-object Metadata Placement

Placing metadata in a header may be more preventive from metadata corruption through buffer overflows or unsafe typecasts than in a footer especially on the current 16-alignment. Downcasts and memory over-runs may be critical, especially for approaches using embedded metadata (e.g. fat pointers or tagged pointers), since memory writes after unsafe typecasts to user data in a program's can pollute metadata in a header of neighboring objects. Protecting metadata is easier with spaceMiu than with fat pointers. We can detect memory overwrites to another object's header caused by downcasts by adding FRAMER, that performs bounds checking. Unlike fat pointers, we do not need to check internal overflows by unsafe downcasts to protect metadata, since metadata is placed outside an object.

However using a header not a footer has a disadvantage – consuming branches at memory release via `free` to decide if a pointer is tagged. If a pointer is tagged, spaceMiu must move its pointer to a hidden base (i.e. a header) to free the object, otherwise frees a pointer passed as an argument.

With support of both spatial memory safety and type safety, attaching metadata at the end of an object is a better implementation for performance, otherwise using a header is safer.

## 5.8 Conclusion

This study presented spaceMiu, that demonstrates how tagged pointers can be used for run-time type confusion checking for C programs. This checker implements physical sub-typing for C and pointer-to-type mapping and detects unsafe typecasts. Based on FRAMER's metadata management, spaceMiu derives a metadata pointer from an object pointer by exploiting the unused top 16 bits of a 64-bit pointer accesses to its type layout and relation information with efficient database management.

Memory safety and type safety are highly similar and connected in the sense that they both require tracking of memory allocation and load/store addresses to detect safety violations. For both application, the memory footprint and runtime overhead of the checking needs to be kept low (e.g. < 5%) so that it can be practically deployed. Type safety enforcement

---

is an easier problem since the occurrence of typecast is much lower than load/store, making type safety more light-weight. Currently spaceMiu is slightly heavy for an *always-on* type safety enforcement (> 5%). However this approach for both memory/type safety could be better with greater static program optimisation and ISA improvements.



# 6

## MemPatrol: Reliable Sideline Integrity Monitoring for High-Performance Systems

---

### 6.1 Overview

Integrity checking using inline reference monitors to check individual memory accesses in C/C++ programs remains prohibitively expensive for practical deployment in the most performance-critical applications. To address this, this chapter presents *MemPatrol* [92], a sideline memory integrity monitoring system that detects a class of memory corruption attacks with very low performance overhead, using available Linux kernel primitives and Intel CPU encryption facilities. *MemPatrol* aims (i) to guarantee the eventual detection of integrity violations regardless of the detection delay, by reliably protecting itself against a compromised application during the time window between the occurrence of the attack and its eventual detection, and (ii) to give engineers the flexibility of tuning the cost of integrity monitoring in a reliable and predictable way by configuring the desired amount of computational resources allocated to it.

*MemPatrol* implements a userspace-based isolation mechanism by using CPU registers as the integrity monitor's private memory, allowing the monitor to safely run as a thread inside the address space of the protected application. The CPU registers cannot, obviously, hold all the information required to run an integrity monitoring system, such as the addresses

and expected values of memory locations. However, they are sufficient to store cryptographic material and run a register-only message authentication code (MAC) algorithm to reliably access the rest of the data required for the monitor's operation.

Attackers in control of a compromised application thread cannot tamper with the monitor thread's information that is offloaded to memory without detection, because they lack access to the key used to authenticate it. The authentication key is only available to the integrity monitor, and threads cannot address each other's registers. The key and intermediate states of the MAC algorithm stay only in registers, never being flushed into userspace memory. The monitor's code never spills registers and does not use primitives such as `setjmp/longjmp`. The registers may only be flushed to kernel-space memory during a context switch, where they remain unreachable to a potentially compromised userspace application. Besides this main idea, this chapter discusses how *MemPatrol* prevents replay attacks, and the mechanisms based on SELinux and the `clone` Linux system call which are required to protect the monitor thread from forced termination by its parent process using `kill` or modification of its code memory to alter its execution.

This work studies a concrete special case of sideline integrity monitoring for detecting heap buffer overflows in a commercial high-performance passive network monitoring system [96] where existing memory safety techniques are too expensive to apply. We believe, however, that periodic integrity checking of memory locations in a program's memory can have additional applications. For example, it could be used to detect malicious hooks installed by modifying persistent function pointers.

Of course, even a concurrent monitoring system incurs performance overhead that may affect application threads, for example, memory bandwidth overhead from increased reads, cache coherency overhead, and other cross-CPU communication in NUMA systems. The low overhead imposed by our isolation mechanism, however, enables engineers to minimize monitoring cost arbitrarily by throttling integrity monitoring without compromising eventual detection.

In summary, *MemPatrol* make the following contributions:

1. An effective, userspace-based isolation mechanism for the monitor thread that does not require new Linux kernel modifications
2. Demonstration of tunable and predictable allocation of resources to security monitoring, in particular memory bandwidth
3. Avoidance of synchronization overhead for heap monitoring by taking advantage of the memory allocation mechanisms used in performance-critical systems

The remainder of the chapter is organized as follows. Section 6.2 describes our threat model. Section 6.3 presents our monitor thread isolation mechanism. Section 6.4 applies our mechanism to monitoring of heap integrity, and Section 6.5 evaluates its performance. Section 6.6 reviews discussion for the better design, and Section 6.7 concludes with final remarks on the current limitations and future directions of this work.

## 6.2 Threat Model

In this section we discuss *MemPatrol's* threat model. Firstly, we discuss the threat model for integrity monitoring using a concurrent thread in general. Secondly, we discuss the threat model for heap memory corruption attacks that we use as a concrete case study of integrity monitoring.

### 6.2.1 Sideline Integrity Monitoring

*MemPatrol's* threat model for integrity monitoring, in general, considers attacks as malicious modification of memory contents, whether on the stack, heap, or static variables. It divides the life cycle of a protected application into two phases: the *trusted* and the *untrusted phase*. *MemPatrol* assumes the program starts in the trusted phase, during which the application is responsible for registering all memory locations to be monitored and then launching the monitor. The program, then, enters its untrusted phase before the application receives any inputs from potentially malicious third parties that could compromise it.

This work assumes that at some point after entering the untrusted phase of its lifetime, the application becomes vulnerable, for example, by starting

to process external input. After this point, it is no longer trusted by the monitor that was launched before the process became vulnerable. In particular, we assume that a compromised process may access any memory location that lies in its address space, and may attempt to restore any data corrupted during the attack leading to the compromise, in order to avoid detection. A compromised process may also invoke any system calls, such as `kill`, to terminate other processes or threads, subject to OS controls. Attacks against the OS kernel, however, are outside the scope of this work.

Finally, while the application is under the control of the attacker, we assume the attacker may perform replay attacks, meaning that older contents of the memory can be saved and reused to overwrite later contents.

### 6.2.2 Heap Integrity

We built a concrete case study of *MemPatrol* by applying it to heap buffer overflow detection based on detecting canary data modifications, and evaluated it with a high-performance passive network monitoring system [96]. Other threats such as stack-based buffer overflows are handled by existing defences, such as GCC's stack protector (`-fstack-protector` set of options), or fall outside the scope of this case study.

The program is assumed to be compromised through heap buffer overflows employing only contiguous overwrites. Buffer overflows often belong to this category, and we do not consider other memory safety violations, such as those enabling corruption of arbitrary memory locations.

The attacker may corrupt any kind of data in heap objects by overruns across adjacent memory chunks. For instance, attackers can overwrite a function pointer, virtual function table pointer or inlined metadata of a free-list-based memory allocator by overflows. *MemPatrol* assumes that attackers may overwrite contents across multiple buffers in both directions, i.e. underflows and overflows.

Finally, *MemPatrol* assumes that the canary value cannot be learned through *memory disclosure*<sup>1</sup> attacks [44, 48, 27]. However, note that the stan-

---

<sup>1</sup>Memory disclosure is one of the common information leak vulnerabilities. It occurs when a system forgets to clear a memory block before using it to construct a message that is sent to an untrusted party.

Standard form of memory disclosure attack is impractical with passive network monitoring systems, such as [96], because there is no request-response interaction with an attacker to exfiltrate the data. An “indirect” elaboration of the attack is conceivable, that caches the contents of the canary to another buffer inside the process, used later to restore the canary. For this to work, the copy must not corrupt another canary, so it must be achieved using random access, which the current solution does not cover. These attacks are outside the scope of this case study.

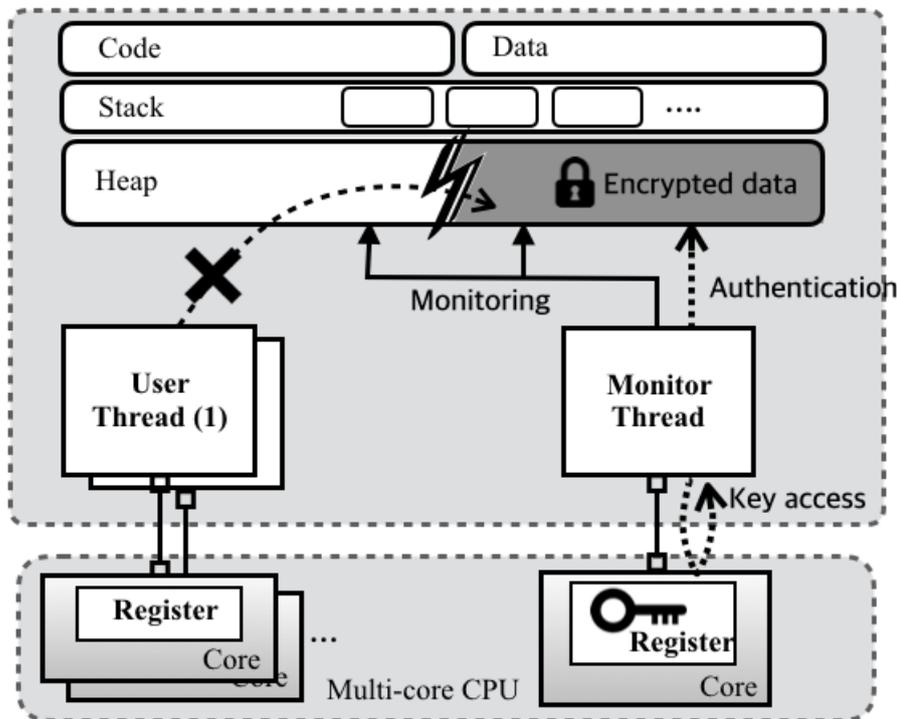
In summary, we assume the attacker can gain control of the execution of the application through heap buffer overflows, but we cannot defend against overflows that stride over heap canaries without overwriting them, other kinds of memory safety violations, or against information leakage through memory disclosure attacks.

## 6.3 Monitor Thread Isolation

Sideline integrity monitoring systems offer asynchronous detection with a delay. Crucially, if this detection latency can be exploited to disable the monitor, no concrete security guarantees can be made. To avoid this, we need to anticipate all possible scenarios under which a compromised application can disrupt the monitor thread, and thwart them. This work has identified the following ways that an attacker with full control of the application can disrupt a monitor thread running in the same address space:

1. Tampering with the monitor’s data structures on heap or its stack
2. Hijacking the control flow of the monitor by manipulating the monitor thread’s stack
3. Hijacking the control flow of the monitor by altering the monitor thread’s executable code in memory
4. Terminating the monitor thread via the `kill` system call
5. Faking application termination

In the following sections, we discuss how to block these attacks.



**Figure 6.1:** The cryptographic key is only accessible to the monitor thread by storing it in a register, and additional information stored in memory is authenticated cryptographically.

### 6.3.1 Protection of Data Structures in Memory

Attackers may attempt to subvert the monitor thread by corrupting data in the program’s memory used by the monitor, such as the list of memory locations to monitor. This would effectively disable monitoring. Besides these data structures that are stored on the heap, an attacker could alter local variables or spilled registers on the stack of the monitor thread.

This solution is for the monitor thread to only trust data in its registers. Of course not all data can be saved in the register file due to the limited space of registers. Instead, any data stored outside of registers must be authenticated to prevent tampering. *MemPatrol* achieves this using cryptographic techniques. The cryptographic key used for authentication is stored only in a register as shown in Figure 6.1. Compromised application threads cannot succeed in corrupting data without detection, because they do not have access to the cryptographic key required for counterfeiting the

stored information. Of course, it is not sufficient to merely protect the key in a register. It is also required that the entire authentication mechanism is implemented using only registers, and that the main loop of the monitor thread also only trusts registers and authenticates any memory used. Next, we describe the memory authentication primitives and the methodology followed to implement the monitor code using only registers for its trusted data instead of memory.

### Authenticated Memory Accesses

To secure data stored in untrusted memory from being counterfeited, we use AES-based Message Authentication Codes (MAC) to sign the value and its location. This work chose AES because we can utilize the AES-NI [45] instruction set of Intel processors which provides a hardware implementation of AES using the `aesenc` and `aesenc1ast` instructions for the encryption operation. Each of them performs an entire AES round without any access to RAM. *MemPatrol* uses the compiler's intrinsics to access these instructions. Note however that these hardware extensions are used in this work for convenience and performance. In principle, our solution does not depend on the availability of dedicated cryptographic instructions, as CPU-only implementations of AES on processors without AES-NI exist [117].

Every AES round requires a different round-specific key expanded from the initial key. These are typically expanded once and stored in a memory-based array and reused for every operation. We cannot use a memory-based table and we also dedicate using 10 registers, one for each round's key, by interleaving the key expansion with the encryption rounds. This technique cannot be used with decryption, because decryption requires the expanded key in reverse order, so all the stages would have to be kept in registers. Fortunately, the decryption operation is not required for implementing message authentication codes.

Figure 6.2 illustrates the authenticated memory access routines used by the monitor thread. The routines can store and load data in units of 64-bits expanded into 256 bits of memory, namely the `sec64_t` type that includes the value and its signature. Specifically, we pack the 64-bit address of the

```
typedef struct {
    __m128i m;    /* Data word and its (albeit redundant) address */
    __m128i mac; /* Message authentication code */
} sec64_t;

/* Store a verifiable word */
void store_sec64(uint64_t word, sec64_t *sec, __m128i key);

/* Return a verified word or execute an illegal instruction */
uint64_t load_sec64(sec64_t *sec, __m128i key);
```

**Figure 6.2:** Untrusted-memory data type and access routines

`sec64_t` object and the 64-bit value into 128 bits of data, and produce an additional 128-bits of MAC by encrypting the concatenation of the address and value using the key with the help of the `store_sec64` routine.

To retrieve the value, the `load_sec64` routine regenerates the signature using the address of the `sec64_t` passed to it, the value from the `sec64_t`, and the key passed to it in a register. If the signature does not match, it raises a trap, otherwise it returns the value.

## Replay Attacks

To block attackers from maliciously overwriting an entry with a signed datum from a different memory location, we include the memory address in the authenticated data. To block attackers from reusing signed data representing previous values of the same memory location, we avoid storing new data to the same address. Note, however, that we can enable a limited form of secure updates by using append-only tables and keeping the table size in a register.

## Writing Register-only Code

While it is entirely possible to implement the monitor thread in assembler by hand, we found that this was not necessary. Here we describe the methodology used to achieve the same result and to verify its correctness.

First, we isolated the code that must avoid using unauthenticated memory into its own source file, compiled with a controlled set of GCC options, and manually inspected the generated assembly. Initially we attempted to

instruct GCC to use specific registers by using `asm` annotations on variable definitions. This achieved control of the registers used, but unfortunately it generated memory accesses for superfluous temporaries. Instead, we had to rely on GCC eliminating register usage through optimization, by compiling the code with `-O3` (and also `-msse4 -maes` for SSE2 and AES-NI). Using stock AES-NI routine implementations for the MAC routines produced code with register spilling. Obviously these routines must not use any memory, as they are the ones that we rely on for authenticating memory use elsewhere. This is addressed by modifying the stock encryption routines to interleave the round-key generation with the encryption rounds. This was sufficient for implementing a MAC algorithm and the memory access routines.

Next, we worked in a similar way on the register-only implementation of the main loop of the monitor thread. Functions calls could not be used, because they would use the stack to save their return address and temporary variables from registers, so we placed the previously crafted `store_sec64` and `load_sec64` routines in a header file and annotated them with the `always_inline` GCC attribute. After some experimentation, the desired code was achieved. Of course, the solution does not rely on these techniques, as we could always write the core routines of the system directly in assembler.

Finally, besides manual verification of the generated assembly code, we zero out the `rsp` register at the start of the integrity checking loop using inline assembly, forcing any stack frame access to cause a crash. This ensures we do not accidentally introduce memory accesses due to spilled local or temporary variables as the code evolves, or in subsequent recompilations.

### 6.3.2 Protection of Code

Another way the application's threads can subvert the monitor thread is by modifying its executable code in memory while it runs. On x86 there is no need to flush instruction caches for program memory modifications like this to take effect. Code segments are write-protected by default, but attackers in control of the process could easily call `mprotect` on a vanilla Linux kernel to gain write access to the code section of the monitor thread.

They could then neutralize the monitor thread without causing it to exit by replacing its code with a version that does not perform integrity checks.

With a vanilla Linux kernel, this attack is entirely possible. However, solutions to prevent the modification of program code are already included in most Linux distributions. For example, the PaX project introduced MPROTECT[101], a kernel patch designed to prevent the introduction of new executable code into the task's address space by restricting the `mmap` and `mprotect` interfaces. Security-Enhanced Linux (SELinux) [113] also contains the `execmem` access control, to prevent processes from creating memory regions that are writable and executable. One of these common solutions needs to be used to prevent this attack. We use Red Hat Enterprise Linux which provides SELinux.

### 6.3.3 Terminating or Tracing the Monitor Thread

A trivial attack scenario that must be tackled is termination of the monitor thread by the compromised process using the `kill` system call, or subverting it using `ptrace`<sup>2</sup>. We address this scenario by using the Linux `clone` system call which allows fine-grained control over sharing parts of the execution context. We start the application as a privileged user and, instead of the regular POSIX thread interfaces, we use `clone` with the `CLONE_VM` flag to create the monitor thread. After the monitor thread is launched, the main application drops its privileges by executing `setuid` and `setgid`. This results in two threads/processes (the distinction between thread and process is not so clear when using `clone`) running in the same address space without sharing user credentials. The monitor thread retains the privileged user credentials, while the rest of the application is running with reduced privileges, and thus cannot use the `kill` system call to signal the privileged thread, nor `ptrace` to debug it.

---

<sup>2</sup>With the `ptrace` system call, one process (a tracer) can pause another process (a tracee), inspect and set the tracee's registers and memory, or intercept system calls. It is used to implement breakpoint debugging and system call tracing.

### 6.3.4 Faking Application Termination

Under this scenario the attacker may call `exit` in order to terminate the process before the monitor thread had a chance to detect the attack. Uninitiated termination of the application process could be considered sufficient grounds for raising an alarm, but we also address this scenario by ensuring that a final integrity scan is performed on `exit`.

### 6.3.5 Detection of Normal and Abnormal Termination

The monitoring system needs to detect normal application termination, in order to also terminate, as well as abnormal termination triggered by a MAC failure in order to raise an alarm.

Unfortunately, it is impossible to receive notification of the termination of the process by a signal through the `prctl` mechanism with the `PR_SET_PDEATHSIG` option, because of the different user credentials used for isolation with the explicit purpose of disallowing signals. Instead, the monitor needs to detect the termination of its application by polling its parent PID using `kill` with a signal number of 0.

As we have discussed, the execution of the monitor thread is severely constrained, to the extent that calling the libc wrapper for `kill` can compromise it by dereferencing the return address saved on the stack. It is technically possible to send a signal to another process in a safe manner on x86 Linux by running the `syscall` machine instruction directly. It accepts its input parameters in registers and stores the return address in a register. However, it is more convenient to use a more flexible scheme described next.

To detect termination, we use an additional monitor *process*, spawned as a child of the monitor thread using the normal `fork` mechanism. Unlike the monitor thread, this process does not share its address space with the monitored application. Therefore it is free of the draconian execution constraints imposed on the monitor thread. This process can poll the main application using `kill` in a loop to detect its termination and signal the monitor thread, which is possible, as they are running under the same UID. The monitor thread must perform a final integrity check of the application

before exiting, to handle the possibility of a process termination initiated by the attacker, as discussed earlier.

As for abnormal termination, once the monitor thread detects an integrity violation, it has limited options due to its constraints. We call the `__builtin_trap` intrinsic instruction which on x86 Linux compiles to an illegal instruction and generates a `SIGILL` signal, terminating the monitor thread. The termination is detected by the monitor *process*, which has the flexibility required to alert operators.

### 6.3.6 Minimizing Performance Impact

The execution of a concurrent monitor thread, unlike inline reference monitors, does not increase the dynamic instruction count of the monitored program's threads. However, its presence may still incur other kinds of overheads affecting them including cache pollution, memory bandwidth increases, and cross-CPU communication.

To minimize last-level cache pollution, we ensure that the monitor thread is using non-temporal memory accesses, which are available to C code by using the `__builtin_prefetch` intrinsic. Unlike inline monitoring, refraining from cache use only affects the performance of the monitor thread itself, which translates to detection delays, rather than slow down of application threads.

Moreover, network monitoring systems go to great lengths to avoid paging overheads because the jitter introduced to execution time by having to walk the page tables on a miss may lead to packet loss. For example, they utilize so-called huge pages introduced in modern processors, typically sized at 2 MiB or 1 GiB instead of the default page size of 4 KiB on x86. We additionally avoid any such overhead by sharing the entire address space, page tables included, with the monitored threads.

To avoid hogging memory bandwidth and minimize cross-CPU communications, the monitor thread should pace its memory accesses. In fact, we allow the rate of memory accesses to be configurable as a means to allow the user to select the desired level of overhead, at the expense of detection delay. This allows the user to tune for minimal impact on the application threads.

In summary, we explore a set of design trade-offs to avoid overhead to the application threads at the cost of the monitor thread's speed. This highlights the importance of protecting the monitor thread itself so that this trade-off does not result in invalidating the approach.

### 6.3.7 Limitations

This approach assumes attackers change the behavior of a monitor by exploiting corruption on data that the monitor uses. Hence, we do not cover other heap exploits of memory writes, for example, heap spraying or JIT spraying filling the heap with many objects containing malicious code, and increasing the success rate of an exploit that jumps to a location which can then be pointed to and triggered. Attacks, not directly meddling with a monitor's data or code, are not covered, so spraying attacks should be handled with other defenses [107, 136]. For wider protection of critical control-flow and data-flow variables in memory, one can use control-flow protection mechanisms (discussed in § 2.3) such as Cryptographically Enforced CFI (CCFI) [78]. CCFI also leverages cryptographic primitives to protect control-flow information such as return addresses, function pointers, and vtable pointers. Compared to *MemPatrol*, CCFI offers comprehensive protection against control flow violations but does not protect against some non-control-flow data corruptions that *MemPatrol* can detect. Moreover, CCFI is an inline solution, hence it directly affects the performance of the application threads (3–18% decrease in server request rate).

As mentioned in § 6.2.2, information leakage attacks, allowing reading and disclosure of the canary values, are not handled for general purposes other than our case study for passive network monitoring systems.

## 6.4 Case Study: Heap Integrity

Heap canaries can be used for detecting heap buffer overflows. They are fashioned after stack-based canaries and work in a similar way. Typically the canaries are checked on deallocation, which for our use case would lead to frequent checking and overhead to the main application threads for short-lived objects, or excessive detection delays for long-lived ones.

As a case study of integrity monitoring, we apply *MemPatrol* to canary-based heap integrity checking. We use the monitor thread to patrol heap buffers, and detect illegal memory overwrites across boundaries between heap objects by checking corruption of canary values placed strategically between heap-allocated objects. In this section we describe our implementation of sideline integrity checking for heap canaries.

### 6.4.1 Memory Pools

To check heap canaries, the monitor needs to keep track of heap allocations. Existing sideline heap buffer overflow detection systems achieve this by intercepting heap allocation functions to track live heap buffers and collect the addresses and sizes of live heap buffers. This can be a significant source of overhead, slowing down the main application threads.

High-performance applications typically use fixed-sized *memory pools* for performance, such as the memory pool library for network monitoring systems provided by the DPDK [51] toolkit. Memory pools is the use of fixed-sized memory blocks (pools) for memory management that allows dynamic memory allocation. The allocator preallocates memory pools, allocates, accesses, and frees blocks represented by handles at run time.

We designed our monitoring system to take full advantage of such memory pools. Instead of tracking individual object allocations, we track information at the granularity of memory pools: the base address of each pool, the number of blocks, and the size of a block in the pool are included in an entry and added to the append-only table used by the monitor thread. This enables the bulk setup of heap canaries before their use in the fast path of the program. Memory pools also enable reusing canaries between allocation lifetimes (since typically the object size is fixed per memory pool). Such *canary recycling* eliminates synchronization overhead suffered by existing solutions designed around a `malloc`-style arbitrary object size interface.

### 6.4.2 Integration with the Monitored Application

*MemPatrol* is implemented in the form of a library offering an API for integration with applications. To use *MemPatrol* the application needs to augment its memory pool implementation with canary objects. These are defined in the *MemPatrol* library by the `canary_t` type. The monitored application is responsible for registering all its canaries using the `patrol_register` function provided by our library. This integration effort is similar to what is required for using debugging solutions such as Google's AddressSanitizer [114] with custom memory allocation schemes.

In the current implementation, all canaries must be registered before the application enters its untrusted execution phase, signified by starting the monitor thread with the `patrol_start` function and dropping its privileges. Figure 6.3 illustrates the API used by the application to integrate with *MemPatrol*.

```
typedef int8_t canary_t[16]; // Data type for canaries

void patrol_init(void); // Called at system startup

// Used for registering canary locations
void patrol_register(void *base, size_t stride, size_t count);

void patrol_start(int cpu); // Start monitoring
```

**Figure 6.3:** Canary-monitoring integration API

Upon calling the `patrol_register` function, the value of the base address as a 64-bit integer and the values of the pool's object size and object count, as 32-bit integers concatenated into a 64-bit integer, are stored in a table using two `sec64_t` entries generated using the `store_sec64` function. The monitor thread has not been started yet, so the key, generated by the `patrol_init` function on program startup, is temporarily stored in memory inside the *MemPatrol* library.

Once the `patrol_start` function is called, it loads the key into a register, zeroes out the copy in memory, and launches the monitor thread. The number of table entries is also kept in a register to prevent it from being tampered to trick the monitor into ignoring table entries.

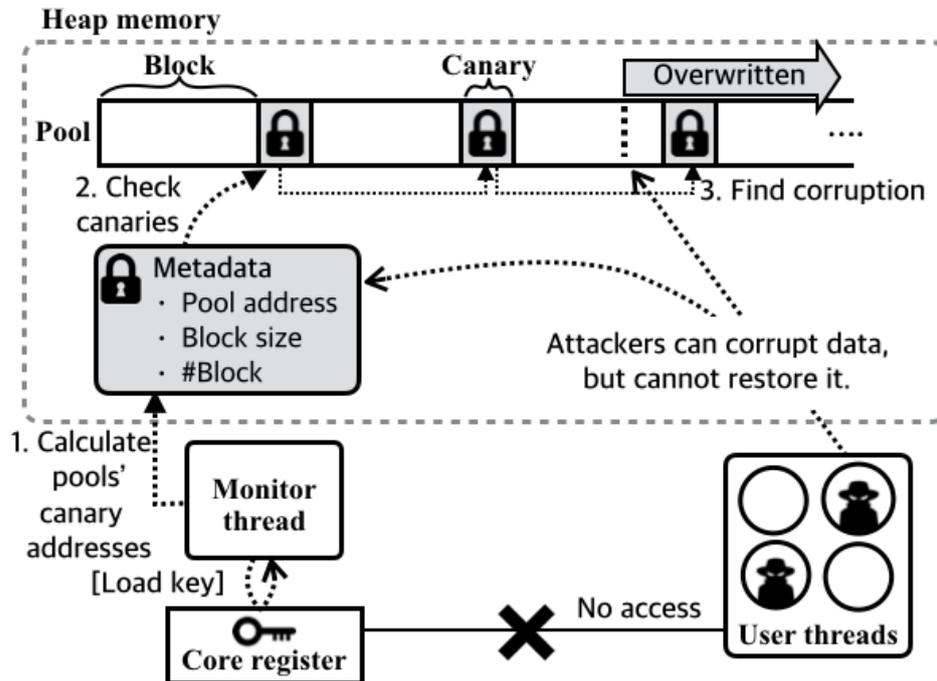


Figure 6.4: Secure canary checking

### 6.4.3 Cryptographically Generated Canary Values

Some existing approaches [95] using random-valued canaries safely store the original copies in the kernel or a hypervisor. In our technique, we generate canaries using cryptographic techniques to prevent attackers from inferring the original canary values and recovering them to prevent detection after a compromise. We use 128-bits for a canary, storing a MAC of the address of the canary. Unlike using random canary values, this does not require storing copies of canary values for comparison.

Since possibly-compromised threads of the application do not have access to the key, even if attackers succeed in exploitation through heap buffer overflows, they cannot recover the overwritten canary’s expected value. The overall checking procedure is illustrated in Figure 6.4.

We place one canary at the end of each block. Memory blocks are typically padded to match a requested alignment. This has to be done after the addition of the canary size to the allocation size. There is a choice on whether to place the canary back-to-back with the actual object, or to align the canary in memory. We chose to pack canaries tightly, to detect even

small, accidental heap buffer overflows and to save memory, at the cost of unaligned memory accesses from the monitor thread.

#### 6.4.4 Canary Recycling

The integrity monitor does not have to track the life cycle of each heap buffer. This is possible since the location and values of canaries are fixed throughout the execution of the program, thanks to the fixed size of pool elements. This allows us to setup all canaries during the memory pool's initialization, and avoid updates on every individual block's deallocation and reuse.

With such canary recycling, blocks with a corrupted canary may be returned to the pool before being checked by the monitor, and later re-allocated. The monitor, however, will eventually inspect the canary and detect the violation, even if the affected objects have been deallocated.

Canary recycling eliminates the communication overheads, but on the other hand, this approach incurs the burden of scanning all blocks of the memory pool, whether they are currently occupied or not. This has the effect of increasing the detection delay but is not a serious problem with appropriately provisioned memory pools.

## 6.5 Evaluation

We evaluated *MemPatrol's* performance by running in alongside it with NCORE [96], a proprietary Deep Packet Inspection (DPI) system, and running experiments using high bandwidth network traffic.

### 6.5.1 Integration with NCORE

We modified NCORE's memory pool library to reserve 16 bytes for one canary at the end of each block, and to call `patrol_register` when the memory pool is created to register all its canaries. Each canary also helps protect against buffer underflows in the next block. NCORE does not store allocator metadata between blocks, but stores free-list pointers at the start of free blocks. These are protected since canaries are active even when

their block is free. Finally, we added a call to the *MemPatrol* initialization routine (`patrol_init`) at the beginning of NCORE's startup, and a call to *MemPatrol*'s monitor thread launching routine (`patrol_start`) after the NCORE has initialized but before it drops privileges. The system's memory pools are initialized between these two calls.

## 6.5.2 Experimental Results

We ran NCORE on an iXsystems Mercury server with 2 Intel(R) Xeon(R) E5-2690 v4 CPUs, nominally clocked at 2.60 GHz (but running at 3.20 GHz thanks to Turbo Boost) with HyperThreads enabled and 256 GiB of RAM at 2133 MHz distributed over all four memory channels of the two CPUs. This system has 56 logical cores, out of which NCORE was configured to use 41 logical cores for pattern matching, and 5 logical cores for packet capture and internal load balancing, with their sibling logical cores left idle to avoid interference. One logical core on the first CPU was assigned to the *MemPatrol* monitor thread, and one physical core per CPU (4 logical cores in total) was left for general purpose use, such as user shells and OS services. We configured NCORE for a pattern matching workload inspecting all network traffic against a list of 5 million random substring patterns with an average pattern length of 100 bytes.

### Space Overhead

After launch, NCORE had registered 203 million heap canaries in 120 contiguous ranges. Thanks to memory pools, the overhead of metadata kept for each contiguous range of canaries is low. Also, the system used 129 GB of heap memory for objects and their canaries. Thus the average object size without canaries is 619 bytes, and the 16 bytes used for each canary amount to a memory overhead of 3.25 GB or 2.58%.

### CPU Overhead

We used another iXsystems server as a traffic generator replaying a 650 MB real traffic trace with a tool that rewrites the IP addresses on the fly to simulate an unlimited supply of flows. To evaluate the performance overhead,

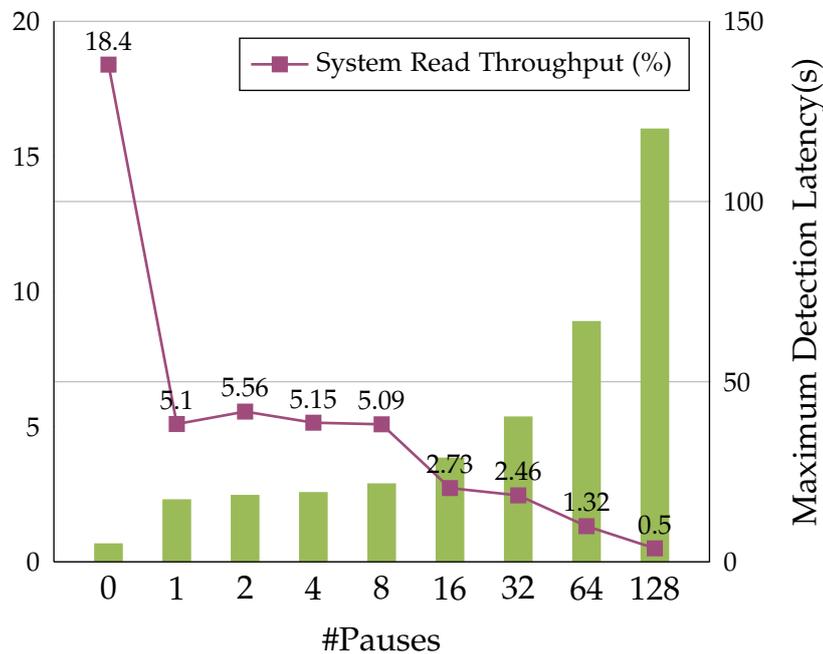
**Table 6.1:** Cache hit rates for different types of application threads and different temporal locality hints used by the monitor thread.

	RX		Workers		Patrol	
	L3 Hit	L2 Hit	L3 Hit	L2 Hit	L3 Hit	L2 Hit
No Patrol	0.31	0.10	0.64	0.66		
Prefetch 0	0.31	0.10	0.62	0.66	0.00	0.00
No Prefetch	0.30	0.10	0.59	0.66	0.00	0.00
Prefetch 3	0.30	0.10	0.60	0.66	0.00	0.00

we generated traffic at the rate of 50 Gb/s at 9.3 M packets/s and 170 K bidirectional flows/s. Under this load, the baseline NCORE without *MemPatrol* had 77% CPU utilization on the pattern matching cores. The traffic capture cores are constantly polling a network interface so are always at 100% utilization irrespective of actual load. There was no packet loss observed in this baseline setup. We repeated the experiment with the monitor thread running, and observed no increase in CPU utilization, with packet loss also remaining zero. By running on a separate core and performing non-temporal memory accesses, *MemPatrol* did not interfere with the instruction count of the application’s processing threads.

### Cache Overhead

We used the Intel Performance Counter Monitor (PCM) tool [138] to measure the cache hit rates on each logical core. The results are shown in Table 6.1. We show separate results for the traffic capture threads (RX), the pattern matching threads (Workers) and the monitor thread (Patrol). The first row is the baseline without the monitor thread. The second row shows the results with the monitor thread running and using non-temporal memory accesses. We observe that there is a small decrease of the L3 cache hit rate. If we disable the non-temporal memory access hinting, or instead we specify a high degree of temporal locality using the value 3 to the third argument of the GCC prefetch intrinsic, we measure a slightly higher degradation. We further get confirmation that it worked adding the hints for non-temporal memory access to the monitor thread by observing that its cache hit rate is zero.



**Figure 6.5:** Relation between system read throughput and maximum detection latency (time to scan all canaries). The user can select the trade-off by controlling the number of pause instructions inserted to throttle the monitor thread.

### Memory Bandwidth Overhead

Subsequently, we used the PCM tool to measure the system memory throughput. The measurement was done over a 60 second interval to smooth out variations. As expected, there was no impact on the memory write throughput, but we could observe the effects of the patrol thread on the system’s memory read throughput: An 18.1% increase over the baseline read throughput of 15.5 GB/s.

### Detection Delay

Running at full speed, *MemPatrol* required 5 seconds to scan all 203 million canaries. This corresponds to the worst-case detection delay after the corruption of a canary. We confirmed the detection capability end-to-end by introducing an artificial buffer overflow.

**Table 6.2:** Effects of the NUMA placement of the monitor thread on the detection latency and local/remote memory bandwidth of the monitor thread’s core.

	Monitored Sockets		
	Both	Local	Remote
#Canaries	203,159,920	76,184,970	126,974,950
Scan Duration ( $\mu$ s)	5,151,152	1,865,777	3,553,998
#Canaries/ $\mu$ s	39.4	40.8	35.7
Remote Memory Bandwidth	1,576	0	3,339
Local Memory Bandwidth	915	3,399	0.1

### Overhead Control

Next, we ran experiments to demonstrate control of the overhead by trading-off detection latency. We slowed down the monitor thread by a configurable amount by adding `pause` hardware instructions (via the `_mm_pause` compiler intrinsic). Figure 6.5 illustrates the effect of different delays determined by the number of additional `pause` instructions executed in each iteration of *MemPatrol*’s monitoring loop that is checking one canary. Insertion of a single delay instruction results in a sharp drop of the read throughput overhead from 18.1% to 5.2% and a roughly proportional increase in detection latency from 5 to 17.7 seconds. By further tweaking the number of `pause` instructions we can bring down the memory throughput overhead to 0.65% for an increased detection delay of 120 seconds. This experiment confirms that the user can decide the amount of overhead that is acceptable for the application, at the cost of detection delay. At the same time, the design of the system does not allow an attacker to avoid detection by exploiting the detection delay introduced for performance reasons.

### NUMA Effects

Modern multi-processor systems employ non-uniform memory access (NUMA). We investigated the performance effects of the CPU socket placement of the monitor thread. In the baseline setup, we use a single monitor thread running on socket 0 to monitor memory on both NUMA sockets. We wish to evaluate the performance of a monitor thread only inspecting

local memory on the socket that it is running. In Table 6.2 we compare this against the default setup inspecting both NUMA nodes, but also against an artificially suboptimal setup where a monitor thread inspects memory on the remote socket only. The number of canaries on each socket is different, because the second socket is running more worker threads that maintain more state compared to RX threads. We normalize this by reporting the number of canaries inspected per unit of time, and observe that the difference, while matching our expectations in quality, is not significant. The reason must be that remote memory accesses suffer significantly in terms of latency, but not throughput. That is of course as long as the interconnect between the CPUs is not overloaded. This is not the case in our experiments, but we can observe the effects of NUMA placements on the interconnect by showing the local vs. the remote memory traffic. We can see that with the optimal NUMA placement there is no remote memory traffic for the core running the monitor thread. This would motivate using multiple monitor threads, one on each NUMA node, inspecting only local memory.

## 6.6 Discussions

### 6.6.1 Tunable Overhead

*MemPatrol* offers developers and operators control over its runtime overhead. A similar idea was pursued by ASAP [132], which automatically instruments the program to maximize its security while staying within a specified overhead budget. Unlike *MemPatrol*, ASAP controls the overhead by decreasing coverage, while *MemPatrol* achieves this by increasing detection delay without compromising eventual detection. Cruiser [148] also discussed a possible approach to increase its efficiency using back-off strategies to pace the monitor thread with `nop` instructions or `sleep` calls. However, without proper isolation of the monitor thread, this approach undermines Cruiser’s security guarantees.

Some KIMs tackle transient attacks by snooping the bus traffic using PCI cards [83] or even commodity GPGPUs [64]. *MemPatrol* is quite similar to KIMs using periodic memory inspection, but monitors applications, so does not require a hypervisor or dedicated hardware.

### 6.6.2 Memory Safety

The case-study of *MemPatrol* for heap memory integrity is based on heap-canary countermeasures, and specifically those that use a monitor running in parallel with the protected application [125, 148]. Note, however, the proprietary network traffic identification system (NCORE) we evaluated, also uses stack-based overflow protections on top of *MemPatrol*, and the mere use of memory pools also offers some level of protection against temporal-safety violations through the reuse of memory only for objects with identical layout, which can prevent most abuses of function pointer fields [4].

Finally, it is worth comparing *MemPatrol* with inline solutions offering similar security guarantees. For example, inline heap canary checking [108] can be very efficient, but suffers from unbounded detection delay, as detection relies on checks triggered by events such as deallocations. In the case of NCORE [96], a passive network traffic identification system on which *MemPatrol* was evaluated, heap allocations for certain objects such as host state may linger for several days. *MemPatrol*, on the other hand, puts a bound on the detection delay. Other inline solutions detect buffer overflows immediately by instrumenting every memory write to check whether it overwrites a canary. WIT [5], for example, uses this approach, which contributes the bulk of its runtime overhead of 4–25% for CPU bound benchmarks, which is prohibitive for some performance-critical applications.

## 6.7 Conclusion

In summary, this work applied an integrity monitoring solution, *MemPatrol*, to a high-performance network monitoring system, demonstrating 0% CPU-time overhead for the application’s threads. The hidden memory bandwidth overheads also concerned us, but *MemPatrol* demonstrated how to minimize them to under 1%. We conclude with some remarks on current limitations and future directions.

Importantly, this case study’s memory safety guarantees are limited to heap buffer overflow detection, and can be thwarted by memory disclosure

attacks. Future work is required to identify additional memory integrity applications.

Moreover, the current *MemPatrol* prototype cannot register additional memory locations to monitor after initialization, but this limitation is not fundamental. We could intermix monitoring with processing of registration requests received through a message queue. As long as bookkeeping data structures are append-only, the threat of replay attacks is averted. A general solution for supporting arbitrary updates, however, is an interesting future direction.

Creating the binary code for a system like *MemPatrol* is currently a tedious, manual process. As pointed out in Loop-Amnesia [117], some level of compiler support, e.g. to control register spilling, would help.

Finally, the full security of AES may be overkill given the bound on detection latency, and lowering the number of AES rounds used could be considered as a way to increase the monitor thread's performance.

# 7

## Conclusions

---

This research presented MIU, Memory Integrity Utilities, including both an inline and sideline monitoring system for memory safety. MIU focuses on the cost of memory safety and the solutions for possible practical deployment of the enforcement.

*MemPatrol* adopted concurrency for the minimal performance to the level of production deployment. Running a concurrent monitor realised 0% CPU-time overhead for applications' threads and configurable overheads that is useful for systems whose workloads changes over the time. The support of synchronous communication may bring more run-time overheads for synchronisation between a monitor thread and user threads, so *MemPatrol* chose asynchronous communication like many other concurrent models. The challenge of an asynchronous concurrent monitor is the time gap between memory corruption and its detection. This work proposed and implemented a monitor isolation technique using registers as the monitor's private memory.

FRAMER and spaceMiu are compiler-assisted, inline monitoring systems based on a tagged pointer-capability model with object granularity. These systems exploit the currently unused significant 16 bits of pointers to store a tag. The complete and generic encoding derives a metadata pointer from a pointer to any kind of objects and this removes the consumption of memory for padding or alignment, that benefits all levels of memory hierarchy. It also demonstrates excellent D-cache performance. This approach tolerates the increase in dynamic instruction used for arithmetic operations for the efficiency in both memory footprint and D-cache hits. This provides great advantages for hardware implementation, since the overheads

for arithmetic operations will be largely resolved with ISA, whereas it is difficult for memory footprint and D-cache hits.

This capability model is evaluated on memory/type safety – array out-of-bounds and type confusion checking. One of the main focuses is to cover all kinds of spatial memory corruptions, while keeping the efficiency in memory. Firstly, this work showed how frame encoding can be used for bounds checking guaranteeing near-zero negatives caused by violation of intended referents that challenged previous memory safety solutions with per-object metadata storage. Secondly, this capability model proposed a framework for near-complete memory safety detecting internal overflows and unsafe type casts with the support of additional descriptors.

For the future design, some parts of the designs can be improved. This work sacrifices dynamic instruction counts for memory efficiency and complete memory safety, since the overhead from arithmetic operations are the easiest to resolve with hardware acceleration. However, this overhead still needs to be reduced for software-based solutions. This work proved the small increase in memory overheads, so we expect to reduce the run-time overhead with more memory overhead by re-arranging objects. Derivation of the header address of large-framed objects requires additional arithmetic operations and more importantly the objects causes cache misses through indirect access through the supplementary table. Minimising or even removing, if possible, the large-framed objects can also resolve the overheads for branches.

This capability design could benefit other programming language security mechanisms as well above C/C++, and has a wide range of applications such as thread safety or garbage collection that requires mapping an arbitrary to metadata.

# 8

## Appendix

---

### 8.1 Proofs

#### 8.1.1 Proof 1

Given an object  $o$  and its wrapper frame  $f$ , let's assume there exists a smaller frame  $x$  that has  $o$  inside. Since  $o$  resides in both  $f$  and  $x$ , we can conclude that  $x$  is a subframe of  $f$ . According to the assumption, the base address of  $o$  ( $base_o$ ) is within the range of  $x$ , hence, we get  $base_x \leq base_o$ . Here,  $f$  is  $o$ 's wrapper frame, so  $base_o$  is placed in  $f$ 's lower subframe.  $x$  is a subframe of  $f$ , hence  $x$  must be  $f$ 's lower subframe. This is resolved to contradiction between the assumption ( $x$  has  $o$  inside) and the definition of wrapper function ( $o$ 's upper bound in the upper subframe). Hence, we can conclude that there is no smaller frame than  $o$ 's wrapper frame; this is actually the unique wrapper frame, and it can be used as a reference point.

#### 8.1.2 Proof 2

We prove that for each  $N$ , there exists at most one  $N$ -object mapped to each entry of a division array, and show  $N$  identifies an object mapped to the same division array. To prove this, we assume there exist two distinctive objects,  $x$  and  $y$ ; both are  $N$ -objects ( $N \geq 16$ ) mapped to the same division array. Since  $x$  and  $y$  are  $N$ -objects, their wrapper frame ( $f_x$  and  $f_y$ ) is  $2^N$ -sized by definition. The division is the only one that  $f_x$  and  $f_y$  are mapped to as shown previously, so  $f_x$  and  $f_y$  have the same base address as the division. In addition, both frames have the same size, so they are

identical. Both base addresses of  $x$  and  $y$  ( $b_x, b_y$ ) must be in the lower  $(N - 1)$ -subframe of  $f_x$  (or  $f_y$ ), and end addresses must be in the other sub-frame. From this,  $b_x$  and  $b_y$  must be smaller than  $e_x$  and  $e_y$ . However, the objects are distinct, so  $b_x < e_x < b_y < e_y$  or vice versa must hold. The assumption leads to a contraction. We conclude that for each  $N$ , there is a unique  $N$ -object mapped to one division array.

# Bibliography

---

- [1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proc. of ACM CCS*, pages 340–353, 2005. Cited on pages 19, 46, 47, 21, 48, and 49.
- [2] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit, 2007. Cited on pages 36, 21, and 38.
- [3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. Cited on pages 40 and 42.
- [4] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security '10*, Washington, DC, USA, 2010. Cited on pages 20, 21, 36, 37, 76, 145, 22, 23, 38, 39, and 78.
- [5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *IEEE S&P '08*, Oakland, CA, USA, 2008. Cited on pages 20, 21, 32, 46, 47, 145, 22, 23, 34, 48, and 49.
- [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 Conference on USENIX Security Symposium, SSYM'09*, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association. Cited on pages 20, 21, 31, 32, 33, 44, 58, 74, 88, 22, 23, 34, 35, 46, 61, 76, and 89.
- [7] AlephOne. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996. Cited on pages 47 and 49.
- [8] ARM Limited. Arm a64 instruction set architecture, 2018. [https://static.docs.arm.com/ddi0596/a/DDI\\_0596\\_ARM\\_a64\\_](https://static.docs.arm.com/ddi0596/a/DDI_0596_ARM_a64_)

- instruction\_set\_architecture.pdf. Cited on pages 21, 34, 37, 39, 23, 36, and 41.
- [9] ARM Limited. Armv8.5-a, 2018. <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a>. Cited on pages 21, 31, 34, 37, 39, 89, 22, 23, 33, 36, 41, and 91.
- [10] Todd Austin. Pointer-Intensive Benchmark Suite, Sept. 1995. Cited on pages 79 and 81.
- [11] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *ACM PLDI '94*, Orlando, FL, USA, 1994. Cited on pages 20, 30, 43, 92, 22, 31, 32, 46, and 94.
- [12] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions. *ACM Comput. Surv.*, 48(1), 2015. Cited on pages 50 and 52.
- [13] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 321–333, New York, NY, USA, 2000. ACM. Cited on pages 89 and 91.
- [14] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. Cited on pages 27 and 29.
- [15] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016. Cited on pages 42, 47, 92, 44, 49, and 94.

- [16] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO 03, page 265275, USA, 2003. IEEE Computer Society. Cited on pages 26 and 28.
- [17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.*, 50(1), April 2017. Cited on pages 47 and 49.
- [18] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 381–392, New York, NY, USA, 2018. ACM. Cited on pages 44 and 46.
- [19] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-Time Type Checking for Binary Programs. In Görel Hedin, editor, *Compiler Construction*, pages 90–105, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. Cited on pages 45, 92, 47, and 94.
- [20] Martin C. Carlisle and Anne Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 29–38, New York, NY, USA, 1995. ACM. Cited on pages 79 and 81.
- [21] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. *SIGPLAN Not.*, 29(11):319327, November 1994. Cited on pages 37, 40, 39, and 42.
- [22] Satish Chandra and Thomas Reps. Physical Type Checking for C. In *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE 99, page 6675, New York, NY, USA, 1999. Association for Computing Machinery. Cited on pages 43, 93, 45, and 95.

- [23] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravisankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, SSYM'05*, page 12, USA, 2005. USENIX Association. Cited on pages 19 and 21.
- [24] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications, ISCC '06*, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society. Cited on pages 32 and 34.
- [25] David Chisnall, Colin Rothwell, Brooks Davis, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Peter G. Neumann, and Michael Roe. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 117–130, New York, NY, USA, 2015. ACM. Cited on pages 20, 30, 22, 31, and 32.
- [26] 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 *USENIX Security '98*, San Antonio, TX, USA, 1998. Cited on pages 26, 47, 49, 28, and 52.
- [27] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 763–780, 2015. Cited on page 126.
- [28] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 11*, page

- 4051, New York, NY, USA, 2011. Association for Computing Machinery. Cited on pages 27 and 29.
- [29] Jack B. Dennis and Earl C. Van Horn. *Programming Semantics for Multiprogrammed Computations*, 1965. Cited on pages 40, 57, 42, and 59.
- [30] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. *SIGPLAN Not.*, 43(3):103–114, March 2008. Cited on pages 30, 37, 38, 31, 32, 39, and 40.
- [31] David Dewey and Jonathon T. Giffin. Static detection of C++ vtable escape vulnerabilities in binary code. In *NDSS*. The Internet Society, 2012. Cited on pages 42, 92, 44, and 94.
- [32] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280, June 2006. Cited on pages 36, 76, 38, and 78.
- [33] 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 '06*, pages 162–171, New York, NY, USA, 2006. ACM. Cited on pages 20, 31, 32, 44, 23, 33, 34, and 46.
- [34] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 06*, page 144157, New York, NY, USA, 2006. Association for Computing Machinery. Cited on pages 78, 92, 80, and 94.
- [35] Gregory J. Duck and Roland H. C. Yap. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design*

- and Implementation*, PLDI 2018, pages 181–195, New York, NY, USA, 2018. ACM. Cited on pages 20, 31, 92, 22, 33, and 94.
- [36] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, 2018. Cited on pages 20, 41, 22, and 31.
- [37] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004. Cited on pages 20, 26, 22, and 28.
- [38] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard E. Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*, pages 781–796. IEEE Computer Society, 2015. Cited on pages 45 and 48.
- [39] Robert Gawlik and Thorsten Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proc. of ACSAC*, pages 396–405, 2014. Cited on pages 42 and 44.
- [40] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, pages 475–490. USENIX Association, 2012. Cited on pages 25 and 27.
- [41] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 575–589, 2014. Cited on pages 19, 25, 46, 21, 27, and 49.
- [42] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119, Austin, TX, 2016. USENIX Association. Cited on pages 45, 46, and 48.

- [43] GPLv2. Coccinelle. <http://coccinelle.lip6.fr/>. Cited on pages 110, 118, 112, and 120.
- [44] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP '15*, pages 3–19, Washington, DC, USA, 2015. IEEE Computer Society. Cited on page 126.
- [45] Shay Gueron. Intel Advanced Encryption Standard (AES) Instruction Set White Paper, Rev. 3.0 ed, 2010. Cited on pages 24, 51, 129, 26, and 53.
- [46] István Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical Type Confusion Detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 517–528, 2016. Cited on pages 41, 42, 44, 92, 110, 46, 94, and 111.
- [47] István Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. METAlloc: efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security, EUROSEC 2016, London, UK, April 18-21, 2016*, pages 5:1–5:6, 2016. Cited on pages 21, 32, 44, 58, 88, 23, 34, 46, 61, and 89.
- [48] Keith Harrison and Shouhuai Xu. Protecting cryptographic keys from memory disclosure attacks. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 137–143, 2007. Cited on page 126.
- [49] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. Cited on pages 79, 111, 81, and 113.

- [50] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring Operating System Kernel Integrity with OSck. In *ACM ASPLOS XVI*, Newport Beach, CA, USA, 2011. Cited on pages 50 and 52.
- [51] Intel. Data Plane Development Kit (DPDK). Cited on page 136.
- [52] Intel. MPX Performance Evaluation. Cited on pages 27, 48, 29, and 50.
- [53] Intel Corporation. Introduction to Intel<sup>®</sup> memory protection extensions, 2013. <http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions>. Cited on pages 20, 21, 30, 37, 38, 43, 88, 22, 23, 31, 32, 39, 40, 46, and 90.
- [54] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proc. of NDSS*, 2014. Cited on pages 42 and 44.
- [55] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *ACM CCS '13*, Berlin, Germany, 2013. Cited on pages 49 and 51.
- [56] Yuseok Jeon, Priyam Biswas, Scott A. Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2373–2387, 2017. Cited on pages 41, 42, 110, 44, and 111.
- [57] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. Cited on pages 20, 30, 43, 92, 22, 31, 32, 46, and 94.
- [58] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *ACM AADEBUG97 '97*, 1997. Cited on pages 31, 32, 74, 33, 34, and 76.

- [59] Stephen Kell. Towards a Dynamic Object Model Within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 224–239, New York, NY, USA, 2015. ACM. Cited on pages 96, 44, and 98.
- [60] Stephen Kell. Dynamically Diagnosing Type Errors in Unsafe Code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, pages 800–819, New York, NY, USA, 2016. ACM. Cited on pages 41, 42, 45, 92, 110, 44, 47, 94, and 112.
- [61] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. Cited on pages 96 and 98.
- [62] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 09*, page 207220, New York, NY, USA, 2009. Association for Computing Machinery. Cited on pages 40, 41, 42, and 43.
- [63] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17*, pages 437–452, New York, NY, USA, 2017. ACM. Cited on pages 30 and 32.
- [64] Lazaros Koromilas, Giorgos Vasiliadis, Elias Athanasopoulos, and Sotiris Ioannidis. GRIM: Leveraging GPUs for Kernel Integrity Monitoring. In *RAID '16*, Paris, France, 2016. Cited on pages 50, 144, and 52.
- [65] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta Pointers: Buffer Overflow Checks

- Without the Checks. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 22:1–22:14, New York, NY, USA, 2018. ACM. Cited on pages 31, 34, 33, and 36.
- [66] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys 17, page 205221, New York, NY, USA, 2017. Association for Computing Machinery. Cited on pages 30, 31, 34, 32, 33, and 36.
- [67] Kuznetsov, Volodymyr and Szekeres, László and Payer, Mathias and Candea, George and Sekar, R. and Song, Dawn. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association. Cited on pages 45 and 47.
- [68] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS 13, page 721732, New York, NY, USA, 2013. Association for Computing Machinery. Cited on pages 44, 57, 58, 88, 46, 59, 61, and 89.
- [69] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 81–96, Washington, D.C., August 2015. USENIX Association. Cited on pages 41, 44, 110, and 111.
- [70] LLVM Team. Hardware-assisted AddressSanitizer Design Documentation. <http://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html>. Cited on pages 57 and 59.

- [71] LLVM Team. LLVM Constant Expression, 2003. <https://llvm.org/docs/LangRef.html#constant-expressions>. Cited on pages 70 and 72.
- [72] LLVM Team. LLVM First Class Type, 2003. <https://llvm.org/docs/LangRef.html>. Cited on pages 93 and 95.
- [73] LLVM Team. Standard C Library Intrinsic, 2003. <https://llvm.org/docs/LangRef.html#standard-c-library-intrinsics>. Cited on pages 72 and 73.
- [74] LLVM Team. The LLVM Gold Plugin, 2003. <https://llvm.org/docs/GoldPlugin.html>. Cited on pages 69, 107, 71, and 109.
- [75] LLVM Team. LLVM Control Flow Integrity, 2016. <https://clang.llvm.org/docs/ControlFlowIntegrity.html>. Cited on pages 19, 46, 47, 21, 48, and 49.
- [76] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. Debugging via run-time type checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, FASE 01, page 217232, Berlin, Heidelberg, 2001. Springer-Verlag. Cited on pages 45 and 47.
- [77] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190200, June 2005. Cited on pages 26 and 28.
- [78] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *ACM CCS '15*, Denver, CO, USA, 2015. Cited on pages 51, 135, and 53.
- [79] Microsoft. Accessing Read-Only System Memory, 2018. <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/accessing-read-only-system-memory>. Cited on pages 26 and 28.

- [80] Microsoft. No-Execute (NX) Nonpaged Pool, 2018. <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/nx-pool-opt-in-mechanisms>. Cited on pages 26 and 28.
- [81] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined Symbolic Taint Analysis. In *USENIX Security '15*, Washington, DC, USA, 2015. Cited on pages 21, 48, 49, 23, 50, and 51.
- [82] MITRE. CWE Top 25 Most Dangerous Software Weaknesses, 2020. <http://cwe.mitre.org/top25>. Cited on pages 19 and 21.
- [83] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *ACM CCS '12*, Raleigh, NC, USA, 2012. Cited on pages 50, 144, and 52.
- [84] James H. Morris. Protection in programming languages. *Commun. ACM*, 16(1):1521, January 1973. Cited on pages 40 and 42.
- [85] Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security '11*, San Francisco, CA, USA, 2011. Cited on pages 50 and 53.
- [86] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*, pages 175:175–175:184, New York, NY, USA, 2014. ACM. Cited on pages 36 and 38.
- [87] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything You Want to Know About Pointer-Based Checking. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 190–208, Dagstuhl,

- Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Cited on pages 21, 29, 30, 43, 88, 23, 31, 32, 46, and 90.
- [88] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. volume 45, pages 31–40, New York, NY, USA, June 2010. ACM. Cited on pages 20, 21, 36, 76, 22, 23, 38, and 78.
- [89] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, pages 245–258, New York, NY, USA, 2009. ACM. Cited on pages 21, 30, 33, 36, 43, 44, 58, 89, 23, 31, 32, 35, 38, 46, 61, and 91.
- [90] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. FRAMER: Fast Per-object Metadata Management for Memory Safety. Paper presented at the Annual Arm Research Summit, 2018. Cited on pages 23, 53, 58, and 25.
- [91] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19*, page 612626, New York, NY, USA, 2019. Association for Computing Machinery. Cited on pages 23, 53, 58, and 25.
- [92] Myoung Jin Nam, Wonhong Nam, Jin-Young Choi, and Periklis Akritidis. MemPatrol: Reliable Sideline Integrity Monitoring for High-Performance Systems. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 48–69, Cham, 2017. Springer International Publishing. Cited on pages 23, 54, 123, and 25.
- [93] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

- Cited on pages 20, 23, 30, 33, 42, 43, 54, 92, 93, 22, 25, 31, 32, 35, 44, 45, 46, 56, 94, and 95.
- [94] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89100, June 2007. Cited on pages 20, 26, 32, 36, 28, 34, and 38.
- [95] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. HeapSentry: Kernel-assisted Protection Against Heap Overflows. In *GI DIMVA '13*, Berlin, Germany, 2013. Cited on pages 50, 138, 52, and 137.
- [96] Niometrics. NCORE DPI System. Cited on pages 124, 126, 127, 139, and 145.
- [97] Ben Niu and Gang Tan. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 577–587, New York, NY, USA, 2014. ACM. Cited on pages 46 and 48.
- [98] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *USENIX Security '16*, Austin, TX, USA, 2016. Cited on pages 49 and 51.
- [99] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2):28:1–28:30, June 2018. Cited on pages 30, 88, 32, and 90.
- [100] Oracle Corporation. Introduction to SPARC M7 and Silicon Secured Memory (SSM), 2016. [https://swisdev.oracle.com/\\_files/What-Is-SSM.html](https://swisdev.oracle.com/_files/What-Is-SSM.html). Cited on pages 30 and 32.
- [101] PaX. MPROTECT, 2003. Cited on page 132.
- [102] PaX Team. Address Space Layout Randomization (ASLR), 2003. <http://pax.grsecurity.net/docs/aslr.txt>. Cited on pages 19, 25, 21, and 27.

- [103] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. *SIGPLAN Not.*, 46(7):157168, March 2011. Cited on pages 26, 27, 28, and 29.
- [104] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security '04*, San Diego, CA, USA, 2004. Cited on pages 50 and 52.
- [105] Marina Polishchuk, Ben Liblit, and Chloë W. Schulze. Dynamic heap type inference for program understanding and debugging. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 39–46. ACM, 2007. Cited on pages 45 and 47.
- [106] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proc. of NDSS*, 2015. Cited on pages 46 and 48.
- [107] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A Defense against Heap-Spraying Code Injection Attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09*, page 169186, USA, 2009. USENIX Association. Cited on page 135.
- [108] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time Detection of Heap-based Overflows. In *USENIX LISA '03*, San Diego, CA, USA, 2003. Cited on pages 50, 145, and 52.
- [109] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '08*, New York, NY, USA, 2008. ACM. Cited on pages 21, 48, 23, and 50.

- [110] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *IN PROCEEDINGS OF THE 11TH ANNUAL NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM*, pages 159–169, 2004. Cited on pages 32 and 34.
- [111] 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 European Conference on Computer Systems, EuroSys '09*, pages 33–46, New York, NY, USA, 2009. ACM. Cited on pages 21, 48, 23, 50, and 51.
- [112] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):3050, February 2000. Cited on pages 20, 26, 22, and 28.
- [113] SELinux Wiki. Main Page — SELinux Wiki, 2013. Cited on page 132.
- [114] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12*, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association. Cited on pages 20, 21, 23, 31, 32, 33, 36, 44, 58, 88, 137, 22, 29, 34, 35, 38, 46, 61, and 89.
- [115] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. *SIGOPS Oper. Syst. Rev.*, 33(5):170185, December 1999. Cited on pages 40 and 42.
- [116] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with Type Casts in C. *SIGSOFT Softw. Eng. Notes*, 24(6):180–198, October 1999. Cited on pages 54, 92, 93, 56, 94, and 95.
- [117] Patrick Simmons. Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption. In *ACM ACSAC '11*, Orlando, FL, USA, 2011. Cited on pages 51, 129, 146, and 53.
- [118] Matthew S. Simpson and Rajeev K. Barua. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Softw.*

- Pract. Exper.*, 43(1):93–128, January 2013. Cited on pages 36, 76, 22, 38, and 78.
- [119] Stacy Simpson. SAFECode Whitepaper: Fundamental Practices for Secure Software Development 2nd Edition. In Helmut Reimer, Norbert Pohlmann, and Wolfgang Schneider, editors, *ISSE*, pages 1–32. Springer, 2014. Cited on pages 78 and 80.
- [120] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, May 2013. Cited on pages 19, 25, 47, 21, 27, and 49.
- [121] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96*, pages 32–41, New York, NY, USA, 1996. ACM. Cited on pages 90 and 92.
- [122] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy, SP '13*, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society. Cited on pages 19, 20, and 22.
- [123] Tian Tan, Yue Li, and Jingling Xue. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 278–291, New York, NY, USA, 2017. ACM. Cited on pages 90 and 92.
- [124] The Chromium Projects. UBSan. Cited on pages 42, 92, 110, 44, 94, and 111.
- [125] Donghai Tian, Qiang Zeng, Dinghao Wu, Peng Liu, and Changzhen Hu. Kruiser: Semi-synchronized Non-blocking Concurrent Kernel Heap Buffer Overflow Monitoring. In *ISOC NDSS '12*, San Diego, CA, USA, 2012. Cited on pages 21, 48, 50, 145, 23, and 52.

- [126] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proc. of USENIX SEC*, pages 941–955, 2014. Cited on pages 46, 47, 48, and 49.
- [127] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: a compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417, 2001. Cited on pages 47, 48, 49, and 50.
- [128] Victor van der Veen, Dennis Andriessse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proc. of ACM CCS*, pages 927–940, 2015. Cited on pages 42 and 44.
- [129] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova, editors, *Research in Attacks, Intrusions, and Defenses*, pages 86–106, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. Cited on pages 19 and 21.
- [130] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanassopoulos, and Cristiano Giuffrida. A Tough `call`: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proc. of IEEE S&P*, pages 934–953, May 2016. Cited on pages 42 and 44.
- [131] Vendicator. Stack Shield: A stack smashing technique protection tool for linux, 2000. Cited on pages 47 and 49.
- [132] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High System-Code Security with Low Overhead. In *IEEE S&P '15*, Oakland, CA, USA, 2015. Cited on page 144.
- [133] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP '93*,

- pages 203–216, New York, NY, USA, 1993. ACM. Cited on pages 50 and 52.
- [134] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. *CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization*. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, May 2015. Cited on pages 57 and 59.
- [135] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. Cited on pages 30, 37, 32, and 39.
- [136] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. *Secure dynamic code generation against spraying*. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, page 738740, New York, NY, USA, 2010. Association for Computing Machinery. Cited on page 135.
- [137] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and Its Operating System*. Elsevier, January 1979. Cited on pages 40, 41, 42, and 43.
- [138] Thomas Willhalm, Roman Dementiev, and Patrick Fay. *Intel Performance Counter Monitor - A better way to measure CPU utilization*, 2012. Cited on page 141.
- [139] Emmett Witchel, Junghwan Rhee, and Krste Asanović. *Mondrix: Memory isolation for linux using mondriaan memory protection*.

- SIGOPS Oper. Syst. Rev.*, 39(5):3144, October 2005. Cited on pages 37 and 39.
- [140] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA '14: Proceeding of the 41st annual international symposium on Computer architecture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press. Cited on pages 40, 57, 42, and 59.
- [141] Dinghao Wu, Peng Liu, Qiang Zeng, and Donghai Tian. Software Cruising: A New Technology for Building Concurrent Software Monitor. In *Secure Cloud Computing*. 2014. Cited on pages 20, 48, 22, and 50.
- [142] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array Bounds Check Elimination for the Java HotSpot™ Client Compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ '07*, pages 125–133, New York, NY, USA, 2007. ACM. Cited on pages 89 and 91.
- [143] Wei Xu, Daniel C. DuVarney, and R. Sekar. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, SIGSOFT '04/FSE-12*, pages 117–126, New York, NY, USA, 2004. ACM. Cited on pages 21, 30, 23, and 32.
- [144] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE S&P '09*, Oakland, CA, USA, 2009. Cited on pages 50 and 52.
- [145] Suan Hsi Yong and Susan Horwitz. Reducing the overhead of dynamic analysis. *Electron. Notes Theor. Comput. Sci.*, 70(4):158–178, 2002. Cited on pages 45 and 47.

- [146] Suan Hsi Yong and Susan Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11*, page 307316, New York, NY, USA, 2003. Association for Computing Machinery. Cited on pages 20, 21, 22, and 23.
- [147] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: An efficient pointer arithmetic checker for C programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS '10*, pages 145–156, New York, NY, USA, 2010. ACM. Cited on pages 31, 32, 33, and 34.
- [148] Qiang Zeng, Dinghao Wu, and Peng Liu. Cruiser: Concurrent Heap Buffer Overflow Monitoring Using Lock-free Data Structures. In *ACM PLDI '11*, San Jose, CA, USA, 2011. Cited on pages 21, 48, 49, 144, 145, 23, 50, and 51.
- [149] Chao Zhang, Dawn Xiaodong Song, Scott A. Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. VTrust: Regaining Trust on Virtual Calls. In *NDSS*, 2016. Cited on pages 19, 42, 46, 92, 21, 44, 48, and 94.
- [150] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proc. of IEEE S&P*, pages 559–573, 2013. Cited on pages 19, 46, 21, and 48.
- [151] Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX SEC*, pages 337–352, 2013. Cited on pages 19, 21, and 48.