PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Inria Paris

Memory Allocators: Formal Verification and Evaluation

Vérification formelle et évaluation d'allocateurs mémoire

À soutenir par

Antonin REITZ

Le 5 décembre 2025

École doctorale n°386

Sciences Mathématiques de Paris Centre

Spécialité

Informatique

Composition du jury :

Jean-Christophe FILLIÂTRE Directeur de recherche CNRS	*Rapporteur*
KC SIVARAMAKRISHNAN Assistant Professor IIT Madras/Tarides	*Rapporteur*
Delphine DEMANGE Maître de conférences Université de Rennes	*Examinatrice*
Xavier RIVAL Directeur de recherche Inria	*Examinateur*
Bruno BLANCHET Directeur de recherche Inria	*Directeur de thèse*
Aymeric FROMHERZ Inria Starting Faculty Position Inria	*Co-encadrant de thèse*
Jonathan PROTZENKO Tech Lead Manager Google	*Membre invité*

Abstract

Software has become pervasive in our lives. Most commonly-used software, including web browsers, instant messaging applications, office suites as well as video-conferencing tools, routinely handles memory to store various contents, such as pictures, music, videos, PDF documents. The *memory allocator* is the software component providing such user applications with memory. Unfortunately, user applications do not always handle provided memory correctly; in fact, commonly-used software is plagued by a class of security issues closely related to memory management: *memory safety issues*. To mitigate such issues, it is possible to use *hardened memory allocators*, that provide memory to applications while mitigating memory safety issues and thus must satisfy heterogeneous goals. In turn, ensuring that these delicate software components are reliable and actually correct is desirable, prompting the use of *formal methods*, that can be leveraged to increase the trust one can place into software.

In this thesis, we present a methodology to prove hardened memory allocators functionally correct, that we used to develop StarMalloc, a verified, efficient, hardened, and concurrent memory allocator. Using the Steel separation logic framework, we show how to specify and verify a variety of low-level patterns and delicate security mechanisms, by relying on a combination of dependent types, SMT, and modular abstractions to enable efficient, iterative verification.

We produce a verified artifact, in C, that implements the entire API surface of an allocator, and as such works as a drop-in replacement for real-world projects, notably the Firefox browser. We then evaluate StarMalloc and show that it exhibits competitive performance by evaluating it against state-of-the-art memory allocators, and against a variety of real-world projects. Finally, aiming at gaining more understanding about memory allocators behaviors, we developed a hardware-based heap tracing prototype.

Résumé

Le logiciel est devenu omniprésent dans la vie quotidienne. La plupart des logiciels couramment utilisés, tels que les navigateurs Web, les applications de messagerie instantanée, les suites bureautiques ainsi que les outils de visioconférence, s'appuient sur la mémoire pour traiter des contenus variés, tels que des images, de la musique, des vidéos ou encore des documents au format PDF. L'*allocateur mémoire* est le composant logiciel qui est chargé de fournir à ces logiciels de la mémoire lors de leur exécution. Malheureusement, les applications utilisateurs ne manipulent pas toujours la mémoire correctement. Dans les faits, de tels logiciels communément utilisés sont affectés par une classe de problèmes de sécurité étroitement liés à la gestion de la mémoire, appelés *problèmes de sûreté mémoire*. Pour limiter les impacts de ces problèmes, il est possible d'utiliser des allocateurs mémoire dits *renforcés*, chargés à la fois de subvenir aux besoins en mémoire des applications utilisateurs ainsi que de pallier les problèmes de sûreté mémoire de celles-ci. Compte tenu de leur rôle crucial, il est souhaitable de s'assurer que ces briques logicielles délicates sont fiables et correctes, ce qui motive l'emploi de *méthodes formelles* qui permettent de placer davantage de confiance dans les logiciels auxquels elles sont appliquées.

Dans le cadre de cette thèse, nous présentons une méthodologie pour prouver la correction fonctionnelle d'allocateurs mémoire renforcés, que nous utilisons pour le développement de StarMalloc, un allocateur mémoire vérifié, efficace, renforcé et concurrent. En utilisant le cadriciel de vérification Steel s'appuyant sur la logique de séparation, nous montrons comment spécifier et vérifier un ensemble varié de structures bas niveau et de mécanismes de sécurité complexes. Pour ce faire, nous nous appuyons sur la combinaison de types dépendants, de l'usage d'un solveur SMT et d'abstractions modulaires, qui permet un processus de vérification efficace et itératif.

Nous produisons un artefact vérifié, en C, qui implémente l'entièreté de l'interface de programmation d'un allocateur et peut ainsi être utilisé comme tel pour des projets réalistes tels que le navigateur Firefox. Ensuite, nous évaluons StarMalloc et démontrons que ses performances sont compétitives en le comparant à d'autres allocateurs sur un ensemble de tests de performance réalistes. Enfin, dans le but de mieux comprendre le comportement des allocateurs mémoire, nous avons développé un prototype permettant de tracer les opérations de gestion de la mémoire sur le tas, l'ensemble s'appuyant sur des fonctions matérielles.

List of Contributions

List of Publications

Antonin Reitz, Aymeric Fromherz, and Jonathan Protzenko. "StarMalloc: Verifying a Modern, Hardened Memory Allocator". In: *Proc. ACM Program. Lang.* OOPSLA2 (Oct. 2024). DOI: 10.1145/3689773

Developed Software

StarMalloc: a formally verified, modern, hardened memory allocator. URL: https://github.com/Inria-Prosecco/StarMalloc

TranscrIPT: hardware-based memory management operations tracing. URL: https://github.com/Inria-Prosecco/StarMalloc/tree/tracing/data

Contents

Abstract **iii**

Résumé **v**

List of Contributions **vii**

Contents **ix**

1 Introduction **1**
1.1 Memory allocators: role and constraints 4
1.2 Formal verification and machine-checked programs 8
1.3 This thesis 11

2 Background **13**
2.1 Memory allocators: design and implementation 14
2.1.1 Basic memory allocation: `malloc`, `free` and K&R free lists . . 15
2.1.2 C standard APIs and other functions: many required APIs . . 18
2.1.3 Concurrent memory management: thread-safety and scalability 23
2.1.4 Hardening allocators for misuses of APIs: additional complexity 25
2.2 Separation Logic and the Steel framework 31
2.2.1 A primer on Concurrent Separation Logic 32
2.2.2 An introduction to the F* proof-oriented programming language 36
2.2.3 Steel: a Concurrent Separation Logic embedded in F* 47
2.2.4 Extraction of Steel programs to C code using KaRaMeL 55

3 StarMalloc: Verifying a Modern, Hardened Memory Allocator **59**
3.1 Background: StarMalloc's architecture 60
3.1.1 Large allocations: a system call wrapper 60
3.1.2 Regular allocations: a slab allocator 63
3.1.3 Security mechanisms 68

3.2 Hiarchical verification of a bare allocator . . . 70
3.2.1 Verifying the "ground floor": building slabs from slots . . . 70
3.2.2 Linking levels together: reusing generic predicates . . . 76
3.3 Iterative verification: performance and hardening . . . 80
3.3.1 The arraylist data structure: optimizing sizeclass metadata . . . 80
3.3.2 Performance and hardening: arraylist iterations . . . 83
3.4 Genericity and partial evaluation: configurability . . . 85
3.4.1 Reusing the sizeclass allocator inside the large allocator . . . 85
3.4.2 Configurable sizeclasses, partial evaluation and genericity . . . 86
3.4.3 Configurable sizeclass selection function . . . 89
3.4.4 Configurable security mechanisms . . . 90
3.5 Modeling as part of the verification process . . . 91
3.5.1 Functional correctness theorems . . . 91
3.5.2 Specifying hardening features' correctness . . . 93
3.5.3 Syscalls modeling: both for correctness and performance . . . 96
3.5.4 Other low-level axiomatization . . . 99
3.5.5 StarMalloc's Trusted Computing Base (TCB) . . . 102

4 Benchmarking and Deploying Memory Allocators 103
4.1 StarMalloc: evaluation of the implementation . . . 104
4.1.1 Implementation overview . . . 104
4.1.2 Performance Evaluation . . . 106
4.1.3 Integration into Firefox . . . 107
4.1.4 Discussion . . . 111
4.2 Background on profiling and Intel Processor Trace . . . 112
4.2.1 A brief history of hardware profiling, from sampling to tracing 113
4.2.2 Using Intel Processor Trace in practice . . . 114
4.3 TranscrIPT: hardware-based heap tracing . . . 116
4.3.1 Design and implementation . . . 117
4.3.2 Preliminary experimental results and perspectives . . . 119

5 Related work 123
5.1 Systems verification . . . 123
5.2 Benchmarking memory allocators . . . 126

6 Conclusion 131

Bibliography 133

A Linked lists using Steel combinators **157**

A.1 A basic linked lists library using combinators 157

A.2 Equivalence to a textbook linked lists predicate 167

Chapter 1

Introduction

> *By June 1949, people had begun to realize that it was not so easy to get a program right as had at one time appeared. It was on one of my journeys between the EDSAC room and the punching equipment that the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.*
>
> Maurice Wilkes[1], *Memoirs of a computer pioneer*, MIT Press, 1985, p. 145

Software has become pervasive in our lives. Whether it is for instantaneous communication such as email, instant messaging or videoconference; online access to various public services such as housing benefits or more generally interacting with government entities such as when filing taxes; making health appointments, medical imaging and the handling of corresponding health data; e-commerce and online banking: the common backbone of it all is software. This is also the case for various means of transportation: there exist automatically operated metro lines [4]; in aviation, autopilot can assist in guiding the underlying aircraft's flight control system and even perform automatic landing in poor weather conditions (runways Cat. III) [5, 6]; even cars are nowadays standardly equipped with electronic stability control (ESC) [7], improving vehicles' stability.

While we pervasively rely on software, software is not always reliable: accidents partly due to software-related issues have occurred. In their 1993 investigation of Therac-25 software-induced radiotherapy accidents between 1985 and 1987 [8], some of which were lethal, Leveson and Turner's opening statement is the following:

> "Computers are increasingly being introduced into safety-critical systems and, as a consequence, have been involved in accidents."

More recently, the UK Post Office has been involved in a scandal related to an accounting software system developed by Fujitsu called Horizon. With local branches of the Post Office moving from paper-based accounting, various software bugs led to mishandling of transactions [9], resulting in hundreds of sub-postmasters (local post

[1] Creator of the first stored-program computer, the Electronic Delay Storage Automatic Calculator (EDSAC), and of the Modula-3 safe programming language at the end of 1980s.

office managers, liable in case of shortfalls [10]) wrongly convicted of stealing money, to which can be added at least a dozen of suicides [11]... Former UK prime minister Rishi Sunak described it in March 2024 "one of the greatest miscarriages of justice in [UK]'s history"; this scandal is at the time of writing still unfolding. Other noteworthy failures that did not result in casualties include Ariane 5's failed maiden flight, due to erroneous code reuse [12]: this led to the loss of roughly €300 millions [13].

The aforementioned cases form a rather diverse set of examples that may seem catastrophic – they are – but one may wonder: do issues occur with basic blocks that most software is based on? To this end, let us introduce very briefly memory management. Most software, including web browsers, office suites as well as video-conferencing tools routinely handles content whose size cannot precisely be predicted at the time it was developed. Such content include pictures, music, videos, PDF documents. As a consequence, user applications need to be equipped with a dedicated mechanism, so that they can be provided while running with a variable amount of memory to store such data, depending on user and environment inputs. *Memory management* precisely is this much-needed feature, handled by a *memory allocator*.

Unfortunately, commonly-used software is plagued by a class of security issues closely related to memory management: *memory safety issues*, also sometimes called memory corruption issues. As a matter of fact, Microsoft reported in 2019 that 70% of all security updates for Microsoft products addressed memory safety issues [14]. In 2020, Google reported that 70% of "serious security issues" in the Chromium-based Chrome web browser also correspond to such issues [15]. In 2024, they estimated that memory safety issues were used in 75% of zero-day vulnerabilities exploited in the wild, a class of noteworthy vulnerabilities leveraged by attackers during the period of time security practitioners are not aware of them and thus do not provide mitigations against them [16]. To give concrete examples of high-profile memory safety issues, we can name Heartbleed (2014) [17] and BadAlloc (2021) [18]. The former is an issue that was part of OpenSSL's implementation of the TLS protocol, used to secure HTTPS: this bug led to the stealing of health data from US hospitals with millions of patients concerned [19]. The latter corresponds to a serious defect in memory allocation handling found in many different memory allocators implementations [20]. In turn, many embedded devices used in a "wide range of domains, from consumer and medical IoT[2] to Industrial IoT, Operational Technology (OT), and industrial control systems" [21] were affected: exploitation of this vulnerability could have resulted in the loss of lives and damages to critical infrastructure. To put even more emphasis on such issues, one can compare the economic cost of the aforementioned Ariane 5 failure with that of events from the past decade: WannaCry ransomware attacks alone in 2017 are estimated to have cost billions US$ in terms of global damages [22]; a month later, the NotPetya wiper cyberattack, initially targeting Ukraine and then affecting various entities worldwide, is estimated to have caused 10 billions US$ of total damages [23]. These two malwares were using the same exploit: EternalBlue [24], which relies on a memory safety issue affecting various Windows versions [25].

[2]Internet of Things, referring to devices that can be connected to local networks or even the Internet.

Memory safety bugs are nowadays well identified as a pervasive class of vulnerabilities requiring a response. Faced with those bugs affecting a very broad range of entities, there have been numerous calls from governements [26] and national cybersecurity bodies [27, 28, 29, 30], industry [31] as well as academia [32] over the past three years for a systemic shift with respect to memory safety issues. Recognizing the breadth of memory safety issues, and "the untenable position of [...] [r]esponding on a crisis-per-crisis basis" [26], most of these advocate for proactively eliminating entire classes of vulnerabilities thanks to a collaboration between all sectors. In this setting, "proactively" means "at the source" [33]. That is, software must be improved with a "secure-by-design" approach [28]: only secure software should be shipped as a product, thereby reducing the burden on cybersecurity practitioners through preventive rather than reactive action. In turn, the definition of software in the aforementioned wording "secure software" requires specific care: it is actually precisely advocated for a precise definition through the *standardization* of memory safety [32]. There is a consensus that most actionable leverage on this matter stems from using memory-safe languages whenever possible, as these enforce memory safety by default while programming. Among them, one can cite Rust through its ownership model [34]; SPARK through static analysis, formal verification enforcement of contracts and the disabling of dynamic memory management [35]; languages where memory management is automatic through garbage collection, such as Python, Go, C#, Java, OCaml; and through Automatic Reference Counting (ARC) such as Swift [29, 32, 36].

However, describing such best practices or even setting them as a requirement does not improve the huge amount of already existing, vulnerable software. Memory unsafe programming languages such as C and C++ remain among the most commonly used programming languages [29]. As a matter of fact, there exist huge codebases written in C and C++: Google alone recognizes having accumulated hundreds of millions of C++ code "in active use and under active, ongoing development" as of 2024 [31]; these two languages form a "multi-billion line-of-code [...] corpus" [32, 37]. In fact, they even are the languages used to implement the runtimes of many memory-safe languages, raising concerns about the actual freedom of codebases programmed using memory-safe languages from memory safety issues [32]. While there already are many projects to transition such C and C++ codebases to memory-safe languages [38, 39, 33], removing memory safety issues altogether is considered as a long-term end goal, with a long gradual transition and corresponding decades-long strategies [32]. This is why other complementary approaches are deemed necessary: the phasing-out of vulnerable codebases are part of "mitigations to reduce prevalence" of memory issues, while these will not disappear any time soon and thus call for "mitigations to reduce impact" [29].

Back in 2012, Berger already identified that the transition to memory-safe languages would be a long journey and advocated for reducing the impact of bugs deemed inevitable, arguing that "Software Needs Seatbelts and Airbags" [40]: among such safety components, *security-oriented memory allocators*. Security-oriented allocators can be defined as memory allocators developed using additional design constraints and providing specific security features to mitigate common memory safety issues. Berger's metaphor compares software to cars and the at-the-time position of

the software industry to that of the car industry in the 1950s, where "safety was an afterthought at best": software had "lots of horsepower" but no suitable safety measure regarding memory corruption issues. In this setting, security-oriented memory allocators can play the same role as seatbelts and airbags: that of a last line of defense, as accidents, referring to memory corruption issues, are bound to happen. In fact, in 2023, as part of the push for memory safety, US CISA[3] also argued that among impact-reducing mitigations, "hardening memory allocators" can reduce the reliability of exploits [29], leveraging Apple's experience with their own hardened allocator [41]. In addition to that, specific support of memory allocators is essential to enable another set of impact-reducing mitigations, that of hardware memory protections. Such protections encompass memory tagging such as ARM Memory Tagging Extension (MTE) [42] or capability architectures like CHERI [43], that both require specific support from memory allocators, thus further stressing the importance of these software components.

Software can be identified as critical when its reliability is of utmost importance. Increasing the amount of trust one can place into such software can be done by applying specific methods encompassing *formal methods*, that is, "mathematically based languages, techniques, and tools for specifying and verifying such systems" [44]. This field has been successfully applied to a variety of specialized safety-critical software, subject to rigorous standardized evaluation and stringent regulation. One example is avionics software: commercial aircraft in Europe and in North America must respect the DO-178C certification, that is accompanied by the DO-333 Formal Methods supplement, as the benefits of these techniques are recognized by the aviation industry [45, 46].

Hardened memory allocators form one of the key ingredients down the road to memory safety: as such, they can be considered as critical software components, whose reliability must be guaranteed. This calls for the use of formal methods and precise study of these software components that correspond to common building blocks shared by commonly-used software, as we have seen.

1.1 Memory allocators: role and constraints

Memory management, also called *dynamic memory management*, is the essential feature through which programs can *dynamically* request additional space in the form of chunks of memory at execution time. As the size of objects handled by programs often cannot be predicted *statically*, it is an important feature provided by *memory allocators* that interact with such client programs.

As (Unix-like) operating systems already provide userspace programs with mechanisms to get memory, one may wonder: why do memory allocators actually exist in modern settings, distinctly from the operating system? *System calls* form the standard way for such programs to request something from the operating system (OS), such as memory resources. Unfortunately, system calls are expensive in terms of execution time: using a system call each time more memory is necessary would

[3] US Cybersecurity & Infrastructure Security Agency.

be too costly for user applications. This is incompatible with the need for such applications in rather direct interaction with the end user to be fast and reactive. Thus, let us precise a bit the role of a userspace memory allocator: it is to *efficiently* provide user applications with chunks of memory, while using few time resources.

Memory allocators must serve as an efficient intermediate between user applications and the operating system; as part of this role, they must reconcile different abstractions. On one hand, at the end of the day, it is the sole role of a (Unix-like) operating system to manage the hardware memory: as part of this task, the OS provides abstraction over *physical memory* through *virtual memory*. For the sake of brevity, we restrict us to say that the OS is responsible for the translation of virtual memory addresses into physical memory addresses. In addition to that, the OS implements paging: memory is divided in fixed-length blocks called *pages*, whose length typically is 4096 bytes, corresponding to the OS basic unit of memory management. On the other hand, client programs can request chunks of memory of arbitrary size, that is, of any number of bytes. As we will see, this discrepancy has several consequences. In practice, the typical behavior of the memory allocator is to initially request a large amount of memory from the operating system, that will progressively be allocated to user applications. When facing additional or specific requests from client programs, there can be additional requests from the memory allocator to the operating system.

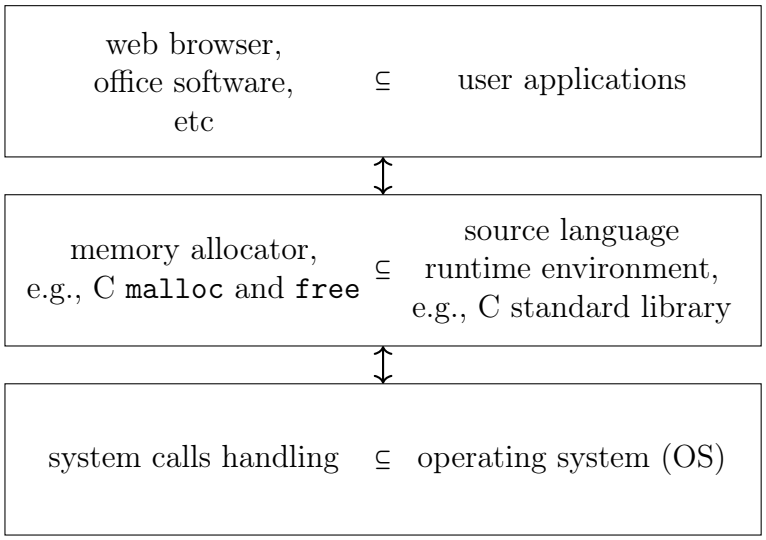
In addition to allocation requests, memory allocators must support deallocation requests. Deallocation requests correspond to the need for client programs to indicate that some previously-allocated memory is no longer needed, so that this memory can possibly be reused. As a consequence of this, the memory allocator must store information during the entire execution of the client program about memory's current state. On one hand, it must keep track of allocated chunks of memory currently in use by the client program, that have not yet been deallocated; on the other hand, it must also keep track of available memory that is not currently in use, but may eventually end up being allocated to the client program. As we will see, while not directly visible from the client program, this fine-grained bookkeeping is at the heart of memory allocators implementations, both as a matter of correctness and of performance. This information about memory's current state is called *metadata*: the allocator must maintain it consistent with the actual memory's state.

As noted by Aho et al. [47, Chapter 7], memory management is part of the runtime environment provided by the source language used to write the source code of the considered program. Memory management may thus vary considerably depending on the source language.

Crucially, the distinction between memory-safe languages and memory-unsafe languages is closely related (albeit not equivalent) to the fact that "allocators come in two basic styles" [48], depending on whether deallocation is manual (that is, explicit for the programmer) or automatic (that is, implicit for the programmer).

On one hand, as allocations always are manual, manual deallocation results in fully manual memory management. The key issue with manual memory management is that it is error prone when not safeguarded by the underlying language. Such languages are considered memory unsafe, as their unrestricted use results in

Figure 1.1: Memory management software stack sketch.



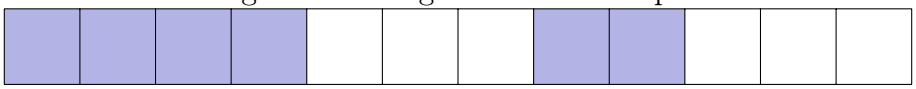
widespread memory safety issues. Examples of safeguards equipping manually managed languages that are considered safe include the use of formal verification tooling around specialized languages, such as Low* for low-level verified programming; and of static analysis such as region analysis to place limitations on deallocations such as the Cyclone dialect of C.

On the other hand, automatic deallocation through *garbage collection* requires the runtime environment to be able to determine whether a block of previously allocated memory is actually still used. There exists a vast research area corresponding to garbage collection, that has led to the development of many widely-used garbage collected programming languages such as Python, Go, Java and OCaml. In some cases where performance is paramount, the incurred additional computing costs and unpredictability are deemed unacceptable, resulting in the use of other approaches: unsafe languages like C and C++, and more recently modern memory-safe languages like Rust and Swift.

In the rest of this document, we will focus on manual memory management, more specifically on the C programming language memory management. C memory management is part of the C standard library, also called libc. Its two main primitives are `malloc` for memory allocation and `free` for memory deallocation. We present a sketch of the C memory management software stack in Figure 1.1.

In this setting, the subset of memory that memory allocators are responsible for is called the *heap*: as part of this role, beyond time performance, memory allocators face numerous constraints [48, 49, 50], such as the required support for arbitrary allocation and deallocation requests sequences. This means that deallocation of previously allocated memory chunks may intervene at any moment, resulting in a large set of possible allocation and deallocation requests sequences. An important consequence of this is that reusing deallocated space for future allocations is not always easy: this can lead to poor organization of the heap, that is, poor usage of memory resources, also called *fragmentation*. We will restrict us to give here one example of fragmentation where, given the current heap organization, currently

Figure 1.2: Fragmentation example.



unused memory cannot be used to satisfy the next request. More precisely, given one allocation request to be fulfilled, the current state of the heap is such that if considering the total amount of unused memory, it should be possible: however, there is not a single suitable contiguous block of unused memory. Figure 1.2 presents such a case, where already allocated blocks are in light blue. There is no single free block large enough to handle a request of 4 bytes in this case.

As noted by Wilson et al. [49] while examining the time and memory performance of various algorithms, there exists a large design space for memory allocators, resulting in a diverse set of implementations. In particular, standardization of memory management fosters diversity, as custom memory allocators can be used as drop-in replacements of the one provided by the system C standard library, of which there also are several implementations. This diversity of implementations is all the more wide-ranging due to the fact that other performance constraints exist. As an example, since client programs can be concurrent ones, memory allocators must support concurrency. Performance constraints corresponding to concurrency handling include: scalability with respect to the number of threads [51], and ensuring that scalability that does incur penalty due to cache-related issues [52].

Outside of these performance considerations, the role of memory allocators in maintaining the heap is crucial as memory safety issues, plaguing software, occur when invariants related to allocation and deallocation are not respected. There are various sorts of memory safety issues: a simple example is when a client program has been allocated a n-bytes memory block and tries to access beyond the allocation end, e.g., the $(n+1)$-th byte; this is called a *buffer overflow*. The issue with this is that access to one allocation can then result to the access of another allocation or even to the allocator metadata, e.g., provided they are adjacent. Let us recall that the operating system memory management unit is that of a page and thus cannot distinguish between correct or incorrect sub-page memory accesses: only the memory allocator has this more fine-grained information about the heap at its disposal. We have seen that fragmentation can happen when small unusable free blocks are intertwined with allocations currently in use and that is not a desirable situation from a performance perspective. However, we also have just seen that, from a security point of view, optimizing the heap layout so that allocations are tightly packed can be undesirable. This results in yet another one possible tradeoff for memory allocators.

Given all of the mentioned tradeoffs, security-oriented allocators, also called hardening or even hardened memory allocators, follow design choices leaning more towards security that pure performance, reducing the impact of memory safety issues. As a matter of fact, security-oriented cannot change whether client programs respect memory safety invariants, that is, they cannot enforce memory safety on their own. Due to their possibly specific memory layout and with the help of additional security mechanisms, they nonetheless can mitigate memory safety issues and

render their exploitation in adversarial contexts more difficult [42, 53]. Depending on the execution environment, it can be more or less useful to favor security: this calls for a diversity of allocators, suitable for each context; memory allocator implementations providing configurable security through configurable security mechanisms is one way to provide this.

A memory allocator is a complex piece of software, with many performance tradeoffs to be made in the form of design choices. On top of this overall delicate system, a security-oriented allocator must be designed and equipped with mechanisms in order to mitigate memory safety issues. This additional requirement comes at the expense of additional complexity, resulting in turn in possible additional implementation issues. As previously mentioned, functional correctness of the resulting allocator is critical to ensure memory safety of the underlying client programs: additional security mechanisms aiming at mitigating memory safety issues should not endanger this fundamental requirement. Finally, this calls for additional efforts be put with the aim of guaranteeing the functional correctness of security-oriented memory allocators implementations.

1.2 Formal verification and machine-checked programs

Studying the correctness of programs and proving them as actually correct corresponds to a field that has been pioneered by Alan Turing as early as in 1949 [54], asking the following question in a 3-pages seminal paper.

> "How can one check a routine[4] in the sense of making sure that it is right?"

In this paper, Turing considers the functional correctness of a function computing factorial ($n \mapsto n!$), that is, whether the output of this function is correct given its specification. In short, does it actually compute factorial? is the question he aims to answers positively. Turing arguably uses invariants to prove this function correct, and a variant to prove its termination. Among other pioneering papers, one could also include works from Goldstein and von Neumann [55] as well as from Curry [56]. This early research and the idea of proving programs remained largely unknown until much later, as Knuth recalled it in 2003 [57]:

> "Bob showed me some work he had been doing about mathematical techniques for verifying that a program is correct – a completely unheard-of idea in those days as far as I knew. The accepted methodology for program construction was quite the opposite: People would write code and make test runs, then find bugs and make patches, [...], and so on until not being able to discover any further errors, yet always living in dread for fear that a new case would turn up on the next day and lead to a new type of failure."

[4] A more modern wording could here be "function".

In this quote, "Bob" actually is Robert Floyd, that gave his name along with Tony Hoare to the Floyd-Hoare logic (or Hoare logic) stemming from two seminal papers in 1967 [58] and in 1969 [59]. Crucially, studying the correctness of programs implies to define precise, unambiguous semantics of the considered programs, or, per Floyd paper's title [58], to manage "Assigning Meanings to Programs"; this amounts to defining formal semantics of the underlying programming languages, such as the ALGOL-60 semantics defined in 1965 by Landin [60]. Indeed, Hoare logic brings Hoare triples, of the following syntax: $\{P\}\, c\, \{Q\}$. This triple specifies, as Hoare puts it, that "[i]f the assertion P is true before initiation of a program c, then the assertion Q will be true on its completion" [59]: in this setting, semantics for the language of c are required. Hoare logic became part of the basis of formal methods and was subsequently largely refined and complemented.

Separation Logic [61] is a refinement extension of Hoare logic that allows reasoning on low-level, imperative programs manipulating pointers. More precisely, its main feature is the reasoning on memory resources with the help of additional, dedicated logical connectives. In the setting of reasoning about memory allocators, another interesting feature is that it can handle ownership transfers of memory [62]. Concurrent Separation Logic [63, 64] is a further refinement of Separation Logic that allows reasoning on concurrent programs.

Formal methods study a diverse set of properties through a variety of techniques, leading to numerous tools that are increasingly used across the industry. Reconsidering Turing's aforementioned research question, we note that in its setting, "right" could have had multiple meanings beyond functional correctness and termination, such as the following ones. 1. When given specific inputs, can considered function crash? 2. Does it have a reasonable resource usage, e.g., execution time? We use these examples to illustrate the diversity of properties that can be considered by formal methods as well as the wide range of techniques developed to study these properties.

Regarding 1., crashes correspond to what is called *run-time errors* (or RTE), that can be caused by programming errors, e.g., accessing an array out of bounds. Determining whether crashes can occur can be tackled in several ways, including through the use of *abstract interpretation* [65], a formal method that static analyzers can be based on. The Astrée static analyzer [66] relies on this technique and has been used to prove the absence of RTEs in Airbus avionics software.

Regarding 2., determining the execution time of a program is impractical in the general setting, as determining whether a program will terminate or not corresponds to *the halting problem*. It thus is undecidable: no algorithm can provide an answer for all possible programs and associated inputs. Worst Case Execution Time (WCET) analysis can however be usefully applied to a subset of all programs through overapproximation of the considered program in order to compute upper bounds of its maximum execution time. WCET analysis can be implemented in various ways and has also been used in industrial settings to provide upper bounds on execution times of programs considering specific underlying hardware [67].

Overall, it is generally considered that more precise properties are more difficult to check and thus less scalable, i.e. less applicable to large programs.

Concerning memory allocators, their specific role calls not only for strong guarantees through the use of formal methods, but also for precise guarantees such as functional correctness that we thus aim for. Indeed, the memory safety of client programs directly depends on the functional correctness of the underlying memory allocators. To tackle this research problem, suitable formal methods require sufficient expressiveness to state functional correctness properties, both for allocation, deallocation as well as security mechanisms in the case of security-oriented allocators. Let us give an example of one of the expected functional correctness properties with respect to allocation. The memory allocator, when faced with an allocation request from a client program of n bytes, must return a memory chunk that should be of length $l \geq n$ if allocation worked as intended. This memory provided to the client is not usable by the allocator until deallocated by the client: this transfer of ownership must be accounted for. To this end, metadata must remain consistent with allocation and deallocation requests.

Deductive verification is the process of using specialized software that enables one to describe objects through specifications and to prove them correct; corresponding proofs are checked as valid by the software.

On one hand, this can be done in an interactive manner by using proof assistants, also called interactive theorem provers. One common class of proof assistants are typed-theory based and rely on the Curry-Howard correspondence to encode logical propositions as types. In this setting, the proof programmer programs in a richly-typed functional programming languages and encode specifications in the form of typed objects. This way, she can provide proofs that are interactively checked by the proof assistant. Examples of type-theory based interactive theorem provers (also called proof assistants) are Rocq (formerly Coq) [68] and Lean [69]. Considered objects can be executable programs or even mathematical theorems. As proving software correct is a very time-consuming task, one of the main challenges is to reduce the burden on the proof programmer so that complex software can be tackled. One way to do so is to rely on *modular abstractions* separating different parts of the proof and code; in type-theory based proof assistants, this can leverage *dependent types* and higher-order predicates. This way, some part of the proof and code can be updated while keeping the rest of it untouched and still valid. Another way to do so is to provide the proof programmer with some sort of automation to manipulate proofs, which brings us to the second sort of deductive verification software.

On the other hand, automated theorem provers form a class of specialized software that can check the satisfiability of logical formulas over various reasoning theories. Among such provers, *SMT solvers* such as the Z3 theorem prover [70] can be used in combination with standard proof assistants to enable automated reasoning about program-related theories (e.g., arithmetic). To this end, the proof assistant must send to the SMT solver a query synthesizing the targeted proof goal so that if the query is satisfiable, targeted property holds: the query is said to be a *verification condition*.

We just mentioned that program formal verification using proof assistants is a complex task faced with significant challenges regarding scalability and iterative development. Proof assistants have nonetheless been successfully used to prove the

functional correctness of complex software such as operating systems (seL4 [71], CertikOS [72]); compilers (CompCert [73], CakeML [74]) or abstract interpretation based static analyzers (Verasco [75]). Thus, deductive verification through the use of a proof assistant seems like a reasonable choice to tackle the formal verification of a memory allocator's functional properties.

In order to verify a security-oriented memory allocator, it seems necessary to overcome two high-level challenges. On one hand, low-level reasoning, e.g., on pointers, can be difficult. On the other hand, updating proofs as part of the development can be time-consuming. As a matter of fact, a userspace memory allocator is a rather low-level software component, that must interact with the operating system and most of all correctly handle raw memory pointers. Furthermore, a security-oriented memory allocator is equipped with security mechanisms: implementing all of it at once may seem like a daunting task. Instead of that, implementing a functional bare albeit already security-oriented allocator to then focus on adding security mechanisms seems more amenable. However, this iterative development, while leaving room for future security extensions, e.g., the support of hardware features, entails that all of the proofs related to functional correctness should support iterative development, that is, be preserved when adding or improving various parts of the allocator.

Targeting the formal verification of a realistic hardened memory allocator with the aim of proving functional correctness properties raises various challenges. While deductive verification through the combined use of a proof assistant and Concurrent Separation Logic seems like a reasonable choice to tackle this task, suitable proof engineering providing iterative development is required.

1.3 This thesis

In this thesis, we argue that **hardened memory allocators can be formally verified and used as drop-in replacements of their unverified counterparts. To tackle the verification of this complex software, we developed a verification methodology extending previous work that relies on the combined use of concurrent separation logic, dependent types, SMT-aided verification and modular abstractions. This methodology provided us with scalability and reduced the proof burden associated with maintaining proofs in an iterative development context. Furthermore, the performance of formally verified hardened memory allocators such as the one we developed can be competitive.**

To validate this thesis, we rely on one case study. We give further background in Chapter 2, first on the design and verification challenges of memory allocators in Section 2.1; as well as a brief overview of Separation Logic and the Steel verification framework that we used as part of our work Section 2.2.

Then, we present our main contribution in Chapter 3: the process of formally verifying StarMalloc, a verified, hardened memory allocator. To this end, we present in this chapter the iterative verification methodology that we extended from previous

work to make iterative development tractable in our setting, e.g., for the verification of data structures that were required as part of StarMalloc's implementation. This verification effort results in a functional correctness theorem about StarMalloc and its configurable security mechanisms, along with the axiomatization of necessary systems calls.

Next, we describe in Chapter 4 the process of benchmarking and deploying memory allocators in real-world setting. As part of it, we demonstrate that StarMalloc exhibits competitive performance with respect to the allocator that it is heavily inspired from on a set of real-world benchmarks aimed at evaluating memory allocators. We also demonstrate that StarMalloc supports the entire set of APIs that can be expected from a realistic memory allocator through the support of Firefox. Finally, we describe a prototype relying on hardware tracing features aimed at gaining understanding related to memory allocators benchmarks and implementations.

Chapter 2

Background

> *Dynamic memory allocation has been a fundamental part of most computer systems since rougly 1960, and memory allocation is widely considered to be either a solved problem or an insoluble one.*
>
> Dynamic Storage Allocation: A Survey and Critical Review, 1995 [49]

In this chapter, we first present challenges related to memory allocators design and implementation in Section 2.1. We then give background about Separation Logic and the Steel verification framework in Section 2.2. As such, this chapter does not contain scientific contributions, presented in following chapters (Chapter 3 and Chapter 4).

2.1 Memory allocators: design and implementation challenges

Introduction

In this section, we will see what are the design and implementation challenges of memory allocators. We have seen in Section 1.1 the role of memory allocators as well as some of their constraints in the setting of C manual memory management, resulting in various possible tradeoffs between time performance, memory performance and security to only name a few. Building upon this, we will present memory allocators designs of increasing complexity and the corresponding implementation challenges. First, in Section 2.1.1, we start from basic memory allocators implementing `malloc` and `free` to show that metadata as part of the allocator bookkeeping is required in simple settings so that memory may be reused. Then, in Section 2.1.2, we present other required APIs as part of the C standard to show that implementing them correctly also requires consistent metadata as well as handling alignment. Next, in Section 2.1.3, we lay the emphasis on the need for modern memory allocators to support concurrent settings: this requires implementations to be both thread-safe and thread-scalable. Finally, in Section 2.1.4, we give examples of practical memory safety issues affecting common memory allocator architectures as well as examples of various security mechanisms to mitigate such issues. Implementing these mechanisms comes at the expense of additional software complexity.

2.1.1 Basic memory allocation: `malloc`, `free` and K&R free lists

Before considering more complex implementations, we believe it is useful first to consider basic memory allocation, only equipped with `malloc` and `free` allocation and deallocation functions: this allows us to highlight minimal implementation requirements in terms of metadata so that previously allocated memory may be reused.

These two functions appeared since the early beginning of the C programming language history, which is closely related to that of the Unix kernel, initially written in assembly by Dennis Ritchie and Ken Thompson. As a matter of fact, similar `malloc` and `mfree` functions were already part of the 4th edition of the Unix kernel (1973) source code [76], the first Unix version to be written in C. They can also be traced back to the `alloc` and `free` functions of the "K&R C" described by Brian Kernighan and Dennis Ritchie in their book *The C Programming Language* [77] first published in 1978, that initially served as an informal specification.

C has since been standardized and subsequently revised, as we will see in the next section. In this section, we will present `malloc` and `free` using the first C standard, C89, completed in 1989 and ratified as an ISO[1]/IEC[2] standard in 1990, also referred to as ANSI[3] C or C90 [78].

2.1.1.1 C89 `malloc` and `free`

First, let us consider the `malloc` prototype, that is, its type signature: its parameters and their types as well as its return type. It has one parameter corresponding to the number of requested bytes to be allocated in the form of a single memory block, that we refer to as `size`; and its return value is a pointer, that is, the address to the allocated memory block if allocation succeeds. In that case, the size (in bytes) of the allocated memory block is greater or equal than `size` and its content is uninitialized. In case of failure, a null pointer is returned: `NULL` is a special pointer value that does not correspond to the address of any valid memory object.

```
1 void* malloc(size_t size);
```

`size_t` is an unsigned integer type that can store the maximum possible size of any object and thus of any memory allocation on the underlying system. `void*` is the type of pointers without associated data type. The returned pointer thus must be (possibly implicitly) cast to the desired type in order to actually be used.

Then, let us have a look at the `free` prototype. It also has exactly one parameter, a `void*` pointer that we refer to as `ptr`: it must correspond to a memory allocation previously allocated by `malloc`. In this case, the space corresponding to `ptr` is deallocated and can be reused for other allocations. Otherwise, this is specified to be an undefined behavior (UB), a case falling outside of specifications, leaving the

[1] International Organisation for Standardization.
[2] International Electrotechnical Commission.
[3] American National Standards Institute.

corresponding result up to the implementation (of the compiler, or in our case of the memory allocator). In turn, this leads to some uncertainty as to the actual runtime behaviour [79].

```
void free(void* ptr);
```

`void` is the return type of functions that do not return any value. A consequence of this is that checking for possible failure is difficult [48, §9.9.1]. Detecting failures is subject to specific tradeoffs, briefly presented in Section 2.1.4.4.

2.1.1.2 An extremely rudimentary allocator

With C89 standard specifications in mind, we aim to lay the emphasis on the need for memory allocators to maintain metadata about previously allocated blocks so that memory may be reused. To this end, we use the example of a very rudimentary memory allocator. Let us now present it: we consider a basic "bump pointer" allocator, only handling the "heap"[4] as one segment $[\texttt{buffer}, \texttt{buffer} + \texttt{HEAP\_SIZE})$ of valid memory, where `buffer` corresponds to the segment's start address and $\texttt{buffer} + \texttt{HEAP\_SIZE}$ to its end. A pointer `next` is maintained such that it partitions this segment in two parts:

- $[\texttt{buffer}, \texttt{next})$ corresponds to the part of the heap that already has been allocated (and that cannot be deallocated, as we will see);
- $[\texttt{next}, \texttt{buffer} + \texttt{HEAP\_SIZE})$ corresponds to the part of the heap that has not yet been allocated and thus is available for further allocations.

Implementing `malloc` can be done the following way. To fulfill an allocation of request of `size` bytes, one can first check whether there is still is enough available space to allocate `size` bytes; if so, store the current value of `next` as `ptr`, increase the value of `next` by `size` (C supports pointer arithmetic) and return `ptr`; otherwise, return `NULL`.

The resulting implementation is very short and simple.

```
#include <stdlib.h>
#include <stdint.h>

#define HEAP_SIZE (1UL << 20) /* 1 MiB */

static uint8_t buffer[HEAP_SIZE];
static void* free_space = buffer;
static size_t total_allocated = 0UL;

void* malloc(size_t size) {
```

[4] In the code snippet below, the region of allocations is not strictly speaking part of the heap: static uninitialized data (and thus implicitly zero-initialized data) is part of the uninitialized data segment (.bss). One may use the `sbrk` system call to use the heap as a resizable memory region.

```
11   // bump allocator
12   if ((total_allocated + size <= HEAP_SIZE) && (size > 0)) {
13     void* ptr = free_space;
14     free_space += size;
15     total_allocated += size;
16     return ptr;
17   } else {
18     return NULL;
19   }
20 }
21
22 void free(void* ptr) { return; }
```

However, simplicity and time performance of this approach comes at a large cost: `free` does not actually perform deallocation, thus resulting in poor memory performance. In fact, deallocation amounts to "marking" blocks of memory as reusable: in this setting, as no metadata is kept about allocated blocks, such as their size, no information is available either once they have been deallocated. In this context, without information about the heap layout of free blocks (neither pointers nor associated memory blocks sizes), it is not possible to reuse them. Overall, this example underlines that in practice, an allocator must keep track in a fine-grained way of free memory, so that it may be reused, as also noted by Bryant and O'Hallaron [48].

2.1.1.3 K&R allocator: freelists

The requirement to keep track of free memory so that it may be reused was already noted by Kernighan and Ritchie [77], that proposed as a solution an allocator using a *free list*, referred to as the *K&R allocator*. The main idea is that due to the almost arbitrary sequences of `malloc` and `free` calls, free memory is unlikely to form a contiguous part of memory: one way to keep track of it is by maintaining a linked list of all free blocks. To implement it, each memory block contains a header at its beginning, storing its size in all cases, and the pointer to the next block in the freelist for free blocks. To allocate, one can either attempt finding a large enough block in the freelist or request more memory from the operating system (e.g., through `sbrk` or the `mmap` system call).

This design is not immune to fragmentation. Let us briefly present the two sorts of fragmentation. On one hand, external fragmentation, as briefly covered in the introduction, occurs when the total amount of free memory is enough to satisfy an allocation request while there is no single large enough memory block to actually fulfill it. On the other hand, internal fragmentation occurs when an allocated block used to satisfy an allocation request is actually larger than the requested amount of memory.

There are many different ways to search for suitable block in the freelist: one can return the first block of suitable size from the beginning of the free list (*first fit*); the best block regarding the allocation request among all blocks of the free list to optimize for internal fragmentation, thus of minimal size (*best fit*); or the first

block of suitable size from a position saved between allocations when the linked list actually forms a ring (*next fit*, due to Knuth). Once a suitable block is found, it is removed from the list (retrieving its predecessor can be done efficiently by using a doubly-linked list), and a pointer to the usable part of the block (beyond the header) is returned.

Deallocation can be done by inserting the block in the freelist following various possible criteria (such as keeping it sorted by block addresses) [48]. Various approaches have been developed to minimize fragmentation: large blocks may be split at allocation to obtain smaller blocks and adjacent free blocks may be merged to obtain larger blocks. In this last case, this is also called *coalescing* and may be done at deallocation in constant-time due to the use of Knuth's boundary tags [80, Chapter 2.5].

To improve performance by reducing the search space for a suitable block when allocating, it is also possible to maintain a set of linked lists, forming a partition by size of all free blocks: the equivalence classes are called size classes. This approach is called *segregated free lists* and is useful for various means, as we will see. Size classes may either be interval of integers, thus providing variable allocation sizes; or singletons, providing fixed allocation sizes. In the rest of this document, we only consider the latter case. *Buddy allocators* form a particular case of this setting, where all size classes are powers of two. The use of size classes can help reducing external fragmentation at the possible cost of increased internal fragmentation, depending on requested allocation sizes. Last but not least, if each sizeclass is given one segment of the heap whose address and size is known, determining the size of a block may be done by only using its address. In this setting, there is no need to store size as part of the metadata for each block, as it is encoded as one underlying memory layout invariant. This can improve performance through less memory accesses and help with code complexity.

2.1.2 C standard APIs and other functions: many required APIs

Beyond `malloc` and `free`, the C89 standard already included other functions: `calloc` and `realloc`. Additionally, as we will see, alignment handling is also required as part of more recent C standard APIs, in addition to other expected memory management functions outside of the C standard. In this section, we present corresponding challenges.

As previously mentioned, C89 was the first international C standard (ISO/IEC). More recent revisions, also all ratified as international ISO/IEC standards, include: C99, completed in 1999 and ratified in 2000 [81]; C11, completed and ratified in 2011 [82]; C17, completed in 2017 and ratified in 2018 [83]; and finally C23, the current C standard, completed in 2023 and ratified in 2024 [84].

2.1.2.1 C89 `calloc`: zeroing and an infamous multiplication on top of `malloc`

Let us consider the prototype of `calloc`. This function is rather similar to `malloc` and as we will see, can use `malloc` as a basis. It has two parameters of type `size_t` `nb_elem` and `size_elem`; its return value is a `void*` pointer. If allocation succeeds, the returned pointer points to an allocation suitable for an array of `nb_elem` elements, each of these being of size `size_elem`. Furthermore, in this case, the returned allocation, that necessarily is of size greater or equal than `nb_elem` × `size_elem`, must be initialized with zeroes. Otherwise, in case of failure, a null pointer is returned.

```
1 void* calloc(size_t nb_elem, size_t size_elem);
```

This function can be implemented as a call to `malloc` requesting `nb_elem`×`size_elem` bytes followed by zeroing. This however requires extra care as the incurred multiplication can lead to integer overflow. A `calloc` example without such a check is part of the BadAlloc memory allocation vulnerabilities [21].

2.1.2.2 C89 `realloc`: requires metadata about allocated blocks

As we will see, `realloc` is a more complex function. This function can be used to resize a memory allocation, either to shrink it or to expand it, possibly at the cost of using a new memory block at a different address, which requires in that case to copy its content. `realloc` has two parameters: `ptr` of type `void*` and `new_size` of type `size_t`; its return value is a `void*` pointer.

```
1 void* realloc(void* ptr, size_t new_size);
```

In order to be precise, let us cite the C89 standard. "The realloc function changes the size of the object pointed to by `ptr` to the size specified by [`new_size`]. The contents of the object shall be unchanged up to the lesser of the new and old sizes." Importantly, as implied by the standard, the memory allocator is required to know the size of all currently in-use memory allocations.

There are four different cases that `realloc` must handle.

- If `ptr` is a null pointer, then the behaviour should be the same as `malloc(new_size)`.
- If `ptr` is a valid pointer but that does not correspond to a memory block allocated by `malloc`, `calloc` or `realloc`, this corresponds to a UB.
- If `ptr` is a valid pointer corresponding to a memory block allocated by `malloc`, `calloc` or `realloc` and `new_size` > 0, it can be assumed that the allocator has access to the size of `ptr` as part of its metadata: we note it `old_size`. The allocator can thus determine whether the existing allocation should be shrinked or expanded. Depending on the memory layout, this can be done in place or through a new memory allocation noted `new_ptr` (e.g., using `malloc`) of size `new_size`, with the min(`old_size`, `new_size`) first bytes of `ptr` being copied

to `new_ptr`: any possible rest of the allocated memory block is uninitialized; `ptr` is then deallocated (e.g., using `free`). A null pointer is returned and the content of the memory block corresponding to `ptr` is left unchanged should any failure occur.

- If `new_size` = 0, assuming `ptr` is pointing to a previously allocated memory block (by `malloc`, `calloc` or `realloc`), C89 specifies that the behaviour should be the same as `free(ptr)`. C99 and C11 do not specify it: according to these standards behaviour is implementation-defined. As foreshadowed by its deprecation in C17 [85], this case corresponds since C23 to a UB.

As is the case for `calloc`, it should be noted that this "swiss-army knife" function can be implemented using `malloc` and `free`. It should however be noted that this is not always the most efficient solution. One example is that the reallocation of large allocations may be done more efficiently in Linux environments by using the specialized `mremap` system call [86, Chapter 49.8].

2.1.2.3 C11 `aligned_alloc`: requires alignment handling

While C99 did not introduce any memory management-related new function, C11 introduces a new function: `aligned_alloc`. Its key new feature is that it allows one to specify the required alignment for requested memory allocations: let us first define alignment.

Memory[5] may be viewed by one process as a single very large array of bytes: in this large array, each byte can be identified by its offset, also called its address. However, in general, computer systems restrict the set of addresses that can be used to store data, "requiring that the address for some objects must be a multiple of some value K (typically 2, 4 or 8)" [48]. C89 already specified that a pointer returned by memory management allocation functions is "suitably aligned so that it may be assigned to a pointer to any type of object", further refined by C11 "... object with a fundamental alignment requirement". In practice, this means that such an allocation must have as minimal alignment the largest fundamental alignment: C11 provides a suitable upper bound in the form of the `size_t` value `_Alignof(max_align_t)`, equal to 16 on a x86_64 machine. As a consequence of that, since C11, all pointers returned by memory management functions on x86_64 machines must be at least 16-bytes aligned (i.e., their address is a multiple of 16).

Coming back to `aligned_alloc`, this allocation function allows client programs to specify stricter alignment as required. Beyond alignment, it is very similar to `malloc`. It has two parameters of type `size_t`: `alignment` and `size` and its return value is a `void*` pointer. If allocation succeeds, the returned pointer points to an allocation whose alignment is specified by (and thus is greater or equal than) `alignment`, whose size in bytes is greater or equal than `size` and whose content is uninitialized. In case of failure, a null pointer is returned.

```
1 void* aligned_alloc(size_t alignment, size_t size);
```

[5] Virtual memory in this setting.

If `size` is not a multiple of `alignment` or if `alignment` does not correspond to an alignment supported by the considered implementation, a null pointer must be returned. While the C11 standard specifies that an alignment is a "nonnegative integral power of two" of type `size_t`, the subset of alignments to be supported by a memory allocator is interestingly implementation-defined.

Implementing `aligned_alloc` can be done through the use of a buddy allocator. Indeed, all objects part of the 2^k size class are naturally aligned, assuming that the start pointer of the considered size class is suitably aligned.

2.1.2.4 C23 `free_sized` and `free_aligned_sized`: additional metadata to detect UBs?

C17 did not introduce any new feature, focusing instead on clarifications and minor fixes; memory management functions were mostly left unaffected. C23 includes two new memory management functions: `free_sized` and `free_aligned_sized`.

```
1 void free_sized(void* ptr, size_t size);
2 void free_aligned_sized(void* ptr, size_t alignment, size_t size);
```

These two deallocation functions are quite similar to the `free` function, with two main differences.

On one hand, using these functions require client programs to keep track of the previously requested allocations: of their size for both, and of their alignment when using `free_aligned_sized`.

On the other hand, they also require zealous memory allocators that aim at detecting UBs to keep track of whether allocations were provided by `aligned_alloc` or other functions. Indeed, deallocating any other allocation than those provided by `aligned_alloc` using `free_aligned_sized` is specified as UB; conversely, deallocating any `aligned_alloc`-provided memory block using `free_sized` is also specified as UB.

The stated interest at the time such functions were proposed depends on the considered setting [87]. On one hand, when focusing on performance, retrieving the size class associated with one object to be deallocated can be costly: avoiding incurred operations improves performance. On the other hand, in security-oriented settings, this enforces heap consistency onto client programs and can be used for hardening.

2.1.2.5 *De facto* required functions outside of the C standard

The bestiary of functions required to implement a memory allocator usable as a drop-in replacement for real-world programs does not stop here. Indeed, there exists many functions that are *de facto* required, as they fall outside of the C standard.

As previously mentioned, the history of the Unix operating system and that of C are entangled. Notably due to the availability of its source code, Unix has

inspired many operating systems that have been since then characterized as "Unix-like" [48, Chapter 1]. The POSIX[6] standards emerged in the 1990s as a response to the need of portability for software across this family of OSes. POSIX has been successful in providing uniform abstraction for diverse operating systems systems, including Linux distributions, Apple's macOS and Android [88]. Specifically, POSIX standards include a standard C library, itself a superset of the ISO C standard: the POSIX 2024 issue defines extensions over the ISO C standard memory management, using C17 as a basis [89, p.612]. In particular, POSIX includes since 1999 the following `posix_memalign` allocation function.

```
int posix_memalign(void** memptr, size_t alignment, size_t size);
```

This function provides aligned allocations and is very similar in that sense to `aligned_alloc`[7]. The main difference is that the allocation pointer is not returned but written at the address pointed to by `memptr`, hence the extra level of indirection in its type.[8] Additionally, the returned integer is 0 on success, and non-null otherwise, mimicking `errno` [89, p.1576].

So far, we only did not name any concrete, actually used C standard library implementations. The most commonly used libc in (non-embedded) Linux environments is the glibc[9] [86, Chapter 3.3]. As a consequence of this, a number of non-POSIX features that are part of the glibc are commonly relied upon by client programs. One of these is the memory management `malloc_usable_size` function.

```
size_t malloc_usable_size(void* ptr);
```

It is neither an allocation nor a deallocation function: instead, it provides a user with coarse access to the internal memory allocator bookkeeping. This function can be used to retrieve the size of a previously allocated memory block. Provided `ptr` is a previously allocated memory block, the returned value is only specified to be at least that of the size of this memory block, thus possibly larger than the originally requested size. If `ptr` is `NULL`, the returned value is 0 [92].

Other obsolete GNU extensions are still expected by a number of programs. As a matter of fact, the glibc documentation requires a replacement allocator to also provide the following functions [93].

```
void* memalign(size_t alignment, size_t size);
void* valloc(size_t size);
void* pvalloc(size_t size);
void* cfree(size_t size);
```

[6] Portable Operating System Interface

[7] `posix_memalign` actually appeared before `aligned_alloc`.

[8] `void**` corresponds to a pointer to a `void*` pointer.

[9] Also called the GNU C Library; the Linux kernel and the GNU Project have long been closely related; as an example the GNU toolchain always has been the default to build Linux [90]. There also exists other Linux general-purpose libc implementations such as musl [91].

`memalign` is similar to `aligned_alloc`, though less restrictive, as the `size` is not required to be a multiple of `alignment`; `valloc` returns a memory allocation of at least `size` bytes that is page-aligned; `pvalloc` is `valloc`-alike but rounds the requested allocation size up to the next multiple of the system page size; `cfree` is a very obsolete `free`-like deallocation function.

2.1.3 Concurrent memory management: thread-safety and scalability

In this section, we present challenges associated with the implementation of a thread-safe, scalable memory allocator, suitable for highly-concurrent settings. Back in the early 1990s, shared-memory processors (SMP) (or multi-core processors) were already in use. In 1998, Larson and Krishnan [94] already identified a set of design constraints for memory allocators used in concurrent settings by long-running client programs on SMP systems. Among them, the need for scalability: allocators "must support highly concurrent operations and execution in SMP systems, ideally scaling linearly with the number of processors" [94]. In 2000, it was noted that "Parallel, multithreaded C and C++ programs [...] are becoming increasingly prevalent" with the then emergence of SMP systems, prompting Berger et al. [52] to develop the Hoard memory allocator aimed at scalability.

2.1.3.1 Thread safety

Outside of performance considerations, a memory allocator supporting concurrency is required to be thread-safe. Thread safety can be defined the following way: "a function is said to be thread-safe if and only if it will always produce correct results when called repeatedly from multiple concurrent threads" [48, §12.7.1]. In order for a memory allocator to be thread-safe, all of the memory management functions it provides must be thread-safe.

There are two main approaches to thread safety: either by not sharing variables between threads; or, when unavoidable, by controlling this sharing between threads through synchronization, e.g., through *mutual exclusion*, that is, by ensuring that each thread has mutually exclusive access to resources [48, §12.5.1]. In the first case, one could imagine that it could be specified that a thread cannot interact with memory blocks allocated by another thread (spawned by the same process). This would be utter nonsense, as the use of threads for concurrent programming would lose any interest in this setting; furthermore, the heap is part of virtual memory, which is shared between threads. As outlined by Larson and Krishnan, "a [memory] block must not be tied to the thread that allocated it [...] it should be possible for one thread to allocate a block, a second thread to use it, and a third thread to free it (even if the original thread has already died)" [94]. Thus, thread safety is necessary.

There exists various approaches for synchronization between threads: mutual exclusion through the use of locks, lock-free data structures as part of non-blocking algorithms, and message passing. All of these have been used in practice [95, 96];

for the sake of simplicity, we will only consider locks.

Using locks, one way to provide thread safety is to have one lock used by all memory management functions so that only one thread can be using one such function at a time. However, using a single lock (more precisely, a mutex) would be very costly, as this would result in a lot of contention: many threads would be waiting to lock the mutex, resulting in "compute congestion" [97].

Another way to proceed is to use more fine-grained locks. Reusing the previously presented basic memory allocator design relying on segregated free lists, partitioning free space into size classes, using one lock per size class reduces contention.

2.1.3.2 Scalability and false sharing

As noted by Larson and Krishnan, while using one lock per freelist indeed reduces contention, this approach does not provide the expected scalability improvements [94]. They attribute this to what they call "cache sloshing", later called "false sharing" [52], an issue related to poor cache utilization by multithreaded programs on multicore systems. Cache is a complex topic on its own: let us present briefly what is at stake here.

In the setting of computer architecture, efficient access to data that need to be processed is of utmost importance. To avoid a related possible bottleneck, a variety of mechanisms exist, including caching. The key idea of caching is that providing faster access to only a subset of all accessible memory can lead to performance improvements. SMP systems can be coarsely described as a set of cores all having access to one shared memory, in addition to each core being equipped with one cache.[10] These caches all store segments of memory of the same fixed size into *cache lines*. In practice, when a core must fetch data from the shared memory, it first checks whether this data is present in its cache: if so, it retrieves it directly from the cache (*cache hit*); otherwise, it retrieves it from the shared memory (*cache miss*) and replaces existing data in its cache by the just-retrieved data.[11] Data fetched from caches must be consistent with the shared memory: when a memory location corresponding to data present in the cache of one core c_1 is updated by another core c_2, this requires synchronization between c_1 and c_2. *False sharing* occurs when a same memory segment fitting into one single cache line is continuously updated by two threads executed on two distinct cores [52], leading to performance penalty associated to the corresponding synchronization cost.

Back to the setting of segregated freelists, when several threads handle allocations belonging to the same sizeclass, this results in updates to the same freelist, albeit it is protected by a mutex. This can result in some segments of memory (each fitting into one single cache line) being continuously updated by two distinct threads on distinct cores, resulting in false sharing and thus limiting scalability.

[10] In practice, modern CPUs are diversely equipped with many different caches forming a cache hierarchy as well as additional specialized caches, e.g., to improve the performance of virtual memory.

[11] In practice, there exists a wide range of cache policies in the literature, further details are not needed here.

One way to mitigate this is to improve per-thread locality (and thus per-core locality at runtime) of data structures handled by the memory allocator: to this end "subheaps", also referred to as "arenas", can be used [94, 51]. Concretely, we can think of it as a number n of copies of one memory allocator: each client program thread t is assigned one such copy $f(t)$ identified as an integer, such that for any thread t, the following holds: $0 \leq f(t) < n$. The number of arenas n may be statically fixed [94] or not; in most cases, arenas are not uniquely assigned to threads and rather shared between some of the threads, but not all of them.

As previously mentioned, any thread may be faced with a deallocation request of a memory block allocated by another thread: it is thus also required that all threads, given a pointer to a memory block originally allocated by a thread t that must be deallocated, can identify the corresponding arena $f(t)$. In practice, this technique works well due to the fact that, while such a case must be supported, this does not correspond to the most common case in most settings [94].

All in all, this results in additional challenges and further fine-grained bookkeeping required as part of the memory allocator implementation. Many modern memory allocators rely on a variation over this idea and try to improve per-thread locality: this is the case for jemalloc [51]; for the glibc malloc, also called ptmalloc2 [98]; for mimalloc [99]; as well as for hardened_malloc [100], among many others.

2.1.4 Hardening allocators for misuses of APIs: additional complexity

In this section, we present challenges associated with the implementation of security-oriented memory allocators aimed at mitigating memory safety issues. Memory safety issues have a long history, and have been exploited since early in the computing history [101]. Furthermore, we already stated that their pervasive impact has many measurable consequences and that hardening memory allocators can be practically used to mitigate them.

It may thus be surprising that on most Linux versions the default C memory allocator, that is named ptmalloc2 and is part of the GNU C standard library (also named glibc), is vulnerable to various exploitation techniques [102]. As we will see, this is partly due to the fact that it relies on the freelists techniques that we have presented in previous sections.

On the opposite, there already exists many other memory management techniques used as part of hardened memory allocators implementations, each aimed at mitigating some classes of memory safety issues. The resulting set of diverse approaches reflects various security tradeoffs, related to just as many various execution settings. We will here present some security mechanisms to show that implementing hardened memory allocators require specific, additional implementation efforts due to incurred complexity.

2.1.4.1 Memory safety issues typology and C lack of safeguards

As part of this technical background, we presented the collection of simple APIs that memory allocators must support in practice, briefly describing that they require client programs to respect several invariants. Memory safety issues precisely corresponding to the violation of these invariants and can be classified into two sorts: spatial safety issues and temporal safety issues [37, 103]. In this setting, we only consider heap-related memory safety issues.

On one hand, spatial safety issues correspond to memory accesses to locations that are outside of the bounds of currently valid objects. They encompass the following heap-related memory safety issues subclasses.

- (Heap-based) Buffer Overflows [104]: a subcase of out-of bounds writes. Such an issue is used by the EternalBlue exploit [25].
- (Heap-based) Buffer Over-reads [105]: these are out-of bounds reads. This is the root cause of the Heartbleed vulnerability [17].

Such issues are due to the fact that C supports the use of raw pointers and associated pointer arithmetic, and lacks bound checking, statically as well as at runtime.

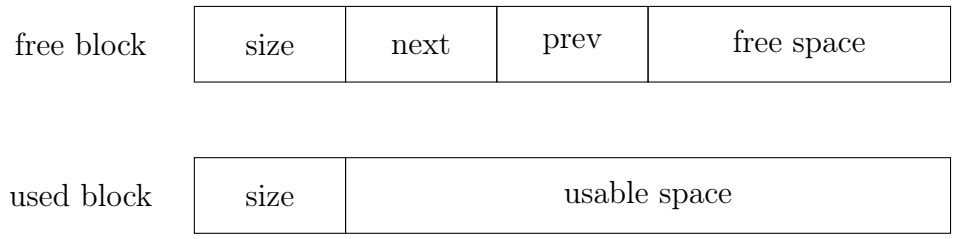
On the other hand, temporal safety issues correspond to invalid handling of objects that are not currently valid.

- Use-After-Frees (UAFs) [106]: when `free` has been called to deallocate a pointer, this pointer does not point to a valid block and is called a *dangling pointer*: any subsequent use of it is invalid. The issue with such bugs is that corresponding memory may be later reused, possibly corrupting data and breaking invariants of the client program.
- Double-Frees [107]: when `free` is called twice on the same pointer, such that the corresponding memory location has not been reallocated in between. Double-Free can lead to the corruption of the allocator's metadata when it relies on freelists [37], as we will see in Section 2.1.4.2.
- Invalid Frees [108] (included as temporal safety issues by [109]): when `free` is called on a pointer corresponding to a memory block that has not been allocated by a heap management function.
- (Heap-based) Uninitialized reads [110, 111] (included as temporal safety issues by [112]): when reading memory from an allocated block that has not yet been initialized. The C standard does not require memory allocators to initialize memory, as we have seen (e.g., using `malloc`); due to memory reuse, uninitialized reads can lead to information leakage.

Most of these issues can be prevented by automatic memory management, considering cases where destroying objects cannot be done manually [37].

Double frees and invalid frees highlight that input validation is required when implementing `free`. Lack of input validation can result in other issues slightly harder to classify, such as the BadAlloc integer overflow vulnerabilities due to the lack of input validation for allocation functions [21, 20].

Figure 2.1: Memory layout of free and used blocks with freelist headers.



2.1.4.2 A case in point for protecting metadata: corrupting freelists metadata

Let use reuse a simple segregated free lists allocator with no cache, assuming all elements inside a given free list have exactly the same size. We further assume that when allocating, the first pointer in the freelist is used; and that when deallocating, the pointer is put back at the head of the freelist. Finally, we assume that the layout of free blocks and used blocks is the one presented in Figure 2.1. Free blocks and used blocks only share in common that their header contains the underlying block size: other metadata are not useful for allocated blocks as these are not part of the free list. As we will demonstrate, using a single double-free allows one to force the allocator to return an arbitrarily-chosen pointer as one of the next allocations.

The main idea is to break metadata invariants, which in this setting correspond to the wellformedness of the linked list, so that one memory block ends up in an ambiguous status. To this end, let us have a look at the following `malloc`/`free` sequence. At some point, there will be one memory block at the same time corresponding to a valid allocation and present in the freelist.

- `a` = `malloc`(16)
- `b` = `malloc`(16)

There may be an arbitrary number of allocations and deallocations in the sizeclass between the previous step and the next step. In the following, let us describe for each step the resulting freelist state in the form of a list traversal from its head, only using the `next` field of each header. We denote by `x` → `y` a subset of the freelist containing memory blocks `x` and `y` such that `x.next` points to `y`.

- `free`(`a`); the memory block corresponding to `a` is added to the freelist, such that the `next` field of its header points to the previous head of the freelist `freelist`.
  We get the following: `a` → `freelist`.
- `free`(`b`); `b` is added to the freelist too.
  Similarly, we obtain: `b` → `a` → `freelist`.
- `free`(`a`); this is the only invalid step from a memory safety perspective, corresponding to the second `free` of the considered double-free. `a` is added to the freelist *again*, thus creating a cycle in the freelist (noted ⋯) and thereby breaking its wellformedness.
  This results in the following freelist: `a` → `b` → `a` → ⋯.

Now, the exploit requires the second occurrence of `a` to be at the head of the freelist. As long as `a` is not deallocated, there may be an arbitrary number of allocations and deallocations between the previous step and the next step. In any such case, the resulting memory layout will be suitable for exploitation.

- `c` = `malloc`(16); `c` is actually equal to `a`, as the first memory block in the free list is used to provide memory.
  At this point, all we know is that the two first blocks of the freelist are the following: `b` → `a`. Indeed, the deallocation process may be such that the `prev` and `next` header fields are reset when the underlying memory block is allocated.
- `d` = `malloc`(16); `d` is actually equal to `b`.
  Similarly, at this point, all we know is that the head of the freelist is `a`.

Let us recall that the memory layout is such that the address of the `next` field of the header of an unused block is the same as that of the start of the allocation when the very same block is allocated. A same memory location can thus correspond to metadata or actually usable space depending on the status of the considered block.

- At this point, one can *legitimately* write within the `c` allocation to corrupt the freelist metadata: `c`[0] = `arbitrary_ptr`.
  The beginning of the freelist then is: `a` → `arbitrary_ptr`.
- `e` = `malloc`(16); `e` is actually equal to `a`.
  This results in `arbitrary_ptr` being the head of the freelist.
- `f` = `malloc`(16); we finally get a pointer `f` returned by `malloc` that is equal to `arbitrary_ptr`.

In fact, this is the exact same sequence that an exploitation technique of the freelist-based glibc allocator relies on, in addition to bypassing some basic pointer protection in place [102]. Remarkably, the same repository [113] proposes a wide range of exploitation techniques usable with the latest version of the glibc allocator, in addition to the one we just sketched. Overall, this argues for a dedicated memory layout strictly separating memory allocations and corresponding metadata in disjoint regions, so that metadata may be protected.

2.1.4.3 Mitigating memory safety issues subclasses through dedicated security mechanisms

There exists a wide-ranging literature proposing various mitigations against C memory corruption issues. We present a brief overview of security mechanisms that can be implemented as part of a security-oriented memory allocator, noting that the description of a rigorous threat model for each of these would be an endeavor of its own. Acknowledging the limits of these short descriptions, we lay the emphasis on the fact that such security mechanisms form a diverse set, sometimes mitigating several memory safety issues subclasses at once.

Out-of-band metadata. Assuming that metadata placed in a region that is disjoint from that of allocations and placed at an address that cannot be guessed by an attacker (e.g., due to the probabilistic ASLR protection), this can be used to

deterministically detect double frees or invalid frees. This is useful since these can be leveraged to corrupt metadata, as presented in previous section.

Zeroing. On one hand, zeroing allocations when they are deallocated can help with leakage of data, e.g., due to a use-after-free. On the other hand, zeroing allocations at allocation time before they are provided to client programs can mitigate uninitialized memory use. If zeroing-on-free is performed, then only checking that the considered block is still zero-initialized is enough.

Heap canaries. Analogous to stack canaries [114], these are values placed at the boundaries of memory allocations at allocation and checked at deallocation, thus possibly detecting buffer overflows.[12]

Guard pages. The OS enforces permissions regarding memory at the pages granularity. To detect large buffer overflows as soon as they occur and not when closest allocations are deallocated, one can intersperse with pages used for allocation other pages that do not have read nor write permissions, also called *guard pages*. Any access to such pages leads to a memory fault, which generally results in the termination of the program.

Quarantine. Quarantine aims at limiting the reuse of a memory allocation in different contexts (possibly for values of different types), thus mitigating use-after-frees [115]. This is done by delaying the reuse of freed allocations, through further partitioning of the heap: quarantined allocations cannot be reused until they are unquarantined.

2.1.4.4 Various adversarial settings: memory allocators as soft safety nets or strict security-enforcing monitors

As we have seen, there exists a rich set of memory safety issues. Given one memory allocator that is able to detect some of these issues, there are two possible choices of implementation when a memory safety issue is detected: tolerating it so that the execution of a faulty program may continue sparing users from impromptu failures [116], or aborting the execution of the program as soon as such an error is detected, as"[m]emory errors in general allow the attacker to read and modify the program's internal state in unintended way" [37].

This corresponds to a trade-off between reliability and security. Indeed, considering an active attacker, in a setting where attacks do not hinder further attacks, e.g., due to the fact that they do not result in crashes, repeated attacks will end up succeeding [117].

It is thus extremely valuable for a security-oriented memory allocator to detect any unexpected behavior. Arguably, the set of specified UBs that we detailed when

[12] In doing so, they can reveal an issue early, hence their name, by analogy with the canaries used in coal mines.

presenting memory management APIs required as part of the C standard are such unexpected behaviors. However, as we have seen, the standard has been revised. Given a valid pointer `ptr` and a bleeding-edge security-oriented memory allocator, should `realloc(ptr, 0)` lead to an application crash, considering this corresponds to a UB since C23, published less than a year ago at the time of writing? Indeed, one could play the devil's advocate and argue that this could hypothetically be part of a double-free memory issue, its detection may thus be hindered by tolerating such deprecated use of `realloc`. This, in turn, argues for being unforgiving of such backwards compatibility issues.

All in all, detecting memory errors is desirable from a security perspective: as we have seen, this can be done through a dedicated memory layout putting allocations and metadata in disjoint memory regions and additional security mechanisms. Aborting the execution of the entire process corresponding to the client program when detecting some API misuse can be reasonable in many cases. Configurable security mechanisms and response to such issues however seems like worth supporting for a real-world hardening memory allocator, as it is likely to be used in heteregeneous settings.

2.2 Separation Logic and the Steel verification framework

Introduction

In this section, we present the Steel verification toolchain, that provides users with a Concurrent Separation Logic embedded within the F* programming language, targeting low-level programs. Such programs, once verified using Steel, can be extracted to C code using the KaRaMeL tool that provides extraction of low-level F* programs to C code. We rely on this entire toolchain as part of our contributions, see Chapter 3.

2.2.1 A primer on Concurrent Separation Logic

In this section, we successively present Hoare Logic (for sequential programs) in Section 2.2.1.1; Separation Logic (for sequential programs), an extension of Hoare Logic, in Section 2.2.1.2; and finally, Concurrent Separation Logic in Section 2.2.1.5.

2.2.1.1 Hoare Logic

Hoare Logic [59] is a formal system to rigorously reason about the correctness of computer programs. Core to Hoare Logic is the notion of a Hoare triple, written:

$$\{P\}\, c\, \{Q\}.$$

This logical statement asserts that in any state satisfying the precondition P, the execution of the program c, it if terminates, produces a state satisfying the post-condition Q. As this statement does not specify all possible behaviours of c, it is called a *partial correctness* triple. Similar statements capturing that t necessarily terminates are called *total correctness* triples but are not used in the remainder of this document.

Let us give an example of a simple partial correctness triple, assuming a simple language:

$$\{x > 0\}\, y := x\, \{x > 0 \wedge y = x\}.$$

In this setting, the precondition is that the x variable is a positive integer, the y variable is assigned with the value of x, resulting in a postcondition reflecting that. A less precise postcondition, such as $Q = x > 0 \wedge y > 0$, would also be valid.

Considering Hoare triples, we implicitly considered a programming language and its semantics used to describe c. In addition to this, P and Q are assertions also called state predicates, as they describe underlying states. This requires a corresponding assertion language: first-order logic can be used as such. The considered programming language's basic features can be precisely specified using triples, corresponding to axioms.[13] Using all of this, Hoare triples about programs can be derived using sound reasoning rules.

Their author envisioned their use to prove programs through deductive reasoning, with the aim to achieve the production of programs that would be at the same time: 1. reliable, thus avoiding consequences of programming errors; 2. portable across machines; 3. equipped with correct documentation, thus making easier iterative development [59]. In particular, regarding 3., Hoare considered using specification as documentation to achieve compositional development.

> "[...] when it becomes necessary to modify a program, it will always be valid to replace any subroutine by another which satisfies the same criterion of correctness."

While it proved very useful, modular reasoning using Hoare Logic turned out to be difficult when handling low-level programs using pointers, especially in presence

[13] Hence the term *axiomatic semantics*.

of aliasing [118]. These limits are all the more important since, as Zhirkov puts it in the context of low-level programming using C, in general, "every pointer is a source of possible memory aliasing" [50].

2.2.1.2 Separation Logic: an extension of Hoare Logic

The need to reason on low-level programs using such idioms prompted the development of Separation Logic (SL) [118, 61], that can be defined as a refinement extension of Hoare Logic [62]. Triples $\{P\}\, c\, \{Q\}$ also exists in Separation logic, with the difference that in this setting P and Q restrict the set of accessible resources for the execution of c, enabling local reasoning.

In particular, P and Q are *heap predicates*; specifications defined using the assertion language describes heaps, that is, parts of the memory. As a first approximation, let us describe heap predicates the following way. A heap predicate can be [62]:

- empty, usually written `emp` or $[]$;
- $r \mapsto v$, a "points-to" assertion atom, also called a resource assertion, describing the *ownership* of a single memory cell allocated at address r, whose content is v;
- $[P]$, a pure predicate characterizing an empty heap and asserting that the proposition P is true;
- $H_1 \star H_2$, describing a heap that can be split into two disjoint parts, one satisfying the heap predicate H_1, the other one satisfying the heap predicate H_2; the $\star$ operator is the *separating conjunction*, sometimes also called the spatial conjunction.

Now, let us take some basic examples. Let us first consider one reference r whose initial value is v and is then assigned the value 42. The following triple holds:

$$\{r \mapsto v\}\; r := 42\; \{r \mapsto 42\}.$$

Indeed, once the assignment has occurred, the reference r points to the value 42. Similarly, let us consider two references r_1 and r_2 whose initial values respectively are v_1 and v_2. Assuming the usual textbook swap function that exchanges the values of two references, the following triple also holds:

$$\{r_1 \mapsto v_1 \star r_2 \mapsto v_2\}\; \mathrm{swap}(r_1, r_2)\; \{r_1 \mapsto v_2 \star r_2 \mapsto v_1\}.$$

Furthermore, this triple can serve as a concise specification for the swap function.

Separation logic can also be leveraged for the formal verification of low-level data structures. Let us use linked lists to give an example of predicates associated with such a data structure. A non-cyclic linked list can be defined using the following textbook recursive predicate, assuming list cells with two fields, respectively to store

data and a pointer to the next cell; we directly reuse the definition from Charguéraud [62].

$$\begin{aligned}\text{MList } L\, p = \text{match } L \text{ with}\\ &\mid \text{nil} \to [p = \text{null}]\\ &\mid x :: L' \to \exists q.\ (p \mapsto \text{data} = x) \star (p \mapsto \text{next} = q) \star (\text{MList } L'\, q)\end{aligned} \tag{2.1}$$

If L is an empty list, then the associated pointer p must be null. Otherwise, let us denote x the head of the list and L' its tail; p points to a cell such that its data field is equal to x and its next field (the pointer to the next cell) is equal to q such that MList $L'\, q$ holds, thereby asserting that the tail of the list is valid in memory.

2.2.1.3 The frame rule and allocation freshness

Beyond specifications with respect to memory ownership inside data structures, one of the things that make Separation Logic very interesting to reason about programs is the *frame rule*, as it allows for modular reasoning.

Informally, the frame rule states that if a program c can be executed correctly using only some part of the memory, then the rest of the memory should remain unaffected by the execution of c.

More formally, if a program c can be evaluated in a state described by P, then, given any heap predicate R, it can be evaluated in a extended heap described by $P \star R$, producing an extended heap described by the corresponding postcondition.

$$\frac{\{P\}\ c\ \{Q\}}{\{P \star R\}\ c\ \{Q \star R\}}\ \textsc{Frame}$$

As an example, one can deduce from the previous example regarding reference assignment the following, only by applying the frame rule to add the heap predicate $r' \mapsto v'$:

$$\{r \mapsto v \star r' \mapsto v'\}\ r := 42\ \{r \mapsto 42 \star r' \mapsto v'\}.$$

In an analogous manner, from the (slightly-adapted) following allocation specification:

$$\{\texttt{emp}\}\ r := \text{alloc}()\ \{r \mapsto \_\},$$

one can deduce the following by applying the frame rule to add the heap predicate $r' \mapsto v'$, thereby deriving a freshness property for the just-allocated reference r [62]:

$$\{r' \mapsto v'\}\ r := \text{alloc}()\ \{r \mapsto \_ \star r' \mapsto v'\}.$$

Indeed, the frame rule actually amounts to an universal quantification on the rest of heap: whatever reference already is allocated, it is disjoint from the just-allocated one.

2.2.1.4 Separation logic and memory management

Regarding previously mentioned memory-safe languages, it should be noted that programs verified using separation logic, even when considering the partial correctness case, are guaranteed to be exempt from memory faults, as noted by Reynolds [61]. In the same research article, Reynolds mentions that Ishtiaq and O'Hearn [119], paraphrasing Milner [120] about its own famous type soundness result, underlined that their triples semantics is aligned with the slogan "well-specified programs [do not] go wrong", even when considering memory disposal.

Beyond allocation, Reynolds, Ishtiaq and O'Hearn proposed in these seminal papers that operations on references such as accessing, updating, allocating and deallocating references can be associated with the following triples, assuming a basic imperative language with heap management [61, 119].

- $\{p \mapsto v\}\ x := \text{read}(p)\ \{[x = v] \star p \mapsto v\}$
- $\{p \mapsto v\}\ p := v'\ \{p \mapsto v'\}$
- $\{\texttt{emp}\}\ r := \text{alloc}()\ \{r \mapsto \_\}$
- $\{p \mapsto v\}\ \text{free}(p)\ \{\texttt{emp}\}$

Overall, separation logic provides a fine-grained tracking of memory resources, using ownership expressed using points-to assertions and separation of resources thereby controlling aliasing using the separating conjunction. In addition to this, there exist other connectives as well as variations over the frame rule [62] that are not needed in the rest of the document.

Finally, we shortly presented a version of separation logic that does not allow one to discard part of the heap, keeping track of all resources: this is well suited for languages relying on explicit memory management. There also are *affine* separation logics that allow such discards, especially useful when considering languages equipped with garbage collection [62].

2.2.1.5 Concurrent Separation Logic

The development of Hoare Logic was at the time followed with numerous developments, including the defining semantics of various sequential programming languages and the support of their features [121]. In addition to these, some other extensions targeted reasoning on concurrent programs, including pioneering work from Susan Owicki and David Gries [122]. In their opinion, it is even more important to reason on concurrent programs relying on parallel programming as nondeterminism can emerge from the "unpredictable order in which actions from different processes are executed".

As the executions of two parallel processes can be interleaved, this can indeed lead to a large number of possible sequences of steps, where each step corresponds to the action of one process. Such interleavings can be described using *interleaving semantics*, assuming a sequentially consistent memory model [123]: interleaving are

made of *atomic* (or "uninterruptible" per Park's wording [124]) processes actions. As noted by Anderson and Gouda [125], this model can be used to treat concurrent programs as sequential nondeterministic ones, which simplifies formal reasoning. Alas, again, pointers limited reasoning on shared-memory concurrent programs.

Concurrent Separation Logic [64, 63] is an answer to the need to reason on concurrent programs using pointers to manipulate shared-memory. This extension over SL brings one main rule, the *parallel composition rule*, generalizing the frame rule.

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \qquad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 \star P_2\}\ c_1 \| c_2\ \{Q_1 \star Q_2\}}\ \textsc{Par}$$

This rule of "disjoint concurrency" [63] also requires that considered threads operate on disjoint portions of memory. More precisely, it is required that the program c_i does not modify free variables of P_j, c_j, Q_j when $i \neq j$.

As noted by Brookes and O'Hearn in their 2016 retrospective paper [126] regarding Concurrent Separation Logic (CSL), CSL has since been enriched by many extensions. One of them is the fact that points-to can be enriched by fractional permissions, notably to support concurrent reads: a reference can be read concurrently without introducing data races, as long as it is not written to at the same time. Last but not least, finer-grained memory sharing can be encoded using (dynamically allocated) invariants. In this setting, each invariant I_P is associated with a separation logic predicate P that is only accessible atomically. This is useful for several synchronization mechanisms, such as locks. Indeed, an invariant I_P can be seen as a lock storing ownership of a predicate P.

2.2.2 An introduction to the F* proof-oriented programming language

2.2.2.1 First definitions

F* is a functional programming language that is specialized for the purpose of *auto-active verification*, a term coined by Leino and Moskal [127] that refers to the combination of interactive and automated proofs. F*'s type system features dependent typing and refinement types. Also presented as a proof-oriented programming language [128] or as a verification-oriented programming language [129], it relies on the Z3 SMT solver [70] to automatically discharge proof obligations.

This language, that can be considered as forming a proof assistant and a program verification framework when including the corresponding toolchain, features effects and tactics support: in both cases, they can be provided by the user, as we will see later in Section 2.2.3.

To familiarize ourselves with F*, let us consider a textbook example: vectors that are indexed by their length.

```
type vec (#a: Type): (n:int{n >= 0}) -> Type =
| Nil: vec #a 0
| Cons: #n: nat-> hd:a -> tl:vec #a n -> vec #a (n+1)
```

Please note that `n:int{n >= 0}` corresponds to a refinement type describing the sets of all integers n such that $n \geq 0$, that is, the type of natural integers that can be abbreviated as `nat` in F*. Additionally, `#n:nat` means that for the `Cons` constructor, this argument is implicit. The following definitions thus both typecheck and are equivalent.

```
let v1 : vec #nat 2 = Cons #nat #1 1 (Cons #nat #0 2 Nil)

let v2 : vec #nat 2 = Cons 1 (Cons 2 Nil)
```

In order to get a glimpse of F*'s proof style and use of automation, let us consider the following example, that allows us to introduce several notions at the same time.

```
1 let rec add (x y: nat): Tot int
2   (decreases x)
3   = match x with
4   | 0 -> y
5   | _ -> 1 + add (x - 1) y
```

On one hand, this example showcases the treatment of integers and the role of the SMT solver with respect to the type system. This recursive function can be automatically typechecked thanks to the SMT solver. More precisely, to typecheck the line 5, the F* language proceeds the following way. First, refinements are unpacked, we get that x has type `int` such that $x \geq 0$. As pattern matching allows one to exclude previous cases, one can deduce from line 4 that $x \neq 0$ in the case corresponding to line 5. This information is sent to the SMT solver that deduces that $x - 1 \geq 0$. At this point, $x - 1$ can then be refined from an `int` to a `nat`.

On the other hand, while F* supports effectful programming, as we will see later, we only consider purely functional programming for now. By default, functions must be total: recursive functions must terminate. The only recursive call here is `add (x - 1) y`: the process through which $x - 1$ is proved to be a natural number is already known; this recursive call is thus valid. Additionally, the use of the `decreases` keyword guides the termination checker to consider a well-founded partial order that is respected by the recursive call in order to prove termination.

As functions must be total by default, the `Tot` effect annotation[14] actually is unnecessary; `decreases` clauses are not always necessary due to built-in heuristics,

[14] Effects in F* are described in Section 2.2.2.4.

see F* manual [130, section 7.4]. Indeed, the same code without the `decreases` successfully typechecks.

This definition is a bit unsatisfactory, as we would like to prove that the result of this function is a `nat`. Let us first examine an extrinsic proof.

```
val add_nats_is_nat (x y: nat)
  : Lemma (add x y >= 0)
```

First and foremost, in F*, lemmas are functions that return the uninformative `()` value of type `unit`, then refined to carry useful information. The `val` keyword allows one to define the prototype of a function: this forms the basis of the F* support for interfaces.

Finally, let us provide an implementation of `add_nats_is_nat`, that is, a proof.

```
let rec add_nats_is_nat (x y: nat)
  = match x with
  | 0 -> ()
  | _ -> add_nats_is_nat (x-1) y

let add_refined (x y: nat): nat =
  add_nats_is_nat x y; add x y
```

This amount to a standard, expected induction proof, that is fairly short.

In this simple case, it would also have been possible to do everything at once using `nat` as the result type. That is, to rely on an (almost empty) intrinsic proof, instead of having to refine the `add` function *a posteriori*.

```
let rec add_nat (x y: nat): nat
  = match x with
  | 0 -> y
  | _ -> add (x-1) y
```

Once again, this works thanks to the SMT solver.

2.2.2.2 Basic interaction with F* and Z3

As previously hinted, the use of F* is interactive. One first way is to use F* is to pass it files on the command line.

```
$ fstar.exe Example.fst
Verified module: Example.fst
All verification conditions discharged successfully
```

However, this does not allow fine-grained interaction: in case of a proof failure, e.g., at the end of the file, attempting to fix this failure through edits to the corresponding proof then requires to pass the whole file again to F⋆.

Another way to use F⋆ is within a text editor or an integrated development environment (IDE) through the use of dedicated plugins, e.g., the `fstar-mode.el` [131] plugin for the Emacs text editor.[15] As noted in F⋆ manual [130, section 2.1], this allows for incremental progress over proofs contained in a file. Let us have a look at what using this plugin looks like.

```
1  module CH2.S22.Example
2
3  let rec add (x y: nat) : Tot int
4    (decreases x)
5    = match x with
6    | 0 -> y
7    | _ -> add (x - 1) y
8
9  val add_nats_is_nat (x y: nat)
10   : Lemma (add x y >= 0)
11
12 let rec add_nats_is_nat x y
13   = match x with
14   | 0 -> assert (False)
15   | _ -> add_nats_is_nat (x-1) y
16
17 let rec add_nat (x y: nat): nat
18   = match x with
19   | 0 -> y
20   | _ -> add (x-1) y
```

In this snippet of code, lines from 1 to 8 (highlighted in light blue) correspond to code that has already been successfully checked by F⋆. Lines 9 to 11 (highlighted in bright orange) correspond to code currently being verified. Lines 12 to 15 (highlighted in light orange) correspond to code currently queued to be later checked. The rest of the file is not queued for verification. As an example, in case of a failure to verify lines 9 to 11, iterating over the corresponding declaration does not require to check again lines 1 to 8. Any modification to already checked code results in the verification status displayed using the previously described text highlighting being updated to account for it.

In the previous paragraph, we only mentioned F⋆; in fact, as previously mentioned, F⋆ heavily relies on Z3 to provide proof automation. To that end, F⋆ generates verification conditions (VCs) in the form of SMT queries associated to considered programs and then calls the Z3 SMT solver to solve them. By default,

[15] There also exists plugins for the Visual Studio Code IDE [132] as well as an unofficial one for the neovim text editor [133].

one single and thus possible large SMT query is sent for each definition to Z3 [130, section 30.1]. As we will shortly see in Section 2.2.2.3, it is possible to inspect SMT queries and analyze their handling by Z3.

Let us introduce a few ways to control and assist the SMT solver's action.

```
3 open FStar.Mul
4 let _ = assert(forall (x:int). x * x >= 0)
```

The `assert` keyword allows one to define an assertion, i.e. some property to be checked. Unlike runtime assertions in other programming languages, in F⋆, these are checked at compile-time, before any execution of the program. The considered example does not involve any specific fact from the ambient set of definitions. This assertion is indeed checked by Z3 using its arithmetic theory to handle non-linear arithmetic (NLA).

Assertions in general play a key role when using F⋆. On one hand, they can have side effects. It is quite common that adding specific assertions can help proofs go through. On the other hand, when using F⋆ interactive mode, contrary to other proof assistant such as the Rocq theorem prover, there is no goal displayed when working on a proof. Assertions thus allow the developer to check that some part of the proof can actually be checked, before moving onto another part. Finally, they can be used to document intermediate steps of the proof.

Assertions add properties to be checked to the (implicit, non-displayed) set of current goals, that is, the set of properties to be checked by F⋆ and Z3. When developing or debugging a proof, it can be convenient to explicitly exclude some properties from this set. This can be done using the `assume` or `admit` keywords.

In the following, with the help of some basic modular arithmetic lemma, let us consider an ongoing proof by case analysis.

```
4 val lemma_mod_mul_distr_l (x y: int) (n: nat{n > 0})
5   : Lemma
6   ((x * y) % n == ((x % n) * y) % n)
7
8 let square_mod4 (n: int)
9   : Lemma ((n*n) % 4 < 2)
10  = match (n % 4) with
11  | 0 -> lemma_mod_mul_distr_l n n 4
12  | 1 -> assume((n*n) % 4 == 1)
13  | _ -> admit ()
```

This proof by cases distinguishes between several cases depending on the value of `n % 4`. The first case where this value is equal to 0 is done, that is, the desired property is proven to hold for that case, provided the lemma's proof is accepted by F⋆. In the case where `n % 4` is equal to 1, the `assume` keyword is used to force F⋆

to consider an intermediate property valid. If the proof is accepted by F⋆ as is, this means that proving this property is sufficient to complete the proof of the lemma in that case. It is noteworthy that this allows the developer to pinpoint the assumed property as remaining work. Finally, in all other cases (where `n % 4` is equal to 2 or 3), abbreviated using the `_` pattern, the `admit` keyword is used to admit the lemma's goal in these cases, thereby temporarily skipping any further proof effort and leaving them entirely as remaining work.

In some cases such as systems programming, external unverified functions may be used. Thus, one may wish to go even further and assume that functions with desired properties exist to further derive proofs from their use. The `assume val` keywords combination can be used to that end. It should be noted that defining external functions prototypes this way amounts to the axiomatization of their properties, which needs to be done with great care. In the following, we give an example about an external function whose purpose is to check whether the multiplication of two 64-bit unsigned integers would overflow. It is modeled as a total function and thus should not have any side effect.

```
3 module U64 = FStar.UInt64
4 open FStar.Mul // import *
5 open FStar.UInt // import fits
6
7 assume val check_u64_mul_overflow
8   (x y: U64.t)
9   : r:bool{r = fits (U64.v x * U64.v y) 64}
```

In this code snippet, `U64.v` is a coercion function from `U64.t` to `nat`. In the rest of this document, the `U8`, `U32`, `U64` and `US` modules respectively correspond to standard library modules providing helpers for 8-bit unsigned integers (`uint8_t` in C), 32-bit unsigned integers (`uint32_t` in C), 64-bit unsigned integers (`uint64_t` in C) and the C `size_t` unsigned integer type. In all of these cases, for any module `U`, `U.t` is the corresponding unsigned integer type and `U.v` is the corresponding coercion function to `nat`.

2.2.2.3 Finer-grained interaction with F⋆ and Z3 to tackle more complex proofs

So far, we presented the use of F⋆ and Z3 without any consideration for time resources, only as an abstract machine that possibly answers "yes". Things actually are a bit more complicated, as the notion of timeout is core to the use of Z3. Indeed, let us recall that SMT stands for the problem of satisfiability modulo theories, which is undecidable for many theories such as nonlinear integer arithmetic. A consequence of this is that, as Zhou et al. [134] put it while focusing on SMT queries from automated program verification, "there is no guarantee that an SMT solver will terminate", hence the need for timeouts. Furthermore, proofs can sometimes be quite long, as dividing lemmas into sublemmas is not always very practical nor

efficient. Same authors [134] nonetheless lay the emphasis on the fact that program verification relying on SMT solving remains iterative in practice, and that the proof developer's available time is a scarce resource. All in all, it is necessary to bound the execution time of Z3 to prevent unreasonable execution times.

Thankfully, it is possible to tweak various Z3 parameters inside proof files using pragmas such as `#push-options` and `#pop-options`. As previously highlighted, the proof developer's available time is important: she may thus be interested in analyzing Z3 handling of SMT requests in order to optimize it. This is doable using using the `query_stats` toggle.

```
15 #push-options "--query_stats"
16 let diophantine_eq_no_sol (_:unit)
17   : Lemma (~ (exists (x y:int). x*x + 1 = 4*y))
18   =
19   let aux (x y: int)
20     : Lemma
21     (requires x*x + 1 = 4*y)
22     (ensures False)
23     =
24     assert((x*x) % 4 == 3);
25     square_mod4 x
26   in
27   Classical.forall_intro_2 (Classical.move_requires_2 aux)
28 #pop-options
```

This new block of code gives a glimpse of quantifier manipulations and nested lemmas. Moreover, the auxiliary lemma `aux` has explicit precondition and post-condition respectively as `requires` and `ensures` clauses. A lemma under the form `Lemma [...]` is actually syntactic sugar for `Lemma (requires True) (ensures [...])`.

As `query_stats` is enabled, typechecking it yields the following log.
`[F* ] (<input>(18,3-27,58)) Query-stats (CH2.S22.AssumeAdmit.diophantine_eq_no_sol, 1) succeeded in 3 milliseconds with fuel 2 and ifuel 1 and rlimit 5 (used rlimit 0.024)`
The Z3 parameters `z3rlimit`, `fuel` and `ifuel` are very commonly tweaked. Let us present them in that order.

When working on some tedious lemma, increasing the `z3rlimit` can be helpful. Indeed, this parameter corresponds to the amount of resources that Z3 can use when faced with some SMT request. It can be seen as a hardware-independent timeout.

```
1 #push-options "--z3rlimit 50"
2 let tedious_lemma ... = ...
3 #pop-options
```

As an example, in this code snippet, `z3rlimit` is increased from the default value of 5 units to 50 units.

Beyond the amount of resources that can be used by Z3, it is also possible to control the way Z3 works. Indeed, as long as definitions are not behind interfaces, the SMT solver has access to the actual body of these definitions. When handling a proof about a recursive function, Z3 is able to unroll definitions: it is possible to control to what extent Z3 actually does it. The `fuel` controls for all definitions how many times they can be unrolled when handling one SMT request. Its counterpart, the `ifuel`, controls inductive types unrolling.

While increasing `fuel` and `ifuel` can help when proving lemmas requiring definition unrolling, whether for concrete values or not, increasing it when not necessary reduces proof performance. Indeed, in such situations, the SMT solver can spend a great amount of time uselessly unrolling functions and types definitions. This is why these parameters are most of the time set to small values, such as 0 or 1.

Regarding concrete values properties, let us consider the following example about computing the length of lists.

```
3  let tl = FStar.List.Tot.tl
4
5  let rec length (#a: Type) (l: list a): nat
6    = match l with
7    | [] -> 0
8    | _ :: t -> 1 + length t
9
10 let l1 = [1]
11 let l3 = [1; 2; 3]
```

The proof programmer may be interested in proving that the length of `l1` and `l3` respectively are 1 and 3. In what follows, the `[@@ expect_failure]` is the syntax corresponding to specify that a definition is expected to fail, so that the corresponding failure does not stop the verification of the considered file.

```
13 #push-options "--fuel 0 --ifuel 0"
14 [@@ expect_failure]
15 let _ = assert(length l1 = 1 + length (tl l1))
16
17 #push-options "--fuel 1"
18 let _ = assert(length l1 = 1 + length (tl l1))
19 // With fuel 1, this still fails
20 [@@ expect_failure]
21 let _ = assert(length l1 = 1)
22
```

```
#push-options "--fuel 2"
// With fuel 2, this finally works
let _ = assert(length l1 = 1)
```

Finally, it is important to mention two last things. First, that it is possible to set specific `fuel`/`ifuel` parameters for specific assertions, in a quite fine-grained manner, when facing difficult proofs. Also, that F⋆ ships a normalizer, a "reduction machinery" [130, section 1.4] capable to computationally reduce symbolic F⋆ terms but also to actually fully compute such terms in applicable cases. Here, `assert_norm` relies on it.

```
// Modifying fuel for a specific assertion
#push-options "--fuel 0"
let _ = assert(length l1 = 1) by (let open FStar.Tactics in set_fuel 2; ())

// Resorting to the normalizer can help...
#push-options "--fuel 0"
assume val a: Type
assume val e1: a
assume val e2: a
assume val e3: a
let _ = assert_norm(length [e1; e2; e3] = 3)

// ...but not in all cases...
[@@ expect_failure]
let f (#a: Type) (l: list a{length l = 3})
  = assert_norm(length l = 3)
// ...even simple ones!
let g (#a: Type) (l: list a{length l = 3})
  = assert(length l = 3)
```

At this point, further diving into the wide array of strategies to improve F⋆ and Z3 performance is not necessary. We restrict us to mention two noteworthy points. On one hand, more information can be found about the use of Z3 by F⋆ in F⋆ manual [130, Chapter 47], including marking definitions as opaque, designing SMT patterns, and profiling Z3. As a last resort, F⋆'s SMT encoding and resulting SMT queries sent to Z3 can be inspected using the `log_queries` toggle. This allows one to save SMT queries in a dedicated file on a per-definition basis, even though part of the encoding corresponding to the ambient set of definitions may be included. On the other hand, this very useful source of information does not yet include documentation about the builtin F⋆ profiler, that allows one to get information about the execution time of various F⋆ internal components beyond Z3 execution time.[16]

[16] For example, through the use of the `timing` toggle and the following set of F⋆ options providing more precise information:
`"--profile '*' --profile_component '*' --profile_group_by_decl"`.

Equipped with many ways to interact with F⋆ and Z3, we already hinted at the use of `assert` and `admit` to develop and debug proofs. There exists many verification frameworks relying on SMT solvers, whether natively or not, thereby providing partial or full proof automation: studying, improving and debugging such proofs is a broad topic in itself [134]. In this context, we lay the emphasis on one very basic F⋆ strategy that cannot be overlooked: the so-called "sliding admit" verification style [135]. Given one incomplete proof to be debugged or developed, assuming its signature typechecks, let us describe this strategy. First, insert the `admit` at the beginning of the proof body. Then, let it "slide" along this very same block of code while progressively ensuring that all intermediate proof steps before the `admit` use actually are successful. Such proof steps may rely on `assert` or even `assume` keywords, e.g., to pinpoint currently missing proof steps. At the end of the process, the considered proof shall be improved if not complete. This technique allows one to determine where proofs fail and thus refine proofs incrementally.

2.2.2.4 Effects

The programs considered so far are purely functional and their termination is checked by F⋆. As previously mentioned, F⋆ also supports a system of effects, enforcing that a given *computation type* can only depend on some specific computation types [130, Chapter 25]. As we will see later, F⋆ supports user-defined effects; in the meantime, let us introduce F⋆ primitive effects.

Total computations. The very first F⋆ effect is the effect of total computations, `Tot`. It lies at the bottom of the effect hierarchy: total computations can only depend on other total computations. As previously seen, the effect of total computations is the default one for function declaration. It is accompanied by the `Pure` effect, very similar, albeit allowing one to explicitly specify precondition and postcondition of total computations [130, section 29.3].

Ghost computations. Proof-oriented programming in F⋆ requires one to prove programs correct with respect to their specifications in a way that F⋆ accepts it. In this context, programs are enriched by ghost terms that are irrelevant with respect to the actual computation. These computationally irrelevant parts of the program do not need to be part of the final, compiled program: they can thus be erased at compile-time. The effect of ghost computations, `GTot`, allows one to separate computationally relevant and computationally irrelevant parts of the program. By definition, a ghost term cannot interfere with the behaviour of a total term.

Using ghost terms as part of specifications or for proof purposes is very common: F⋆ provides the user with a specific library, `FStar.Ghost` [136]. Ghost terms are encapsulated values encompassing `GTot` computations and erased values of type `erased t`, where `t` can be any type.

```
val erased (t: Type): Type

val reveal (#a: Type) (v: erased a): GTot a
val hide (#a: Type) (v: a): Tot (erased a)
```

`reveal` and `hide` functions provide a bijection between `a` and `erased a`. These functions can be used to relate proof-related ghost values and data actually handled by programs.

```
val hide_reveal (#a: Type) (x: erased a)
  : Lemma (ensures (hide (reveal x) == x))
  [SMTPat (reveal x)]
val reveal_hide (#a: Type) (x: a)
  : Lemma (ensures (reveal (hide x) == x))
  [SMTPat (hide x)]
```

To facilitate such type coercions, SMT patterns associated to the `hide_reveal` and `reveal_hide` lemmas are also provided. These two lemmas can thus be automatically instantiated by Z3 to relate a value `v` and its hidden counterpart `hide v` [130, section 27.4].

Crucially, `Tot` can be used as part of `GTot` computations. Due to possible soundness issues, the converse is not possible, except when the underlying type of the `GTot` computation is not informative [130, section 27.5]. In this specific case, effect promotion from `GTot` to `Tot` is possible, as demonstrated in the following.

```
open FStar.Ghost

// Tot used within GTot
assume val f1: nat -> Tot nat
let f2 (x: nat): GTot nat = hide (f1 x)

// GTot used within Tot, specific case
assume val f3: nat -> GTot nat
let f4 (x: nat): Tot (erased nat) = f3 x
```

In this code snippet, we first pick a total function `f1` from natural numbers to natural numbers. `f1` is used to define a function `f2` from natural numbers to ghost natural numbers, thus demonstrating the effect promotion from `Tot` to `GTot`. We then pick a ghost function `f3` from natural numbers to ghost natural numbers, that is used to define a total function `f4` from natural numbers to erased natural numbers. This demonstrates the effect promotion from `GTot` to `Tot`, as `erased nat` is a non-informative type.

Non-termination. In F⋆, total and ghost functions must terminate. Its effect system allows one to express non-terminating or divergent computations, separating such computations from its terminating core. It is the role of the effect of divergent computations, `Dv`.

This case is interesting as it allows us to make a reminder that will be useful in the next section. On one hand, as stated in F⋆ manual [130, section 28.2], the `Tot` effect has a *total correctness* semantics: if a term `e` has type `Tot t`, then the evaluation of `e` terminates and produces a value of type `t`. On the other hand, the `Dv` has a *partial correctness* semantics: if a term `e` has type `Dv t`, then the evaluation of `e` may loop forever; in the case where it terminates, then the resulting value has type `t`.

This allows one to write `Dv` computations like the following one, further motivating careful separation of this effect from the logical core of F⋆.

```
3 let rec loop_false () : Dv False = loop_false ()
```

Indeed, `Tot` is a subeffect of `Dv`: no `Tot` computation can depend on the `Dv` effect; this example thus does not lead to a proof of `False` in F⋆, thankfully. Finally, it should be noted that it is not possible to do extrinsic proofs of `Dv` computations, as specifications of lemmas rely on total computations.

2.2.3 Steel: a Concurrent Separation Logic embedded in F⋆

Steel is a higher-order, dependently-typed concurrent separation logic embedded within the F⋆ programming language. It provides the end user with a high level of automation: verification conditions (VCs) are partially discharged by a mixture of tactics and SMT solving [137].

Steel builds upon SteelCore [138], a semantics for a higher-order, dependently-typed Separation Logic that has been formalized in F⋆, relying on F⋆'s support for effectful programming. SteelCore provides partial correctness proofs of concurrent programs and support for user-defined partial commutative monoids (PCMs) as well as dynamically allocated invariants to encode various memory sharing idioms. Its soundness is derived from a definitional interpreter defined in F⋆ using effectful programming, modeling the execution of programs (or computations) as a non-deterministic interleaving of atomic actions. In this setting, computations are described as concurrent, potentially divergent and stateful using a Hoare triple alike type `SteelCore` a p q, where a is the return type and p and q correspond to separation logic precondition and postcondition. Noting as `vprop` the type of separation logic predicate, p is of type `vprop` and q of type $a \rightarrow$ `vprop`, possibly depending on the returned value.

Steel is the result of efforts to improve automation and thus usability in comparison to SteelCore. To this end, it describes computations using quintuples of the following form.

```
Steel a (p: vprop)
        (q: a → vprop)
        (requires (r: pre p))
        (ensures (s: post p a q))
```

The newly added indices r and s are called *selector predicates*: they only depend on the memory content of the initial memory described by p and the final memory described by q. In this setting, `pre` p corresponds to the type of functions that take as argument the heap content of a heap specified by p and return a first-order logic predicate. Similarly, `post` p a q corresponds to the type of functions that take as arguments (1) the heap content of a heap specified by p, (2) a value of type a, (3) the heap content of a heap specified by q and return a first-order logic predicate.

The key idea is that this restriction enables the decoupling of proof obligations respectively related to memory shape and memory content. On one hand, proof obligations related to memory shape correspond to a frame inference problem: these can be partially discharged by a dedicated tactic. On the other hand, proof obligations related to memory content can be encoded using first-order logic and can thus be partially discharged using SMT solving.

In this section, we focus on presenting Steel from a client point of view with respect to the automation support it brings, that is, on its practical consequences for the proof programmer.

2.2.3.1 First examples

Let us consider the textbook swap function again, recalling that its specification using standard Separation Logic can be expressed using the following triple.

$$\{r_1 \mapsto v_1 \star r_2 \mapsto v_2\}\ \mathrm{swap}(r_1, r_2)\ \{r_1 \mapsto v_2 \star r_2 \mapsto v_1\}$$

In this setting, memory shape and memory content specifications are blended altogether. Let us know compare this with the Steel signature of the same swap function.

```
1 val swap (r1 r2: ref int)
2   : Steel unit
3   (vptr r1 ⋆ vptr r2)
4   (fun _ -> vptr r1 ⋆ vptr r2)
5   (requires fun h0 -> ⊤)
6   (ensures fun h0 _ h1 ->
7     h1[r1] == h0[r2] ∧
8     h1[r2] == h0[r1]
9   )
```

In this code snippet, r_1 and r_2 of type `ref int` are mutable references, analogous to that of OCaml. The rest of the signature is as follows. On one hand, lines 3

and 4 correspond to the first couple of precondition and postcondition related to memory shape. In each of these lines, the validity of a reference in the considered memory (initial or final) is expressed using the `vptr` selector predicate. It should be noted that `vptr` only specifies the shape of a fragment of the heap, not the underlying content of the reference specified to be valid. The context only is made of two valid references r_1 and r_2, both at the beginning and the end of the execution of the function. On the other hand, lines 5 to 9 correspond to the second couple of precondition and postcondition related to memory content: the precondition is trivial, and the postcondition asserts that the content of the two references has been swapped, using the initial heap h_0 and the final heap h_1.

Steel authors report on the fact that implementing this function can be done in a very straightforward manner using the automation building upon the separation between memory shape and memory content [137], in comparison with the implementation using SteelCore. As a matter of fact, this example does not require any proof annotation, see the following code.

```
1 let swap r1 r2 =
2   let x1 = read r1 in
3   let x2 = read r2 in
4   write x1 r2;
5   write x2 r1
```

Steel supports a counterpart to the `admit` keyword used for total computations: the `sladmit` keyword. This can be used to specify that any further proof effort regarding memory shape proof obligations should be skipped, which is convenient when developing or debugging proofs.

2.2.3.2 Combinators

This separation between proof obligations related to memory shape and memory content is not always seamless. Indeed, in many cases the memory shape of the heap can depend on the memory content of some other part of the heap. As an example, let us consider textbook linked-lists.

Using textbook separation logic, as previously seen in Section 2.2.1.2, the separation logic predicate corresponding to textbook linked-lists would be:

```
1 type cell (a: Type0) = {
2   data: a;
3   next: ref (cell a);
4 }
5 let t (a:Type0) = ref (cell a)
6
7 let rec llist (p: t a) (l: list a) = match l with
8 | [] -> pure (p = null)
9 | x :: tl -> ∃q. pts_to p {data = x; next = q} ⋆ llist q tl
```

It should be noted that the use of an existentially quantified variable q (line 9) is required here since `llist` is a pure separation logic predicate, that is, not linked to any actual memory. Thus, directly accessing the `next` field of the cell pointed to by `p` is not possible as part of this definition.

To account for the dependency between different separation logic predicates, Steel relies on combinators. A first very useful combinator is `vdep`, used to express a dependency between the memory content of one predicate and another predicate; its selector is a dependent pair made of the memory content of both predicates. To ensure that separation logic predicates provide properly typed memory contents, they are enriched with a type field specifying the type of the corresponding selector, that can be retrieved using `t_of`. Such separation logic predicates are of type `vprop`.

```
1 val vdep (v: vprop) (p: (t_of v) -> vprop) : vprop
```

Using `vdep`, one can encode linked lists the following way, though it is partly unsatisfying, as we shall see.

```
1 let rec llist (#a: Type0) (ptr: t a) (n: nat)
2   : Tot vprop
3   (decreases n)
4   =
5   if (n = 0)
6   then pure (is_null ptr)
7   else vdep (vptr ptr) (fun v -> llist v.next (n-1))
```

The `n` natural number is here used to prove the termination of `llist` and is actually equal to the length of the underlying list: the interested reader can find corresponding details in Appendix A.1. Yet, this definition of `llist` does not provide us with a selector, that is, a way to retrieve memory.

To this end, a useful combinator is `vrewrite`, used to rewrite the memory content of one predicate, e.g., to get a suitable selector.

```
1 val vrewrite (v: vprop) (#t: Type) (p: (t_of v) -> GTot t)
2   : r:vprop{t_of r == t}
```

As we are implementing linked lists, the expected selector is of type `list a`. Refining previous implementation using the `vrewrite` combinator, we obtain the following.

```
1 let rec llist (ptr: t a)
2   : Tot (sl:vprop{t_of sl == list a})
3   =
4   if (n = 0) then (
```

```
    pure (is_null ptr) `vrewrite' (fun _ -> [])
  ) else (
    vdep (vptr ptr) (fun v -> llist v.next)
    `vrewrite'
    (fun x -> (dfst x).data :: (dsnd x))
  )
```

In particular, while the case analysis is on `n` and not on the associated list, it is equivalent to the previous textbook definition: the interested reader can find details in Appendix A.2. In the basic case, where the pointer is null, we want the associated selector to be an empty list. In the other case, we know that the memory content `x` of the predicate `vdep (vptr ptr) (fun v -> llist v.next)` is a dependent pair, whose first and second element are returned by the `dfst` and `dsnd` functions. As the goal is to retrieve a list of type `list a` and not `list (cell a)`, only the data field of the cell associated to `ptr` corresponds to the intended head of the list.

Let us now build the `cons` operator on top of this separation logic predicate. To this end, handling `vdep` and `vrewrite` combinators is required; for this purpose, each combinator comes with both an introduction and an elimination function. The following code makes use of `SteelGhost`, the effect corresponding to ghost, computationally irrelevant `Steel` computations.

```
val intro_vdep (v: vprop) (q: vprop) (p: (t_of v) -> vprop)
  : SteelGhost unit
  (v `star` q) (fun _ -> vdep v p)
  (requires fun h -> q == p (h v))
  (ensures fun h _ h' -> let x2 = h' (vdep v p) in
    q == p (h v) /\
    dfst x2 == (h v) /\
    dsnd x2 == (h q)
  ))

val intro_vrewrite (v: vprop) (f: (normal (t_of v) -> GTot t))
  : SteelGhost unit
  v (fun _ -> vrewrite v f)
  (requires fun _ -> True)
  (ensures fun h _ h' -> h' (vrewrite v f) == f (h v))
```

In turn, these can be used to implement `cons`, whose unsurprising specification is met by using both `intro_vdep` and `intro_vrewrite`.[17] In the following code, `sel` is the selector associated with `vptr` predicates describing references; `v_llist` is the selector associated with the previously defined linked list predicate.

[17] Few details have been omitted and are included in Appendix A.1.

```
let cons (#a: Type0) (hd: t a) (tl: t a)
  : Steel unit
  (vptr hd `star` llist t)
  (fun _ -> llist hd)
  (requires fun h0 -> True)
  (ensures fun h0 r h1 ->
    let v : cell a = sel hd h0 in
    let l0 : list a = v_llist tl h0 in
    let l1 : list a = v_llist hd h1 in
    l1 == v.data :: l0
  )
  =
  let c = read hd in
  write hd {data = c.data; next = tl};
  (**) intro_vdep
    (vptr hd)
    (llist tl)
    (fun v -> llist v.next);
  (**) intro_vrewrite
    (vdep (vptr hd) (fun v -> llist v.next))
    (fun x -> (dfst x).data :: (dsnd x));
```

In addition to that, there exists yet another builtin combinator called `vrefine`, that can be used to refine the memory content using pure predicate.

```
val vrefine (v: vprop) (p: (t_of v) -> prop)
  : r:vprop{t_of r == x:t_of v{p x}}
```

Finally, Steel supports user-defined combinators, which can be useful, as we will see in Chapter 3.

2.2.3.3 Concurrency support

Steel is a Concurrent Separation Logic: as such, it allows one to write and prove correct shared-memory low-level concurrent programs. Parallel computations can be implemented using the `par` combinator, composing two different threads executing code operating on disjoint regions of memory in parallel, following the PAR rule presented in Section 2.2.1. However, using only `par` would not be very interesting as it does not provide fine-grained memory sharing.

Building upon SteelCore, Steel inherits its expressive memory model based on typed, higher-order references. Each allocation is associated with a user-defined PCM (partially commutative monoid): we refer to the SteelCore paper for more details [138]. In practice, fractional permissions are commonly used: the basic ideas is to extend resource assertions such as points-to assertions with a number $p \in (0, 1]$.

Partial ownership ($p < 1$) only gives read access to the corresponding resource; full ownership ($p = 1$) gives read and write access to it. Equipped with this, resources can be shared between two different threads, such as references expressed using points-to assertions. Let us have a look at a subset of corresponding APIs provided by the `Steel.Reference` library [139].

```
1 val readp (#a:Type0) (r:ref a) (p: perm) : Steel a
2   (vptrp r p) (fun _ -> vptrp r p)
3   (requires fun _ -> True)
4   (ensures fun h0 x h1 ->
5     h0 (vptrp r p) == h1 (vptrp r p) /\
6     x == h1 (vptrp r p))
```

The `readp` function allows one to read a reference for which it only has partial ownership, refining the previously presented API. Line 5 is noteworthy, as one may wonder why it is required to specify that the memory content of the considered reference is left unchanged. This line is necessary due to the fact that, while the content of total definitions is in general freely accessible and may be freely unrolled, e.g., by the SMT solver, definitions whose effect is `Steel` are "opaque" to the rest of the code. More precisely, it is not possible to do extrinsic proofs of `Steel` computations. The signature must thus specify all properties required by other functions. As a final comment regarding `readp`, it is noteworthy that frame equalities (such as the one used line 5) are stronger than selector equalities. Indeed, a selector does not necessarily describe the entire memory content corresponding to a selector predicate.

```
1  let vptr r = vptrp r full_perm
2  val write (#a:Type0) (r:ref a) (x:a) : Steel unit
3    (vptr r) (fun _ -> vptr r)
4    (requires fun _ -> True)
5    (ensures fun _ _ h1 -> x == sel r h1)
6
7  val share (#a:Type0) (#p: perm) (r:ref a)
8    : SteelGhost unit
9    (vptrp r p)
10   (fun _ -> vptrp r (half_perm p) `star` vptrp r (half_perm p))
11   (requires fun _ -> True)
12   (ensures fun h _ h' -> h' (vptrp r (half_perm p)) == h (vptrp r p))
```

The `write` function is the one previously used to implement the textbook swap function: in particular, it outlines that by default, references are used in full-ownership settings. Finally, the `share` function is as expected, in addition to its counterpart `gather` here omitted for brevity: they both allow one to modify permissions.

In many cases, however, several threads may require read and write access to a same memory location. This requires additional synchronization mechanisms, e.g., locks. Steel supports locks through their encoding as dynamically allocated invariants. Skimming over details of the encoding, the API to use locks is the following

one, provided by the `Steel.SpinLock` library [140]. `SteelT` is an abbreviation for `Steel` functions with trivial precondition and postconditions.

```
val new_lock (p:vprop)
  : SteelT (lock p) p (fun _ -> emp)

val acquire_lock (#p:vprop) (l:lock p)
  : SteelT unit emp (fun _ -> p)

val release_lock (#p:vprop) (l:lock p)
  : SteelT unit p (fun _ -> emp)
```

Let us briefly describe these functions. First, creating a lock associated to a separation logic predicate `p` requires to have this predicate `p` in the context. As a result of the creation of the associated lock, `p` is no longer in the context at the end of the execution of the function. Once created and returned, the lock controls the access to the objects described by `p`. Second, acquiring a lock `l` associated to a predicate `p` comes with a trivial precondition, as this process amounts to waiting until the lock becomes available. This can be naively implemented as a loop until the lock becomes available; the function then terminates and `p` is added in the context.[18] Finally, releasing a lock `l` associated to a predicate `p` requires to have `p` in the context. Releasing the lock amounts to giving back the ownership of `p` to the lock and thus removing `p` from the context.

Unfortunately, the use of mutexes is not always suitable to implementation constraints. Indeed, using locks induces a performance overhead in case of lock contention, when a large number of threads try acquiring the same lock. Additionally, using locks improperly may incur deadlocks at run-time, where no further progress can be made by executing program due to impossible lock acquisition. In some cases, it may thus be preferable to use lock-free algorithms, that do not require mutexes.

Implementing lock-free algorithms is done most of the time by relying on hardware support to implement lock-free data structures using atomic operations (such as compare-and-swap). Between the `Steel` and `SteelGhost` effects, there is one additional subeffect layer called `SteelAtomic` that can be used to model atomic operations, such as the aforementioned compare-and-swap (CAS).

```
val cas_u32 (r: ref uint32_t) (v_old v_new: uint32_t)
  : SteelAtomic bool
  (vptr r) (fun _ -> vptr r)
  (requires fun _ -> True)
  (ensures fun h0 b h1 ->
    let v0 = sel r h0 in
    let v1 = sel r h1 in
    if b then v1 = v_new /\ v0 = v_old else v1 = v0
  )
```

[18] This strategy results in a specific sort of lock called a *spinlock*, relying on busy waiting. Most of the time, mutexes should be preferred [50, sections 17.8.8 and 17.8.9].

Finally, in some other cases, it can also be useful to directly reason using dynamically allocated invariants to ensure a form of logical synchronization: this can be useful to initialize data structures, as we will see in Section 3.4.2.

2.2.4 Extraction of Steel programs to C code using KaRaMeL

Support for extracting a subset of F⋆ to C originated with the Low⋆ language [141], that has notably been used to implement a formally verified cryptographic library called HACL⋆ [142], among other verification projects. Extraction of Low⋆ code is done through a dedicated compiler called KaRaMeL [143], that aims at producing readable C. Since then, KaRaMeL has been adapted to also support the extraction of Steel programs.

Let us reuse the swap function as an example.

```
module CH2.S24.Swap

module U64 = FStar.UInt64
open Steel.Reference
open Steel.Effect
open Steel.Effect.Atomic

let swap (r1 r2: ref U64.t)
  : Steel unit
  (vptr r1 ⋆ vptr r2)
  (fun _ -> vptr r1 ⋆ vptr r2)
  (requires fun h0 -> True)
  (ensures fun h0 _ h1 ->
    sel r1 h1 == sel r2 h0 /\
    sel r2 h1 == sel r1 h0
  )
  =
  (**) slassert (vptr r1);
  (**) slassert (vptr r2);
  // get current heap's state
  (**) let h0 = get () in
  let v1 = read r1 in
  let v2 = read r2 in
  (**) assert (sel r1 h0 == v1);
  (**) assert (sel r2 h0 == v2);
  write r1 v2;
  write r2 v1;
  // get current heap's state
  (**) let h1 = get () in
  (**) assert (sel r2 h1 == v1);
  (**) assert (sel r1 h1 == v2)
```

```
#include "CH2_S24_Swap.h"



void CH2_S24_Swap_swap(uint64_t *r1,
                       uint64_t *r2)
{
  uint64_t v1 = *r1;
  uint64_t v2 = *r2;
  *r1 = v2;
  *r2 = v1;
}
```

In the left-hand column, the entire F* code corresponding to a monomorphized Steel implementation of the swap function on 64-bit integers is shown. Assertions related to the shape of the heap (in the form of `slasserts`) and to the content of the heap (in the form of `assert`s), that can be used as part of proof debugging, have been deliberately retained in the presented code. Corresponding lines have been prefixed with `(**)`, as these are erased and not extracted to C. In the right-hand column, corresponding C code after extraction by KaRaMeL is shown.

In the rest of this section, we briefly present additional KaRaMeL extensions that were required to aim at the verification of a realistic memory allocator.

2.2.4.1 Implementation-defined `size_t` and `ptrdiff_t` integer types

As previously presented in Section 2.1.1.1, `size_t` is an unsigned integer type that can store the maximum size of any object, used in the setting of memory management to store sizes and alignments of objects. Its bit width is implementation-defined, mandated since C99 to be at least 16.

In some cases, pointer subtraction can be very useful, as we will see in Section 3.1. This special case of pointer arithmetic is only defined for pointers belonging to the same underlying array. The corresponding result must be stored using a dedicated C signed integer type called `ptrdiff_t`. Its bit width is implementation-defined, mandated since C23 to be at least 16.

16-bits integers are not enough to store the size of objects commonly allocated by client programs. The use of values beyond this default bit width is thus precisely axiomatized and guarded in C code using static assertions. The role of these static assertions is to verify at compilation-time that implementation-defined limits of `size_t` and `ptrdiff_t` are suitable. Functions introducing these static assertions are included in the `Steel.Array` library [144] and correspond to special cases of KaRaMeL extraction.

2.2.4.2 Arrays

Arrays are represented using pointers associating consecutive cells with a single fractional permission. The corresponding memory content is of type `seq a`, that is, sequences of type `a`.

```
1 val varrayp (arr: array a) (p: perm)
2   : r:vprop{t_of vprop == seq a}
```

Offsets used to access arrays are of type `size_t`; concurrent reads in case of partial ownership are supported. In addition to permission-based sharing, spatial sharing is supported: arrays can be split in two spatially disjoint parts. In turn, this requires pointer arithmetic support; more specific pointer arithmetic making use of the `ptrdiff_t` type is also supported.

A last feature related to arrays that was added to the Steel framework and toolchain is top-level immutable arrays; these are a common C feature that was until then missing from Steel. Support for those was added, and they compile with the expected `const` qualifier. Corresponding APIs are implemented as pure functions: this design makes those top-level arrays easy and convenient to use. By virtue of being immutable, one can retrieve the corresponding values, and show that they remain unaffected by (necessarily) disjoint updates, without any tedious memory reasoning. This mechanism relies on a custom `vprop`, which is duplicable and supports concurrent accesses.

2.2.4.3 Locks

The POSIX thread (also called pthread) API for locks is the following, as part of the commonly-used `<pthread.h>` header.

```
1 int pthread_mutex_init(pthread_mutex_t* mutex, [...]);
2 int pthread_mutex_lock(pthread_mutex_t* mutex);
3 int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

This results in a mismatch between the axiomatization of locks, where mutexes are value, and the corresponding C code, operating on corresponding memory addresses. In the pthread library, the type of locks is `pthread_mutex_t`. Unfortunately, locks are not identified with a unique identifier that would be contained within that value, but rather, by the address of the value itself. This means that a simple `let y = x` in F* yields, once extracted to C, a statement of the form `pthread_mutex_t y = x;` which effectively generates a new lock with a different identity from `x`. Worse, the model of locks in Steel generates prototypes in C, such as the following one.

```
1 void acquire_lock(pthread_mutex_t p);
```

which then requires an implementation. Here, this simply cannot be done, as calling `pthread_mutex_lock(&p)` would operate on a copy of the original lock passed by value via the function call. Fixing the Steel model is not possible; it would require materializing the lock itself as an `vprop`, preventing us from passing the lock around.

Fixing this issue relies on an existing nanopass in the Steel-to-C KaRaMeL compiler, namely the struct-by-address pass, initially implemented to target the CompCert compiler, and covered by the original soundness result [141, Appendix E.6]. This pass ensures that function arguments of struct types are always passed by reference and never by copy, which in turn avoids relying on an unverified compilation pass in the CompCert frontend. This pass was extended to also operate on lock values, meaning the signature of `acquire_lock` in Steel now produces a C prototype of the form `void acquire_lock(pthread_mutex_t *p);`, which avoids the

earlier issue. A translation validator was then added that errors out if any subexpression ever materializes at type `pthread_mutex_t`. This means that once a lock is declared, it may only be passed around by address, and never by value. This reduces the soundness of the lock compilation scheme to that of the (trivial) validator. The validator then identified situations that *did* lead to locks being copied. Those essentially stemmed from F* lifting effectful subexpressions into their own let-bindings, because of its monadic effects. Those were eliminated via a series of one-liner cosmetic optimizations. For instance, in KaRaMeL's AST, expressions such as `let tmp; new_lock(&tmp); *p = { *p with lock = tmp }` are now rewritten into `new_lock(&p->lock)`.

Chapter 3

StarMalloc: Verifying a Modern, Hardened Memory Allocator

> *I fought to bring the software legitimacy so that it – and those building it*[1] *– would be given its due respect and thus I began to use the term 'software engineering' to distinguish it from hardware and other kinds of engineering, yet treat each type of engineering as part of the overall systems engineering process. When I first started using this phrase, it was considered to be quite amusing. It was an ongoing joke for a long time. They liked to kid me about my radical ideas. Software eventually and necessarily gained the same respect as any other discipline.*
>
> Margaret Hamilton

This chapter is adapted from the eponymous article published at OOPSLA'24 [1].

In this chapter, we present StarMalloc, a verified, efficient, security-oriented and concurrent memory allocator. This verification work was entirely performed in F* using the Steel separation logic framework. In Section 3.1, we first present StarMalloc's architecture, whose design is inherited from the hardened-malloc memory allocator and thus is not part of our contributions. Then, we proceed with presenting our main contribution: a verification methodology that, beyond challenges inherent to the outlined architecture, allows easily updating and extending the allocator implementation while limiting the overhead of adapting corresponding proofs. In Section 3.2, we outline the verification of a bare memory allocator through modular abstractions and generic predicates. In Section 3.3, we demonstrate that our verification methodology is well-suited for iterative verification, targeting performance and hardening improvements over the bare allocator. In Section 3.4, we show that, through genericity and partial evaluation, we obtain a configurable memory allocator. In Section 3.5, we present our functional correctness theorems as well as the axiomatizations used, that are part of the trusting computing base (TCB).

[1] Mostly women at the time, as hardware was considered more prestigious.

3.1 Background: StarMalloc's architecture

In this section, we present StarMallos's design top-down through its hierarchical levels and describe the sequence of steps taken in order to fulfill memory allocation and deallocation requests.

StarMalloc's design is entirely inherited from that of hardened-malloc [100], that is, a modern security-oriented allocator providing the standard malloc API as well as some additional extensions, such as Android-specific primitives and support for memory tagging. This allocator, itself originally inspired by the OpenBSD malloc[2] has been developed to be used within a security-oriented Android distribution called GrapheneOS [146]. Our choice of hardened_malloc as a baseline stems from several reasons: its advanced design and hardening features raise verification challenges representative of modern allocators, and its emphasis on security makes formally ruling out implementation bugs even more attractive than for other allocators. The design choices especially aim for low fragmentation, low contention, security by default, and good long-term performance and scalability. While the high-level architectures are identical, some differences however exist between these two allocators and are discussed in Section 4.1.2 as part of the evaluation.

StarMalloc is actually composed of two allocators, retaining hardened_malloc's design: one allocator handling regular allocations and one allocator handling large allocations. The threshold used to distinguish between regular allocations and large allocations is the size of a system page, hardcoded to 4KiB in StarMalloc's implementation. As a consequence, any allocation strictly larger than 4KiB will be handled by the "large" allocator.

In the following, we will successively describe: the architecture of the large allocator in Section 3.1.1, the architecture of the allocator handling regular allocations in Section 3.1.2, as well as security mechanisms integrated as part of these two allocators in Section 3.1.3.

3.1.1 Large allocations: a system call wrapper

We first describe the allocator handling large allocations, as it is simpler than StarMalloc's other allocator. For this reason, it actually is the one we first tackled.

This allocator amounts to a system call wrapper. Indeed, allocation and deallocation requests are directly forwarded to the underlying operating system[3] through the `mmap` and `munmap` system calls, that can be roughly described as the system calls equivalent for `malloc` and `free` [89, p.1437 and p.1490]. These system calls, especially `mmap`, can be used in various ways and have in the general case complex specifications. We rely instead on specialized versions of `mmap` and `munmap`, see Section 3.5.3, that have the following C signatures and specifications, handling `uint8_t*` pointers instead of `void*` ones.

- `uint8_t* mmap_u8(size_t length)`: allocate a memory mapping (contigu-

[2] Otto Moerbeek's malloc released in 2008, replacing phkmalloc since OpenBSD 4.4 [145].
[3] Assumed to be a Unix-like OS.

ous pages) whose length in bytes is at least the specified `length` and return the corresponding pointer; due to arguments provided to `mmap`, zero-initialized pages are provided.

- `void munmap_u8(uint8_t* addr, size_t length)`: deallocate all mappings (pages) of the specified address range: [`addr`, `addr` + `length`).

3.1.1.1 Large allocator: allocation and deallocation

Allocation process. Let us describe the allocation process, e.g, initiated by a `malloc` call.

1. The client program requests `size` bytes through the C standard library `malloc` function, that points to the StarMalloc's `malloc` implementation in our setting.
2. As `size` > 4096, this allocation request is deemed to be a large allocation request and is forwarded to the "large" allocator.
3. The large allocator requests the operating system for memory to fulfill this allocation request. This is done through the `mmap` system call that returns a pointer `ptr` to the just-created memory mapping. If `ptr` is a null pointer, allocation stops and a null pointer is returned.
4. The large allocator stores the pair (`ptr`, `size`) inside an associative data structure storing all of the allocator's metadata.
5. The value `ptr` is returned to the user.

Deallocation process. Similarly, let us describe the deallocation process, e.g., initiated by `free`.

1. The client program requests some a pointer `ptr` to be deallocated through the C standard library `free` function, that points to the StarMalloc's `free` implementation in our setting.
2. Given `ptr`, StarMalloc can check whether this allocation corresponds to a possibly regular allocation or not[4]; if not, deallocation request is forwarded to the large allocator.
3. The large allocator checks whether `ptr` corresponds to a valid large allocation, that is, whether `ptr` corresponds to a large allocation that has not been yet deallocated. This is done by checking whether `ptr` is one of the keys stored in the associative data structure storing metadata.
    - If is this is not the case, the deallocation process stops.
    - Otherwise, it continues and the size of the corresponding valid allocation `size` is retrieved.

[4] As we will see, all regular allocations live in one large memory segment.

4. The large allocator requests the operating system through the `munmap` system call to remove the $[\texttt{ptr}, \texttt{ptr} + \texttt{size})$ address range corresponding to the once-allocated memory mapping.

5. Once this is done, the pair $(\texttt{ptr}, \texttt{size})$ is removed from the metadata.

Behavior of the allocator when the deallocation process stops is by default to emit a failure. Described cases correspond to memory safety issues such as invalid or double frees: in a security-oriented setting, this justifies terminating the execution of the program.

Metadata usage. Consistent metadata about valid allocations is necessary to actually perform deallocation. As we shall see in Section 3.5.3, semantics of system calls and their modeling can sometimes be quite subtle. In the deallocation case, using an incorrect length as part of the `munmap` call could lead to correctness issues: would it be too small, only some of the pages backing the considered allocation's would be unmapped; would it be too large, possibly unrelated pages could be unmapped. This is the reason for the deallocation step retrieving the size associated with a pointer (already checked as valid).

In addition to that, as outlined in Section 2.1.2.2, keeping valid allocations metadata consistent is mandatory to implement `realloc` correctly. This further justifies why the storing of metadata is required.

As a final point regarding metadata's practical use, it should be noted that in this setting, we rely on the OS to reuse available space. As a consequence of this, not keeping metadata about the memory layout about free space, e.g., previously allocated memory blocks that have since been freed, does not hinder efficient memory reuse.

Thread safety. Provided APIs are made thread-safe through the use of a dedicated mutex guarding the use of the large allocator, without any further effort to reduce contention, in line with hardened_malloc's implementation. This may be surprising at first glance but is actually justified by the use of the Linux kernel to implement serialized changes to memory mappings used by user processes using a global lock (the "`mmap_lock`"). It is however noteworthy that there are some efforts to replace this lock by more efficient data structures, calling in turn for possible future improvements [147].

Metadata data structure. In the above, it should be noted that we left the associative data structure unspecified: any such data structure would be suitable. In practice, StarMalloc's implementation relies on a map implemented using an AVL tree. While hardened_malloc stores metadata using a hash table, using an AVL tree does not require reasoning on hash collisions. One reason arguing for the use of an AVL tree without worrying about associated costs is that system calls are overall very costly, likely surpassing by far any corresponding overhead.

Overall, this allocator could be described as a "toy allocator" when used as a standalone allocator, as it design is very quite simple and its performance quite poor. Indeed, on one hand, regarding time performance, it requires one system call per allocation and one system call per deallocation: this is quite costly, as previously mentioned. On the other hand, regarding memory performance, using it to provide small allocations would lead to important internal memory fragmentation. Indeed, each allocation is provided through one dedicated memory mapping, whose size is at least that of one page.

3.1.1.2 Large allocator: initialization

The initialization of the large allocator is quite straightforward, as only the associated lock and the AVL tree require to be initialized. However, as we will see, the large allocator relies on the slab allocator implementation for efficient allocations of the AVL tree's nodes, thus avoiding slow per-allocation syscalls. In turn, it requires the large allocator to initialize the dedicated sizeclass allocator, whose size is suitable for memory size of the AVL tree node.[5]

3.1.2 Regular allocations: a slab allocator

While we first described the allocator handling large allocations, most allocations are rather small and thus are the main target of verification efforts. As such, among the numerous set of possible constraints partially described in Section 2.1, they should be fast.

The allocator that we describe is a slab allocator that operates on a single, very large memory mapping of fixed sized that is allocated at initialization and used for all regular allocations. Its implementation relies on the large address space that virtual memory brings on modern hardware. As such, it would only make sense to use StarMalloc in suitable Unix-like modern environments.

3.1.2.1 Slab allocator's architecture

Slab allocator as a collection of suballocators. A slab allocator is a memory allocator that only provides a fixed set of allocations sizes by partitioning the described large memory mapping into size classes, presented in Section 2.1.1. In turn, the set of allocator-provided allocations sizes is the set of all allocator's size classes. As the original slab paper puts it [148], a "slab allocator is not a monolithic entity, but rather is a loose confederation of independent caches". Their setting is that of kernel memory allocation, where performance was and remains of utmost interest in many cases [149]. StarMalloc follows the design of hardened_malloc's slab allocator: in its setting and that of hardened_malloc, where the usual trade-off between security and performance leans rather towards security, cache is not a desirable feature and is thus not provided. The generic principle that the resulting allocator can be considered as a collection of independent suballocators is nonetheless retained.

[5]This is axiomatized and checked at compilation-time using C static assertions.

Supported size classes and memory performance. Let us use an example a slab allocator that only has 3 size classes: 64, 256 and 1024: it could only serve allocations up to 1024 bytes and would likely face large internal fragmentation. Indeed, 512-bytes allocations would in this case be provided by the 1024-bytes allocator. In order to avoid such issues, larger and thus finer-grained size classes sets are commonly used, so that resulting memory allocators have generally good performance. As we will see in Section 3.4, StarMalloc comes with a configurable set of size classes.

Now, let us proceed with a hierachical presentation of the slab allocator's architecture.

Arenas. The large memory region for regular allocations is first subdivided into *arenas*. As described in Section 2.1.3, the goal is to spread concurrent allocations from multiple threads onto distinct arenas, to avoid contention. The number of arenas is set at compile time; using thread-local storage, we associate an arena to each thread (e.g., `thread_id % n_arenas`, where `n_arenas` is the number of arenas). When a thread requests memory, the allocator looks up its associated arena then allocates there. Threads can however access and free memory located in a different arena, the main point of arenas is to spread out allocations in order to statically reduce contention. Arenas are *not* protected by locks, which operate at a smaller level of granularity.

Size classes. Each arena is further subdivided into size classes; a given suballocator operates on a single submapping of the initial very large memory mapping, and only handles allocations and deallocations related to this submapping.

Thanks to virtual memory, only the portions of each size class that are in use (or that were in use and then freed) are backed by physical memory. Each size class is associated to metadata that needs to be updated whenever an allocation takes place within that size class: for that reason, size classes are each protected by an individual lock. Recall that the mapping of threads to arenas is not injective (the number of arenas is fixed): the lock thus prevents two threads from racing to allocate within the same size class.

Slabs. In order to manage this memory mapping, the size class allocator further divides it into *slabs*. In the general setting, a *slab* designates "one or more pages of virtually contiguous memory" [148]: their size is fixed inside a size class and can depend on the considered size class. In StarMalloc's setting, a slab always is exactly equal to one system page.[6] Each size class allocator tracks all of its slabs using *size class metadata*. To this end, each size class allocator maintains a number `md_count` such that only the first `md_count` slabs have been made available for allocation, leaving any further slab untouched. For each slab made available for allocation, the size class metadata keeps track of the status of the slab, depending on whether it can be used for allocation or not. A slab is said to be *full* if it no longer has any free

[6] In practice, an extension of StarMalloc provides extended size classes, with allocations spanning several pages. We omit this in the rest of this document.

space for allocation; it is said to be *empty* if all its space is available for allocation; otherwise, it is said to be *partial*. Distinguishing between empty and partial slabs aims at reducing memory fragmentation: partial slabs are prioritized for allocation. In addition to this, other slabs statuses related to security mechanisms are used, see Section 3.1.3. The tracking of all three kinds of slabs is achieved using a dedicated, complex data structure based on doubly-linked lists to efficiently insert and remove elements when implementing `malloc` and `free`, described in Section 3.3.

Slots. Slabs are further divided into *slots*, that is, chunks of memory whose size always is equal to that of the underlying size class. One slot corresponds to one allocation unit, that is, one chunk of memory that ought to be allocated. The previously mentioned status associated with each slab thus depends on the number of slots available in the considered slab for allocation. To keep track of whether a slot is available or not, each slab is equipped with *slab metadata* keeping track of the status of each slot.

To give a concrete example of size classes, within the 4KiB size class, each slab contains exactly one slot; within the 16B size class (the smallest supported one), each slab contains 256 slots.

Metadata implementation. At this point, metadata has not yet been precisely described. StarMalloc actually follows hardened_malloc's quite opiniated choice of using segregated metadata instead of metadata intertwined with allocations; doing so is considered both as one important design choice [150] and security feature [151]. Using segregated metadata implies that all of the metadata is stored in a region that is entirely disjoint from the allocation region, which corresponds to the aforementioned very large mapping for all regular allocations. This is applicable to both size class metadata (about slabs) and slab metadata (about slots).

Let us refine the presented architecture to account for metadata. In practice, there actually are three large mappings requested from the operating system at initialization: one for the allocation region, one for size class metadata one for slab metadata. These three mapping are divided between all independent size classes allocators. We focus again on the architecture of a single suballocator, whose size class is `sc` and number of slots per slab is `nb_slots sc` = 4096/`sc`. Figure 3.1 provides a high-level presentation of the architecture of StarMalloc, including its metadata.

Let us note by `slabs`, `sizeclass_md` and `slabs_md` respectively the three suballocator-specific submappings of these of the global mappings. For any suitable k, the correspondence between these is as follows.

- The type of `sizeclass_md` is `array status`, where `status` is an enumerated type whose set of values is the set of all slab statuses. `sizeclass_md`$[k]$ is the status of the k-th slab, whose address range in bytes is $[\texttt{slabs} + 4096 * k, \texttt{slabs} + 4096 * (k + 1))$.

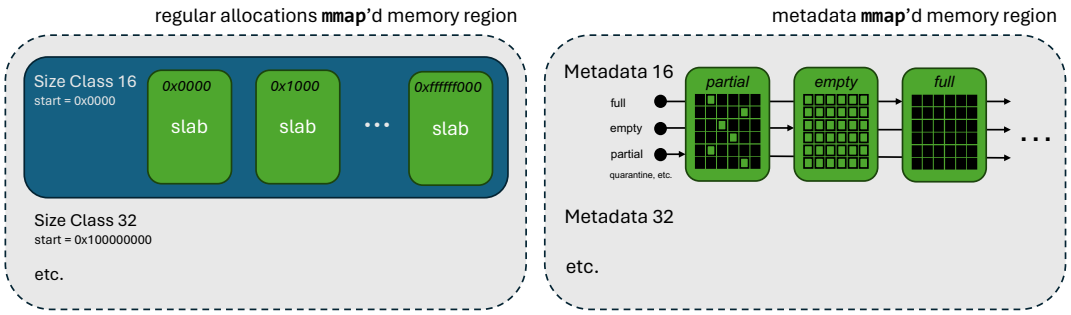


Figure 3.1: Architecture of StarMalloc. All regular allocations live in a single `mmap`'d memory region (left). All metadata lives in a single distinct `mmap`'d memory region (right). All allocations are contiguous. All metadata is contiguous. Accessing the n-th slab of the m-th size class thus boils down to pointer arithmetic, and similarly for the metadata. We rely on reserved, but uncommitted virtual memory to ensure only in-use pages occupy memory. Only a single arena is shown for conciseness; other arenas and their metadata are laid out contiguously in the same two `mmap`'d regions.

- `slabs_md`$[k]$ is the metadata of all slots that are part of the k-th slab. This metadata is a map from slot ids to boolean that is implemented as a 256-bit bitmap, represented by four 64-bit integers and thus `slabs_md` is an array of integers of type `array uint64_t`. The slab metadata corresponding to the k-th slab is thus stored in the range $[\texttt{slabs\_md} + 4 * k, \texttt{slabs\_md} + 4 * (k + 1))$.

One noteworthy difference with the large allocator is that for each allocation, its size is not explicitly stored but rather encoded as part of the memory layout invariants: one can retrieve the size of an allocation using its address.

3.1.2.2 Slab allocator: allocation and deallocation

Allocation process.

1. The client thread requests `size` bytes through the C standard library `malloc` function, that points to the StarMalloc's `malloc` implementation in our setting.
2. As `size` ≤ 4096, this allocation request is deemed to be a regular allocation request and is forwarded to the slab allocator.
3. Relying on thread-local storage, the allocator selects the arena associated with the thread requesting memory.
4. Given the size of the requested allocation, the statically defined set of size classes and the statically defined size class selection function, the request is forwarded to the corresponding size class allocator: the request is rounded up to the nearest available size class.
5. Corresponding suballocator's mutex is locked.

6. At this point, a non-full slab must be selected to proceed: the size class allocator relies on the following strategy. If there is any partial slab left in the partial slabs list, it selects the head of this list; otherwise, if there is any empty slab left in the empty slabs list, it selects the head of this list; otherwise, it increments the counter of used slabs `md_count`, and selects the first so far-unused slab. If no such slab exists, allocation stops, sizeclass mutex is unlocked and a null pointer is returned.

7. By construction, the selected slab has at least one slot that is free. One free slot is selected; slab metadata is updated; size class metadata also is updated; sizeclass mutex is unlocked; a pointer to the corresponding chunk of memory is returned.

Deallocation process.

1. The client program requests some memory allocation `ptr` to be deallocated through the C standard library `free` function, that points to the StarMalloc's `free` implementation in our setting.

2. Given its address `ptr`, StarMalloc checks whether this allocation is part of the very large mapping corresponding to the regular allocations region. If so, the deallocation request is forwarded to the slab allocator along with the pointer difference `diff` between `ptr` and the start of the regular allocations region.[7]

3. Given `diff` and that all size classes have the same share of the regular allocations region `sc_size`, computing `diff/sc_size` allows the allocator to retrieve the size class that `ptr` should correspond to. The deallocation request is forwarded to this size class allocator, along with `diff_sc` = `diff` mod `sc_size`.[8]

4. Corresponding suballocator's mutex is locked.

5. Given `diff_sc`, computing `pos_slab` = `diff_sc`/4096 allows the allocator to retrieve the slab that `ptr` should be part of. At this point, some checks are possible, such as the following ones. If `pos_slab` > `md_count`, that is, if `ptr` should be part of a slab that has not yet been used by the allocator, something clearly is wrong. Additionally, if `md_region`[`pos_slab`] is equal to an unsuitable slab status (e.g., due to security mechanisms, or the empty slab status), it is also clear that something is wrong.

    - In both described cases, sizeclass mutex is unlocked and the deallocation process stops.

[7] This is not corresponding to a UB once it has been checked that `ptr` is indeed part of the very large memory allocation unit that is the regular allocations region; the `ptrdiff_t` C integer type is used appropriately here.

[8] `diff_sc` is equal to the pointer difference between the `ptr` and the start of the size class share of the regular allocations region.

- Otherwise, the value `diff_slab` = `diff_sc` mod 4096 is computed and the process continues along with this value. [9]

6. Given `diff_slab`, computing `diff_slot` = `diff_slab` mod `sc` is necessary. Indeed, if `diff_slot` > 0, then `ptr` is misaligned with respect to valid allocations and thus cannot correspond to a valid allocation.

   - In that case, sizeclass mutex is unlocked and the deallocation process stops.
   - Otherwise, the value `pos_slot` = `diff_slab/sc` is computed and the process continues along with this value.

7. Given `pos_slot`, the allocator can determine to which slot `ptr` should correspond. At this point `md_bm_region[pos_slab][pos_slot]` is retrieved.

   - If the metadata corresponding to the alleged slot indicates that this slot is free, sizeclass mutex is unlocked and the deallocation process stops.
   - Otherwise, `ptr` indeed corresponds to a valid allocation and the process continues.

   Deallocation follows; slab metadata is updated; sizeclass metadata (including lists) is updated; sizeclass mutex is unlocked.

As previously mentioned, the default behavior of the allocator when the deallocation process stops is to terminate the execution of the program.

3.1.2.3 Slab allocator's initialization

As size class suballocators all are independent from each other, initializing the slab allocator can be done by initializing all of them sequentially. Initializing a suballocator is done the following way: memory obtained from the three large initial memory mappings is guaranteed to be zero-initialized (see Section 3.1.1), so that most well-formedness invariants for considered data structures hold for free. `md_count` is set to zero and doubly-linked lists storing sizeclass metadata initially are empty.

Finally, StarMalloc provides an additional feature over hardened_malloc as it offers configurable sizeclasses. This feature and its consequences on initialization implementation are further discussed in section Section 3.4.

3.1.3 Security mechanisms

In addition to the presented architecture, our design incorporates extra security mechanisms in order to defend against heap corruption vulnerabilities. This highlights the need for verification: the design above is non-trivial with many subtle invariants, and becomes even more so once the security mechanisms are added.

[9] `diff_slab` is equal to the pointer difference between the `ptr` and the start of the slab that `ptr` should be part of.

In Section 2.1.4, we briefly presented a variety of security mechanisms that can be used to mitigate some classes of memory safety issues. In this section, we present how security mechanisms blend with the bare allocator architecture we just described.

Segregated metadata. This security feature is intrinsically part of the architecture that was just presented: all metadata is placed in (`mmap`'d) memory regions that are entirely disjoint from the memory regions used for allocation. Buffer overflows affecting the allocation region should result in limited damage since the allocator's internal data structures would remain untouched.

Zeroing. Also implemented by Apple's OSX allocator [152], StarMalloc implements zeroing-on-free and checks that memory blocks to be allocated are still zero-initialized at allocation.

In the large allocator case, memory reuse is handled by the OS. Allocated memory through the use of the `mmap` system call is guaranteed to be initialized with zeroes. Memory is deallocated through the use of `munmap` and thus becomes immediately inaccessible; it corresponds to physical pages that will be zeroed as part of the OS memory management, so that future allocations may be provided zero-initialized.

Regarding the slab allocator, zeroing can be implemented transparently regarding its architecture, without relying on architecture-specific details.

All of the following security mechanisms are implemented as part of the slab allocator.

Guard pages. Guard pages are inserted each time `md_count` is incremented and the size class allocator is provided with more pages and thus available memory. Pages that can be used as allocations and guard pages are intertwined in a deterministic way, using a configurable pattern. One guard page is basically inserted every n pages. Guard pages have a dedicated additional status as part of the sizeclass allocator bookkeeping regarding slabs.

Heap canaries. Heap canaries are implemented transparently without affecting the rest of the slab allocator's architecture: all is required is to adjust the requested size and increasing it to account for the amount of memory required by canaries. Canaries are set at allocation and checked at deallocation: if values do not match, a buffer overflow has occurred and deallocation stops.

Quarantine. Quarantine is implemented at the hierarchical level of slabs handling, as it is quite coarse: no precise bookkeeping of slots use is done to implement it. Instead, when a slab whose status was full (in the case of the 4096-bytes size class) or partial becomes empty, it is put in quarantine. Quarantined pages have a dedicated additional status as part of the sizeclass allocator bookkeeping regarding slabs: they form one additional doubly-linked list that is actually exposed as a

queue. Slabs are selected from the quarantined only when no partial or empty slab is available, refining the allocation strategy outlined in previous section.

3.2 Developing a bare size class allocator: hierarchical verification

The architecture presented in the previous section divides functionality in several hierarchical levels. Beyond architecture details and as part of our verification methodology, our verified implementation leverages this, relying on F* support for interfaces that allows one to abstract away details from other components when verifying one specific file.

Let us recall that there are various hierarchical levels: 1) slots, chunks of memory part of 2) slabs, larger chunks of memory belonging to the allocation region of 3) size classes. Considering that a slab allocator can be approximated to a set of size classes, that is, a set of independent suballocators, we do not focus here on further hierarchical levels: 4) arenas, 5) the slab allocator as a whole and of course 6) the entire StarMalloc allocator.

In this section, we focus on the slab allocator and showcase our verification methodology by demonstrating how we can build separation logic predicates relating various data structures at a given hierarchical level as well as predicates relating different hierarchical levels. This relies heavily on Steel support for user-defined combinators: we use higher-order, custom combinators developed as part of StarMalloc's libraries. First, we consider in Section 3.2.1 the predicate defining a slot then move onto building the predicate describing a slab made of slots, whose ownership depend on slab metadata. We then proceed in Section 3.2.2 with reusing the same approach and generic predicates to construct predicates describing most of a sizeclass allocator, including sizeclass metadata consistency with respect to slabs it must keep track of.

3.2.1 Verifying the "ground floor": building slabs from slots

Starting at the bottom of the hierarchy, we first describe the hierarchical level corresponding to slots and several associated challenges. On one hand, following one of the main high-level design principles, slots and their metadata belong to disjoint memory regions: we thus must ensure that metadata are consistent with the actual allocator's internal state. On the other hand, slots is the hierarchical level that must deal with fine-grained ownership transfers, corresponding to allocation units provided to the client program at allocation and released by the client program at deallocation: a slot may be owned by the allocator or by the client program.

3.2.1.1 Modeling slot ownership

In what follows, we consider a size class `sc` and denote the number of slots per slab `nb_slots sc`.

Slots are chunks of memory of length `sc`: the i-th slot of a given slab corresponds to the address range in bytes $[\texttt{sc} * i, \texttt{sc} * (i + 1))$, where $i <$ `nb_slots sc`. The corresponding array-slicing function has the following signature[10].

```
val slot_array (sc: size_class) (slab: array U8.t) (pos: U32.t)
  : Pure array U8.t
  (requires U32.v pos < U32.v (nb_slots sc) /\
    slab_length sc <= length slab)
  (ensures fun r -> length r == U32.v sc)
```

Recall that an array does not correspond to a separation logic predicate in itself. This is the role of the following simple wrapper, equipped with a twin lemma, stating that the corresponding memory content (actually, the selector, see Section 2.2.3) is of appropriate type.

```
let slot_vprop' (sc: size_class)
  (slab: array U8.t{length slab >= slab_length sc})
  (pos: U32.t{U32.v pos < U32.v (nb_slots sc)})
  : vp:vprop
  = varray (slot_array sc slab pos)

val slot_vprop_lemma' (sc: size_class)
  (slab: array U8.t{length slab >= slab_length sc})
  (pos: U32.t{U32.v pos < U32.v (nb_slots sc)})
  : Lemma(t_of (slot_vprop' sc slab pos) == Seq.lseq U8.t (U32.v sc))
```

As previously mentioned, a slot may either be in use and thus not part of the allocator's ownership or free and thus part of the allocator ownership. This refinement over the `slot_vprop'` separation logic predicate is expressed thanks to the following custom `vprop` combinator we define. Depending on a supplied boolean value `b`, it wraps a `vprop vp` so that its selector is derived as an option over `vp`'s selector: if `b` is false, then the selector is `None`, otherwise it is `Some` v, where v is `vp`'s selector value.

```
let vp_to_opt (#a: Type) (b: bool) (vp: vprop{t_of vp == a})
  : vp':vprop{t_of vp' == option a}
  =
  if b
  then emp `vrewrite` (fun v -> Some v)
  else vp `vrewrite` (fun v -> None #a)
```

[10] Some details have been omitted.

```
7
8  let slot_vprop (sc: size_class)
9    (slab: array U8.t{length slab >= slab_length sc})
10   (md_as_seq: Seq.lseq bool (U32.v (nb_slots sc)))
11   (pos: U32.t{U32.v pos < U32.v (nb_slots sc)})
12   : vp:vprop{t_of vp == option (Seq.lseq U8.t (U32.v sc))}
13   = vp_to_opt
14       #(Seq.lseq U8.t (U32.v sc))
15       (Seq.index md_as_seq pos)
16       (slot_vprop' sc slab pos)
```

Once again, this comes with a twin lemma `slot_vprop_lemma` about selector type, here expressed as a refinement type over `slot_vprop`'s result for the sake of brevity. We will see in next paragraph why this is required.

At this point, expressing ownership about one slot depending on underlying metadata is possible. Given one slab, tying the sequence of its slots and the metadata, itself a sequence of booleans, remains to be done.

3.2.1.2 The generic `starseq` combinator

This pattern, corresponding to an iterated conjunction, is generic and occurs in several contexts, such as those described in Section 3.2.1.3 and in Section 3.2.2. We therefore capture this through a custom helper combinator called `starseq` that we define, whose signature is the following one.

```
1 val starseq (#a #b: Type)
2   (f: a -> vprop)
3   (f_lemma: (x: a) -> Lemma (t_of (f x) == b))
4   (s: Seq.seq a)
5   : vprop
```

The corresponding separation logic predicate is the following, using the notation $\mathtt{s}[i] := \mathtt{Seq.index\ s}\ i$:

$$\star_{i=0}^{\mathtt{length\ s}-1}\ \mathtt{f\ s}[i] = \mathtt{f\ s}[0] \star \mathtt{f\ s}[1] \star \ldots \star \mathtt{f\ s}[\mathtt{length\ s} - 1],$$

the associated selector being typed as `Seq.lseq b (Seq.length s)` through the use of `f_lemma`.
Leveraging dependent types and separation logic predicates as first-class values, this combinator is fully generic and works for sequences of any type `a` with any separation logic predicate `f`.

In practice, when used within StarMalloc's codebase, as will see, `a` is equal to a type of unsigned sized numbers (e.g., 32-bit unsigned integers). `s` corresponds to an interval of such numbers: with `v` denoting the coercion function from `a` to `nat`, for any suitable k, we have in such cases that $\mathtt{v}(\mathtt{s}[k]) = \mathtt{v}(\mathtt{s}[0]) + k$. As this combinator

is pervasively used in StarMalloc's codebase, it has been carefully designed: key design choices are the following.

- Using sequences instead of lists allows for easier manipulation of slices.
- Using a generic sequence allows for use with various machine integers types: StarMalloc's implementation relies on it.
- Although this comes with some indirection, this makes splitting a `starseq` predicate easier, as there is no need for an offset. Indeed, we prove as a lemma that the two following terms are equivalent.

```
1  starseq f f_lemma (s1 `Seq.append` s2)}
2  ≡
3  starseq f f_lemma s1 ⋆ starseq f f_lemma s2.
```

All of this reveals quite handful when defining helpers to rewrite separation logic predicates, such as the following admittedly verbose one, justifying why `f_lemma`s related to selector type are carried over.

```
1  val starseq_unpack (#opened:_) (#a #b: Type0)
2    (f: a -> vprop)
3    (f_lemma: (x:a) -> Lemma (t_of (f x) == b))
4    (s: Seq.seq a)
5    (n: nat{n < Seq.length s})
6    : SteelGhost unit opened
7    (starseq #a #b f f_lemma s)
8    (fun _ ->
9      f (Seq.index s n) ⋆
10     starseq #a #b f f_lemma (Seq.slice s 0 n) ⋆
11     starseq #a #b f f_lemma (Seq.slice s (n+1) (length s))
12   )
13   (requires fun _ -> True)
14   (ensures fun h0 _ h1 ->
15     // f (Seq.index s n)'s selector is of type b
16     f_lemma (Seq.index s n);
17     let v0 = h0 (starseq f f_lemma s) in
18     let v10 = h1 (starseq f f_lemma (Seq.slice s 0 n)) in
19     let v11 = h1 (starseq f f_lemma (Seq.slice (n+1) (length s))) in
20     length v == length s /\
21     Seq.slice v 0 n == v0 /\
22     Seq.index v n == v10 /\
23     Seq.slice v (n+1) (Seq.length s) == v11
24   )
```

In the setting of this unpacking stateful lemma, `f_lemma` is used to establish proper typing of the n-th element, which is required to establish equality over selectors.

The `starseq` definition is left opaque through the use of an interface file for the corresponding library. This way, the definition of `starseq` is carefully crafted to behave properly with regards to the unifier and normalizer of F*; notably, this recursive definition does *not* unfold during type-checking. This means we avoid a profusion of predicates in the context, which would cause performance and unification issues. Instead, we provided manipulation of this combinator through specialized lemmas such as the one we just presented.

Iterated conjunctions are not specific to Steel: other tools such as Viper [153] or CN [154] also support them in conjunction with SMT verification. Our approach differs however in two key ways. First, the `starseq` predicate is not natively encoded in Steel, rather, it is defined as a verified library on top of the Steel program logic, therefore ensuring that it is not a trusted component of the framework. Second, the predicates it iterates over are arbitrary separation logic predicates, which are more expressive than Viper's permission logic and do not exhibit the shape limitations required by CN's symbolic execution [154, p14]. In particular, the predicates inside the iterated conjunction can themselves be iterated conjunctions: `starseq` is just another separation logic resource, of type `vprop`, which can be passed to functions and predicates in Steel. As we will see in the rest of this chapter, we heavily rely on this genericity to nest predicates across abstraction layers.

3.2.1.3 Building slabs from slots

As presented in Section 3.1.2, slab metadata is a bitmap that we here abstract as a sequence of booleans of suitable length, that is, the content of the bitmap is of type `Seq.lseq bool (nb_slots sc)`, where `Seq.lseq` a n is the type of sequences of type a whose lengths are equal to n.

Equipped with all of these ingredients, we may now define the separation logic predicate corresponding to one slab in two more steps. First, given one slab and its metadata as a sequence of booleans, the following instantiates the just-presented generic `starseq` combinator.

```
let slab_vprop_aux (sc: size_class)
  (slab: array U8.t{length slab >= slab_length sc})
  (md_as_seq: Seq.lseq bool (U32.v (nb_slots sc)))
  : vprop
  = let incr_seq = seq_u32_init (nb_slots sc) in
  starseq
    #(pos:U32.t{U32.v pos < U32.v (nb_slots sc)})
    #(option (Seq.lseq U8.t (U32.v size_class)))
    (fun pos -> slot_vprop sc slab md_as_seq pos)
    (fun pos -> slot_vprop_lemma sc slab md_as_seq pos)
    incr_seq
```

In this definition, `slab_length sc` is equal to `sc * nb_slots sc`, which is the usable part of the slab as we will see. `seq_u32_init` is used to provide a sequence support for our of starseq, composed of the sequence of `U32` integers from 0 to `nb_slots sc`.

At this point, we need to use the real representation of slab metadata: the bitmap is actually stored as an array of four 64-bit integers. To this end, we rely on a dedicated library providing bitmap support that we developed as part of StarMalloc. Then, we can use the builtin `vdep` combinator to express the relationship of dependence between the internal state of the considered slab and its actual metadata. More precisely, the `vdep` allows us to express the fact that the actual value of the slab metadata is consistent with the actual ownership of slots using the `slab_vprop_aux` function relying on `starseq`.

```
1  val view_as_bitmap (sc: size_class)
2    (md_as_seq: Seq.lseq U64.t bitmap_size)
3    : Seq.lseq bool (nb_slots sc)
4
5  let slab_vprop (sc: size_class)
6    (slab: array U8.t{length slab >= slab_length sc})
7    (slab_md: array U64.t{length slab_md = bitmap_size})
8    : vprop
9    =
10   vdep
11     (varray slab_md)
12     (fun (md_as_seq: Seq.lseq U64.t bitmap_size) -> slab_vprop_aux sc
13       (split_l slab (slab_length sc))
14       (bitmap_to_bits sc md_as_seq))
15   *
16   varray (split_r slab (slab_length sc))
```

Finally, we observe that this definition has one more tweak. Indeed, only powers of two divide 4096: to support a larger set of sizeclasses, the array is split in two parts, using `slab_length` as the bound between subarrays. First part is composed out of slots (lines 10 to 14); second part is of length 4096 mod `sc` and unused (line 16). The `slab_vprop` is thus accordingly constructed as a separating conjunction on top of these two disjoints parts of the slab. In practice, the corresponding StarMalloc code is slightly more complicated due to bitmap-specific invariants.[11]

[11] Only the size class 16 uses the 256 bits of the size class. In all other cases, all of the unused bits are kept equal to zero so that it can be easily checked whether a slab is empty or not, see next subsection. A custom combinator called `vrefinedep` mimicking both `vdep` and `vrefine` is used accordingly instead of the `vdep` combinator to express this property on top of the code presented above.

3.2.2 Linking levels together: reusing generic predicates

In the previous subsection, we demonstrated how higher-order combinators can be used to follow the hierarchical organisation of allocator components in order to define corresponding separation logic predicates. In this subsection, leveraging F⋆ support for interface files, we lay the emphasis on the separation between different parts of the proofs, so that proof structure can be adapted with respect to the hierarchical organisation of allocator components.

As a first approximation, slabs are full or not, that is, for any given slab, either it is full or it contains some empty slots: as previously mentioned in Section 3.1.2, this information is stored as part of size class metadata. To ensure that this metadata is consistent with the slab metadata of each slab, low-level details regarding slots such as memory layout inside a slab or even slot allocation and deallocation functions can be abstracted over. Indeed, all is required is a way to distinguish for each slab between a full slab and a non-full slab using slab metadata. To this end, the previously defined `slab_vprop` predicate can be accordingly refined, using the `vrefine` combinator[12], as in the following example.

```
val is_full (sc: size_class) (md_as_seq: Seq.lseq U64.t 4): bool

let full_slab_vp (sc: size_class)
  (slab: array U8.t{length slab >= slab_length sc})
  (slab_md: array U64.t{length slab_md = bitmap_size})
  : vprop
  =
  slab_vprop sc slab slab_md
  `vrefine`
  (fun ((|md_as_seq,_|),_) -> is_full sc md_as_seq)
```

Recall that `slab_vprop` is of the form: `vdep (varray slab_md) _ ⋆ _` and that `(|v1, v2|)` is the syntax for a dependent pair in F⋆, corresponding to the underlying selector type of a `vdep` combinator.

Abstractly, the size class metadata maps each slab to a value of type `status = | Full | Available`. Foreshadowing future extensions (see Section 3.3), we use an inductive type for `status` instead of a boolean. As presented in Section 3.1.2, each size class allocator is equipped with three large memory mappings whose size is statically fixed. In fact, their sizes depend on a number `md_max` that is used as an upper bound for `md_count` such that:

- `slabs`, the region of all slabs, is actually composed out of `md_max` slabs;
- `slabs_md`, the region of all slabs' bitmaps (i.e., slabs metadata), is actually composed out of `md_max` bitmaps;

[12] Actually slightly modified.

- `sizeclass_md`, storing metadata about all slabs (ie sizeclass metadata), is an array storing statuses of length `md_max`.

The two following functions are helpers to carve out from these large regions the i-th slab and the i-th slab metadata.

```
val slab_array (sc: size_class)
  (slabs: array U8.t{length slabs = US.v md_max * U32.v page_size})
  (i: US.t{US.v i < US.v md_max})
  : arr:array U8.t{length arr = page_size}

val slab_md_array (sc: size_class)
  (slabs_md: array U64.t{length slabs_md = US.v md_max * bitmap_size})
  (i: US.t{US.v i < US.v md_max})
  : arr:array U8.t{length arr = bitmap_size}
```

Abstractly, the handling of slabs' and slots' metadata share many similarities: in both cases, metadata maps each slab (resp. slot) to a status (resp. boolean), with a separation logic predicate capturing slot or slab ownership based on the metadata value. Leveraging the genericity of the `starseq` predicate, we can follow the same proof structure, but with a different instantiation of `starseq`.

First, instead of the bitmaps used for slots, we rely on the just-defined statuses. This is due to the fact that slab metadata requires more information than slot metadata: slabs can be in five different states (full, partial, empty, quarantine, guard page), requiring the use of an enumeration, while slots are either available or in use, which can be captured by a simple bit of information.

The second step is to replace the `slot_vprop` predicate by `dispatch` below. `dispatch` captures the correspondence between the three sizeclass memory mapping. Indeed, our sizeclass metadata is conceptually a map from slabs to statuses. In our specifications, however, we use a different formulation that avoids tedious reasoning on the domain of the map. Since we statically know the number of slabs, and that each slab has associated metadata, we can represent metadata as a sequence of statuses of length equal to the number of slots. More precisely, for any suitable i, `md_region[`i`]` specifies the status of the i-th slab part of `slabs`, itself based on this very slab's metadata, stored in the i-th bitmap part of `slabs_md`.

```
val t (sc: size_class): Type

let dispatch (sc: size_class)
  (slabs: array U8.t{length slabs = US.v md_max * U32.v page_size})
  (slabs_md: array U64.t{length slabs_md = US.v md_max * bitmap_size})
  (sizeclass_md_as_seq: Seq.lseq status (US.v md_count))
  (md_count: US.t{US.v md_count <= US.v md_max})
  (i: US.t{US.v i < US.v md_count})
```

```
9   : vp:vprop{t_of vp == t}
10  =
11  let status = Seq.index sizeclass_md_as_seq (US.v i) in
12  let ith_slab = slab_array slabs i in
13  let ith_slab_md = slab_md_array slabs_md i in
14  match status with
15  | Full -> full_slab_vprop sc ith_slab ith_slab_md
16  | Available -> available_slab_vprop sc ith_slab ith_slab_md
```

It should be noted that it is restricted to slabs whose position is strictly less than the `md_count` integer used as a boundary between slabs in use and untouched slabs (part of per-size class metadata). In addition, foreshadowing another use of `starseq`, the underlying selector of the resulting `vprop` always is of type `t`; we abbreviated the corresponding lemma `dispatch_lemma` as a refinement type over the result line 9.

Reusing the `starseq` combinator in this new setting is then straightforward.

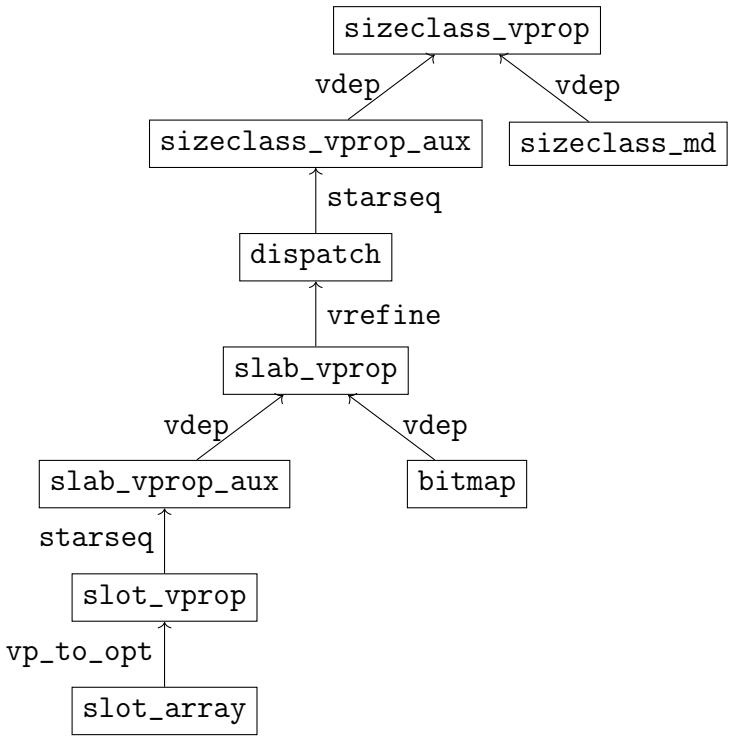
```
let sizeclass_vprop_aux (sc: size_class)
  (slabs: array U8.t{length slabs = US.v md_max * U32.v page_size})
  (slabs_md: array U64.t{length slabs_md = US.v md_max * bitmap_size})
  (md_count: US.t{US.v md_count <= US.v md_max})
  (sizeclass_md_as_seq: Seq.lseq status (US.v md_count))
  : vprop
  = let incr_seq = seq_us_init md_count in
  starseq
    #(pos: US.t{US.v pos < US.v md_count})
    #(t sc)
    (fun i -> dispatch slabs slabs_md size_class_md_as_seq i)
    (fun i -> dispatch_lemma slabs slabs_md sizeclass_md_as_seq i)
    incr_seq
```

`seq_us_init` is used to provide a sequence support for our of starseq, composed of the sequence of `size_t` integers from 0 to `md_count`.

The remaining part is to model storage of sizeclass metadata in memory, and to link it against the `sizeclass_vprop_aux` predicate. In our case, we will store the sizeclass metadata as an array of `status` values, whose contents are represented as a sequence. To link the two, we will therefore rely on the Steel `vdep` predicate previously presented. We emphasize that adopting a different memory representation for metadata would be straightforward: the only requirement is to provide a `vprop` whose contents can be interpreted as a `seq status`. This first refinement from implementation to "concrete" specification provides a first layer of abstraction which hides implementation details and serves as a foundation for subsequent proof layers.

```
let sizeclass_vprop (sc: size_class)
```

Figure 3.2: Final SL predicate as a tree where leafs are SL predicate refined and combined using higher-order combinators (edges).




```
2   (slabs: array U8.t{length slabs = US.v md_max * U32.v page_size})
3   (slabs_md: array U64.t{length slabs_md = US.v md_max * bitmap_size})
4   (sizeclass_md: array status (length sizeclass_md = US.v md_max))
5   (md_count: US.t{US.v md_count <= US.v md_max})
6   : vprop
7   =
8   vdep
9     (varray (split_l sizeclass_md md_count))
10    (fun v -> sizeclass_vprop_aux slabs slabs_md md_count v)
```

In the next section, we will see how to refine this to improve performance and support hardening.

It should be noted that the resulting final SL predicate can be seen as a tree where most non-leaf nodes are higher-order generic combinators, such as presented in Figure 3.2. `vdep` and `starseq` are used several times in various settings. Although this combination of predicates is nontrivial, e.g., with nested `starseq` combinators, thanks to the use of careful abstraction and combinators introduction/elimination pairs, this remains SMT-friendly and more generally verification-friendly.

3.3 Targeting performance and hardening: iterative verification

Beyond challenges inherent to the outlined architecture, we recall that our chief goal is to design a verification methodology that allows easily updating and extending the allocator implementation, while reducing the overhead of adapting the proofs. As a matter of fact, despite recent advances in proof repair [155, 156, 157], adapting such proofs when modifying code remains challenging. On the other end of the spectrum, SMT solvers can automatically discharge verification conditions, and can thus simplify proof repair when reverifying a modified program or specification.

In this section, we extend our efforts depicted in the previous section leveraging modular abstractions and Steel support for higher-order separation logic predicates to ease StarMalloc's verified implementation as part of our verification methodology. Notably, we fully leverage Steel's design, especially its separation of spatial proofs, which are tactic-driven, and pure functional predicates, which are SMT-driven. In Section 3.3.1, we show how to implement a complex data structure called an *arraylist* to improve performance regarding sizeclass metadata while preserving proofs outlined in previous section; in Section 3.3.2, we demonstrate how to efficiently iterate on further refinements of this data structure to implement security mechanisms.

3.3.1 The arraylist data structure: optimizing sizeclass metadata

The predicate `sizeclass_vprop` gives us a separation logic model of the slabs, their metadata, and their respective memory representations. It ensures the well-formedness of the slab architecture, namely, that metadata is coherent with respect to the parts of each slab that the allocator still owns, that is, that were not given to clients. While simple, this first model is also inefficient when attempting to pick a slab during allocation: a linear traversal of the metadata is required to find the first available slab.

To solve this issue, one natural solution is to use a linked list to keep track of the available slabs, thus skipping full slabs when traversing metadata. While lists have a well-known, textbook separation logic specification, one difficulty arises however when attempting to use them to implement a memory allocator. List-like structures typically rely on dynamic memory allocation when, e.g., creating and inserting a new cell, which is exactly what our allocator must implement! To circumvent this circularity, we leverage the fact that the maximum number of slabs is statically known, and instead implement a library for linked lists where all cells are part of one single static array: the sizeclass metadata. We call this data structure an *arraylist*.

A standard linked list implementation consists of cells with data and a pointer to the next element; the corresponding ownership predicate recursively covers all of the cells in the list. This, however, cannot apply here, as cells are already owned

by the outer array. A natural solution is to rely on a non-separating conjunction to combine ownership of the outer array with a shape predicate that captures the structure of the list. However, the automation facilities (notably the framing tactic) in the Steel framework do not support non-separating conjunction well.

To circumvent this issue, we rely on two components. First, as we know that all list elements are located in a given array, we replace pointers to the next element by array indices, of type `size_t`. Second, we separate the specification into two parts: on one hand, the ownership of the underlying array; on the other hand, the well-formedness of a list operating on a sequence view $\texttt{sizeclass\_md}_v$ of the array contents, expressed through `is_list`.

```
type cell (a:Type) = {data : a; next: size_t}

let rec is_list' (sizeclass_md_v: seq (cell a)) (pred: a -> prop)
  (idx: size_t) (count: nat)
  : prop =
  // Cyclic list
  if count > length sizeclass_md_v then False
  // Terminal case
  else if idx = null_idx then True
  // Out-of-bound
  else if idx < 0 || idx >= length sizeclass_md_v then False
  else
    pred sizeclass_md_v[idx].data /\
    is_list' sizeclass_md_v pred (sizeclass_md_v[idx].next) (count+1)

let is_list sizeclass_md_v pred idx
  = is_list' sizeclass_md_v pred idx 0
```

The predicate `is_list'` has two particular points of interest. First, it takes as parameter an abstract predicate `pred` operating on data, which must hold for all elements in the list. In our case, we instantiate it below with the property that all statuses in the list are `Available`. Second, this predicate does not capture ownership, and has therefore type `prop`, and not `vprop`.

Leveraging our combinator-based approach, we finally tie the list shape with ownership of the underlying array. To do so, we use the builtin `vrefine` generic Steel combinator: `vrefine slp p` captures the ownership of the `vprop` predicate `slp` while specifying that the predicate `p` holds on its selector.

```
let arraylist_sl (sizeclass_md: array (cell a)) (pred: a -> prop)
  (idx: size_t) : vprop
  = vrefine (varray ptr)
      (fun sizeclass_md_v -> is_list sizeclass_md_v pred idx)
```

As the predicate `is_list` also requires the start index of the list, we store it in an additional pointer `r`, and combine it with `arraylist_sl` using the previously seen `vdep` combinator.

```
let ind_arraylist_sl (sizeclass_md: array (cell a)) (pred: a -> prop)
  (r: ref size_t) : vprop
  = vdep (slptr r)
      (fun idx -> arraylist_sl ptr pred idx)
```

Equipped with this predicate, we can now propagate changes to `sizeclass_vprop` to replace the simple array of metadata by an arraylist. Importantly, the link between metadata contents and slabs, captured by the predicate `sizeclass_vprop_aux` remains unchanged. On one hand, the view of predicate `ind_arraylist_sl` still can be interpreted as a sequence of statuses: its selector type is a dependent pair: `dtuple2 size_t (size_t -> seq (cell status))`. An additional *ad hoc* combinator `dataify` is used to transform it into a `seq status`. On the other hand, the use of the `vdep` combinator separates between properties about the underlying data structure shape (the arraylist) and predicates depending on its contents.

```
let is_available (x: status) = x == Available

let sizeclass_vprop (r: ref size_t)
  (slabs: array U8.t{length slabs = US.v md_max * U32.v page_size})
  (slabs_md: array U64.t{length slabs_md = US.v md_max * bitmap_size})
  (sizeclass_md: array (cell status){length sizeclass_md = US.v md_max})
  : vprop
  = vdep
      (ind_arraylist_sl sizeclass_md is_available r)
      (fun (|idx, sizeclass_md_v|) ->
        sizeclass_vprop_aux slabs slabs_md (dataify sizeclass_md_v))
```

The example presented throughout this section is highly representative of our methodology. By relying on custom separation logic combinators, we achieve a high degree of modularity: case in point, the replacement of the initial metadata array by a more complex arraylist structure did not impact proofs relating to the metadata contents. The definition of these combinators relied on two key ingredients, namely, dependent types, and the ability to parameterize over separation logic predicates. Our dependently typed combinators also enable genericity, and reduce code duplication. In particular, by abstracting over the values stored in the array and predicates they must satisfy, we could define helpers on the generic arraylist once and for all (omitted here), e.g., to access metadata in constant time or to update the list, and instantiate them at no extra cost for our specific usecase; this data structure would be directly reusable in other contexts and with other predicates. Finally, our structure allowed us to preserve the Steel separation between shape and contents

reasoning, and to leverage Steel's facilities for SMT solving: predicates operating on memory contents, e.g., to reason about indices or on the metadata values, can be handled by SMT, lowering the cost of verification. In the next subsection, following our workflow during the development and verification of StarMalloc, we show how this methodology allows us to continue to efficiently iterate on the `arraylist` data structure, progressively obtaining the architecture outlined in Section 3.1.2.

3.3.2 Performance and hardening: arraylist iterations

The arraylists presented in the previous section allow us to efficiently find an available slab in which to allocate. This model however presents some limitations.

Tracking full slabs and partition invariant. First, while it guarantees that all slabs in the list are available for allocation, it does not ensure that all available slabs are present in the list, possibly leading to some unused memory, degrading the performance of the allocator.

To avoid this issue, we modify our invariants to also model that full slabs form a list, and that the two lists partition the metadata. Materializing both lists at runtime presents several advantages; notably, rather than relying on a global universally-quantified predicate, we can instead perform our proofs in a constructive manner by directly manipulating those lists, which leads to much better proof performance. This also makes inductive predicates over those lists much easier to write, a design decision that will prove useful as we later add even more structure onto our slabs.

Thanks to the structure of our separation logic predicates, these changes are well-contained. We only need to extend the `arraylist_sl` predicate, and to add an additional pointer $\texttt{r}_f$ to the head of the new list. In particular, proofs and code operating at the array level (e.g., to access metadata) or on the metadata sequence (to tie its values with ownership of the slabs) are entirely unchanged.

```
let arraylist_sl (sizeclass_md: array (cell a))
  (pred_a pred_f: a -> prop)
  (idx_a idx_f: size_t) : vprop
  = vrefine (varray size_class_md) (fun sizeclass_md_v ->
      is_list sizeclass_md_v pred_a idx_a /\
      is_list sizeclass_md_v pred_f idx_f /\
      partition sizeclass_md_v idx_a idx_f)

let ind_arraylist_sl (sizeclass_md: array (cell a))
  (pred_a pred_f: a -> prop)
  (r_a r_f: size_t) : vprop
  = vdep (slptr r_a `star` slptr r_f)
      (fun (idx_a, idx_f) -> arraylist_sl ptr pred_a pred_f idx_a idx_f)
```

Similarly, incremental changes allow us to add support for the additional lists for partial slabs, as well as quarantine and guard pages needed to harden our allocator.

Efficient deallocation: doubly-linked lists. A second limitation relates to our handling of another base operation: freeing memory. While allocation only appends and removes elements from the head of the list (e.g., moving the head of the available list to the full list when the slab becomes full) which can be done efficiently, freeing often requires removing an element in the middle of a list: the client can free allocated memory in any order, and is therefore not required to free an element in the slab that last became full. Removing an arbitrary element from a linked list induces a complexity linear in the list length in the worst case; this is at odds with the performance expected from modern allocators. To circumvent this issue, we instead replaced our implementation of singly linked lists with doubly linked lists, which allow removal in constant time. Changes were again small: we needed to extend the `cell` struct with an additional `prev` field, and to replace the `is_list` invariant by its `is_dlist` counterpart, capturing additional properties about the doubly linked list structure. Importantly, our reliance on the SMT solver to reason about the `is_list` predicate, enabled by the design of our combinators, greatly simplified updating proofs when switching to `is_dlist`: most of the inductive proofs, e.g., to show that the `is_(d)list` invariant is preserved when inserting or removing an element, are required by the combinators to have the same structure. This not only forces modularity, but also imposes the same shape on SMT queries, meaning the SMT solver was often able to automatically prove the new versions, or only required minimal changes.

Quarantine list is actually a queue. One last change focused on the handling of quarantined slabs, by implementing a queue data structure on top of the corresponding doubly linked list. Following our methodology, this change was localized to the `is_dlist` predicate corresponding to quarantine in the `arraylist_sl` predicate. In particular, in addition to not modifying proofs relating to the underlying array or the link between metadata and concrete slabs, this change also did not impact helpers and proofs on neighboring lists. Required modifications to interpret the doubly linked list as a queue were technical, but are a standard algorithmic construction; we omit them for brevity.
At the end of the day, we get an arraylist refined using the following predicate.

```
let varraylist_refine (#a:Type)
  (pred_e pred_p pred_f pred_g pred_q: a -> prop)
  (hd_e hd_p hd_f hd_g hd_q tl_q size_q:nat)
  (s:Seq.seq (cell a)) : prop
  =
  is_dlist pred_e hd_e s /\ // list of empty slabs
  is_dlist pred_p hd_p s /\ // list of partial slabs
  is_dlist pred_f hd_f s /\ // list of full slabs
  is_dlist pred_g hd_g s /\ // list of guard slabs
  is_queue pred_q hd_q tl_q s /\ // queue of quarantined slabs
  Set.cardinality (ptrs_in hd_q s) == size_q /\ // consistent queue size
  size_q <= Config.quarantine_queue_length /\ // bounded queue
  disjoint5 hd_e hd_p hd_f hd_g hd_q // lists are disjoint
```

The five ordered lists are: empty slabs, partial slabs, full slabs, guard slabs (that is, guard pages), quarantine slabs. They are disjoint, as specified using the `disjoint5` predicate. $\texttt{pred}_{e,p,f,g,q}$ correspond to the underlying lists predicates; $\texttt{hd}_{e,p,f,g,q}$ are the corresponding heads of lists; $\texttt{tl}_q$ is the tail of the quarantine list interpreted as a queue. The function `ptrs_in` constructs the set formed by a list starting at the considered head indice: $\texttt{size}_q$ corresponds to the size of the set formed by elements in the quarantine queue, that must not be higher than the (configurable) bound `quarantine_queue_length`. Finally, we note that it is only always required for this data structure to be well-formed that lists are disjoint. Requiring them to form a partition would prevent any extension of the data structure, which is an additional requirement to keep metadata consistent when more slabs are needed. Indeed, in that case, this results in more slabs to tracks, thus using more metadata cells.

3.4 Aiming for a configurable allocator: genericity and partial evaluation

The approach presented so far focused on designing modular, generic abstractions that reduced the verification effort, when initially developing verified components as well as when iterating over it.

In this section, we will see how the benefits of genericity can also appear in implementation efforts, namely, in making StarMalloc more performant and easily configurable. In Section 3.4.1, we present the reuse of the sizeclass allocator to improve performance of the large allocator. Then, in Section 3.4.2, we demonstrate the support for configurable size classes through partial evaluation and genericity. Furthermore, in Section 3.4.3, we show how to make this configurability competitive regarding performance, by supporting configurable sizeclass selection function, thus improving efficient slab memory allocations. Finally, in Section 3.4.4, we lay the emphasis on the fact that StarMalloc offers configurability through F*-safeguarded configuration files that are abstracted over in StarMalloc's implementation, thereby avoiding end users pitfalls while providing strong correctness guarantees.

3.4.1 Reusing the sizeclass allocator inside the large allocator

We also leverage our generic abstractions when handling large allocations, that is, allocations larger than a page size. As described in Section 3.1.1, these allocations are directly forwarded to the OS via the `mmap` syscall. The main role of the allocator is to keep track of the pointers currently in use, as well as their size; this is done through an AVL tree.

Similarly to the lists previously presented, tree-like structures have standard separation logic definitions; in our setting, the main implementation challenge is that they typically rely on dynamic allocation to, e.g., create new tree nodes when inserting a value, which is what the allocator aims to provide. To do this, our initial implementation was creating and freeing AVL nodes on-demand through the use of `mmap` and `munmap`. This was however largely inefficient, both in terms of speed and

memory usage: syscalls such as `mmap` are expensive, and the OS can only provide memory at the page granularity; much more than needed to store a small AVL node.

To avoid this issue, we instead reinstantiated our code for slab allocations to support dynamic allocation for AVL tree nodes. Concretely, we instantiated a slab allocator with only one size class, corresponding to the size of AVL nodes. To do so, we reused the `arraylist_sl` predicate and leveraged the genericity of our implementation in the sizeclass configuration (further described in Section 3.4.2). Adapting the AVL tree library we implemented to replace node allocation through `mmap` by our optimized allocation was then straightforward: node allocation and deallocation was already abstracting over implementation details, and performing the change did not impact the rest of the code and proofs.

3.4.2 Configurable sizeclasses, partial evaluation and genericity

General-purpose allocators such as StarMalloc can be used in many contexts: clients operating in a purely sequential setting might want to disable arenas (described in Section 3.1.2) to avoid memory waste or even locking (e.g., sizeclass locking in our setting) altogether, while clients that only perform very small allocations might want to disable the larger sizeclasses. When modifying the allocator configuration, some changes simply require modifying global variables (e.g., to adapt the size of a global array), but others are more involved, requiring deeper changes to the allocator's implementation (e.g., to initialize different numbers of size classes for different sizes, and hence with different slot layouts).

When implementing an unverified, configurable allocator in C, a common technique is to rely on preprocessor macros, which are expanded at compile time into specialized code depending on static, user-supplied parameters. This is however error-prone, and not ideal for security-critical code. Performing deep modifications into the code is also not acceptable; even in a verified setting where these changes could be safely performed, the effort involved would deter most clients.

In this section, we show how to write a generic Steel implementation, parametric in a number of arenas and size class architecture, while ultimately retrieving specialized, idiomatic C implementations after extraction. The key technical ingredient is to leverage custom compile-time reduction through F*'s normalizer (briefly presented in Section 2.2.2.3), a technique previously applied to other verification projects [158, 159, 160].

Our goal is to write an allocator generic in a size class architecture, described using a list of size classes `l` and a number of arenas `nb_arenas`. As sizeclass are independent suballocators with individual locks and all arenas come with the same set of sizeclasses, a list of sizeclasses with `nb_arenas` repetitions of the list `l` can be used, see Section 3.1.2. As a consequence, we wish to write the size class initialization function `init_size_classes` below, which schematically takes as arguments an array to store sizeclasses objects and sizeclasses mappings (slabs region, slabs metadata region, sizeclass metadata region) for each of these, as well as a list of sizes `l` representing the size classes configuration.

```
let rec init_size_classes size_classes sc_mappings l = match l with
| [] -> ()
| sc :: tl -> init_size_class sc size_classes[0] sc_mappings[0];
              init_size_classes tl size_classes[1..] sc_mappings[1..]

let init sc_list nb_arenas () =
  let l = repeat nb_arenas sc_list in
  let size_classes = mmap_sc (length l) in
  // basically three mmaps, one for each large mapping
  let sc_mappings = init_sc_mappings (length l) in
  init_size_classes size_classes sc_mappings l
```

In particular, all of the size classes spanning all arenas are initialized as part of this initialization. No arena data structure is initialized: it is only at runtime that threads will be assigned an arena upon their first memory allocation or deallocation request, using thread-local storage, which is beyond the scope of this section.

While this code is valid Steel and successfully verifies, it is not compatible with KaRaMeL extraction: pure lists do not extract to C (they would require a GC, since their ownership is not tracked). For presentation, we will assume that a client wishes to have two size classes: 16B, 32B, and two arenas; they will specify this by modifying the global variables `sc_list` and `nb_arenas`. We rely on the F* primitive `norm` to instruct the compiler to reduce the application `init sc_list`, according to the reduction steps `custom_steps` (omitted), which include, e.g., unfolding the definition `sc_list` and reducing pattern matchings operating on lists, but not unfolding the definition of `init_size_class`.

```
let sc_list = [16; 32]
let nb_arenas = 2
let init = norm custom_steps (init sc_list nb_arenas)
```

After normalization, all the recursive calls to `init_size_classes` have been unfolded, and all occurrences of lists are gone, which enables KaRaMeL to extract to the idiomatic C `init` shown below.

```
// Generated C code after normalization
void init() {
  size_classes* size_classes = mmap_sc(4U);
  uint8_t* slabs_region = mmap_u8(4096 * metadata_max * 4U);
  uint64_t* slabs_md_region = mmap_u8(4 * metadata_max * 4U);
  arraylist_cell* sizeclass_md_region = mmap_cell(metadata_max * 4U);
  // 2 arenas with 2 size classes inside of each = 4 size classes
  sc_mapping* sc_mappings = divide_mappings(slabs_region,
    slabs_md_region, sizeclass_md_region, 4U);
```

```
  // first arena
  init_size_class(16, size_classes[0U], sc_mappings[0U]);
  init_size_class(32, size_classes[1U], sc_mappings[1U]);
  // second arena
  init_size_class(16, size_classes[2U], sc_mappings[2U]);
  init_size_class(32, size_classes[3U], sc_mappings[3U]);
}

// Generated code as part of the rest of the initialization
const uint32_t sizes[4U] = {16U, 32U, 16U, 32U};
```

We comment in the last paragraph of this subsection as to why the `sizes` array of type `const uint32_t sizes[]` is useful.

This approach makes it straightforward to configure StarMalloc according to a client's needs. We only need to modify configurable variables `sc_list` and `nb_arenas` before recompiling our project; apart from this variable, all the code and proofs are left unchanged. We apply this methodology to all parts of StarMalloc where the static configuration impacts the shape of the code; this also includes selecting an appropriate size class in which to allocate (Section 3.4.3).

Implementation consequences. The initialization we described is quite complex, in the sense that it does not only rely on purely static initializers. Instead, global mutable state such as the `size_classes` array is modified at initialization before being used by all size classes of all arenas. This initialization is only done once with exclusive ownership on just-`mmap`'d memory that is not further modified during the entire execution of the allocator. In a naive verification setting, all subsequent memory accesses to these mutable size classes data structures would however require in itself using a mutex, very likely incurring performance overhead. To avoid this issue, we encode the fact that this array of size classes high-level state is never modified by using Steel's support for dynamically allocated invariants, briefly described in Section 2.2.3.3.
Let us describe schematically the way we proceed to this end. In a way similar to the creation of a lock, we create during initialization a Steel invariant associated to the array of size classes, losing plain access to it. The created invariant can then be used to provide read-only access to this array. As part of this optimization, we developed a dedicated "frozen array" library, including the following functions: `create_ro_array` is used at initialization, `index` during the entire execution of the allocator.

```
val create_ro_array (#a: Type) (r:array a)
  (s: Ghost.erased (Seq.lseq a (length r)))
  : Steel (ro_array r s)
  (varray r) (fun _ -> emp)
  (requires fun h -> asel r h == Ghost.reveal s)
  (ensures fun _ _ _ -> True)
```

```
7
8  val index (#r:array a) (#s:Ghost.erased (Seq.lseq a (length r)))
9    (ro: ro_array r s) (i: US.t{US.v i < length r})
10   : Steel a
11   emp (fun _ -> emp)
12   (requires fun _ -> True)
13   (ensures fun _ x _ -> x == Seq.index s (US.v i))
```

The compiler however is not necessarily aware of the fact that this array only is read. To further improve performance and code quality, we logically tie some of the content of the size classes array and a top-level immutable array together, leveraging Steel's support for it, briefly described in Section 2.2.4. This array only contains statically known size classes; implementation prioritizes accessing it over the size classes array when possible. The compiler can in turn determine that these reads are performed to a read-only array. Finally, this justifies the usefulness of the additional `const uint32_t sizes[]` array in the previous C code snippet.

3.4.3 Configurable sizeclass selection function

As previously mentioned, our support of different size classes configurations departs from hardened_malloc's implementation, which hardcodes one specific configuration. Our approach allows us to easily adapt StarMalloc to a wide range of contexts; however, naively implemented, it can also come at a performance cost. Configurations are abstract in the generic Steel code that implements, e.g., size class initialization; these functions cannot be optimized based on one concrete configuration.

One telling example is our function selecting a size class in which to allocate, based on the number of bytes requested. To be applicable to any list of size classes, our generic implementation successively compares the requested number of bytes n to the size of each size class, until it finds one large enough to accommodate n. After specialization, this results in a cascade of if-then-else branching, which can be expensive when branch prediction is poor. Leveraging its specific choice of configuration, hardened_malloc instead performs bitwise manipulations on n to find an appropriate size class in constant time. To address this inefficiency, we leverage well-known compile-time specialization techniques in F* [160], and allow the user to provide not only a choice of size classes, but also a corresponding, provably correct size class selection function. In the version of StarMalloc used in our evaluation (see Section 4.1), we pick both the size class configuration and the selection function used by hardened_malloc.

This can be summed up using the main steps of our iterative development. First, only a quite restricted set of size classes was supported, namely the set of powers of 2 between 16 and 4096, that is, 9 size classes. Lifting the sizeclass restriction, we adopted the set of 28 sizeclasses used by hardened_malloc[13] in order to be eventu-

[13]Regarding sizeclasses, one difference remains: the sizeclass 48 is currently unsupported by StarMalloc to simplify reasoning regarding bitmaps invariants.

ally able to proceed with a fair performance comparison. At this point, when doing performance experiments, we noticed some variability between old and recent workstations; testing with an unverified optimized selection function seemed successful and we thus proceeded with implementing a verified optimized selection function.

Finally, please note that while user-provided functions must be provably correct, the corresponding type used as specification does not capture any notion of optimum, e.g., the sizeclass selection function always selecting the largest sizeclass would be deemed correct. In addition to that, in line with hardened_malloc's implementation, no support for constant-time sizeclass selection function suitable for `aligned_alloc` has been implemented yet.

3.4.4 Configurable security mechanisms

This style is pervasive and also apply to most security mechanisms: for instance, quarantine, guard pages, and zeroing are all conditional on F⋆ compile-time switches, meaning that if the user opts out of those, F⋆ completely eliminates the corresponding `if false` expressions and those features are absent from the generated C code. We emphasize that this specialization at extraction-time does not extend our trusted computing base: F⋆'s normalizer is a key component of its typechecker, and hence already part of the Trusted Computing Base (TCB), further discussed in Section 3.5.5.

From a practical verification point of view, all actual configuration values are abstracted behind an interface file. This has two important consequences. First, proven theorems and their associated proofs hold for a generic set of configuration values, that is, for any set of configuration values. As a consequence of this, they fully depict the correctness of StarMalloc and are stronger than in the case where they would hold given fixed configuration values. We examine in more details theorems associated to StarMalloc's verification in Section 3.5.1. Second, given a state where all of the project has been verified and extracted, modifying configuration values and extracting again to update the produced C code is extremely fast. Indeed, given the interface file serving as an abstraction barrier, there is no need to reverify the whole project, which the verification toolchain automatically determines.

To be more precise, only code actually needing concrete configuration values, such as the list of size classes and the number of arenas for the aforementioned generic functions that are partially evaluated through normalization, needs to be verified again in such a setting. Only this part of StarMalloc's implementation explicitly breaks the configuration file interface through the use of the F⋆ builtin `friend` keyword.

Conclusion. All in all, StarMalloc uses a slightly different approach than that of hardened_malloc. As a matter of fact, while StarMalloc offers configuration switches in a F⋆ file safeguarded by the verification toolchain, hardened_malloc provides the user with configurable switches as part of user-editable dedicated configuration files (`config/default.mk` and `config/light.mk`, see [100]), whose static values (booleans or integers) are imported manually through `make` and then used in

the code as simple macros. The associated integer values are guarded by a set of C `static_assert` compile-time checks. Overall, this is is a careful version of the standard way to proceed using C macros, as previously mentioned in Section 3.4.2. In addition to that, StarMalloc provides additional configurability, with the possibility for a user to change the supported set of size classes (and the corresponding sizeclass selection function, see Section 3.4.3); excluding large size classes is the only way to configure the otherwise "hardcoded" sizeclasses set on hardened_malloc's side.

One could object that this comes at the cost of non-configurable C code on StarMalloc's side. In fact, this lower-level configurability on the C code more likely to be used by end users could be restored for simple configuration options (actually the configuration switches provided by hardened_malloc) through KaRaMeL's support for emitting macros.[14]

As a final word regarding this matter, this signals that configurability is not lost as a consequence from verification in our case, rather far from it. In line with other verification projects [158, 159, 160], we leveraged compile-time reduction through F⋆'s normalizer to obtain a configurable memory allocator.

3.5 Modeling as part of the verification process

In this section, we present our functional correctness theorems, corresponding to specifications of our verified memory management functions using Steel, and the underlying TCB, including required axiomatizations of low-level functions used as part of StarMalloc's implementation. This is all the more important since modeling both the desired correctness properties and the "outside" environment is necessary as part of the corresponding proof and forms one of its possible weakness, beyond the rest of the TCB as the usual list of software components.

We start by giving an overview of the basis of our functional correctness theorems in Section 3.5.1. In addition to that, we present how we specify the correctness of hardening features in Section 3.5.2. We then move onto the modeling of specialized versions of syscalls in Section 3.5.3, demonstrating that preciseness is both a matter of correctness and of performance in our setting. Furthermore, we give some examples of additionally required low-level axiomatized functions in Section 3.5.4, modeling being limited to what is strictly necessary. Finally, we examine StarMalloc's Trusted Computing Base (TCB) in Section 3.5.5.

3.5.1 Functional correctness theorems

Leveraging Steel support for CSL to specify and prove the correctness and safety of concurrent F⋆ programs, we target the functional correctness of standard APIs required to form a real-world allocator.

As an example, a minimum specification of memory allocation expressed in Steel

[14] E.g., using the `CMacro` attribute on F⋆'s values.

that forms the basis of our functional correctness theorem for `malloc` is the following.

```
val malloc (size: size_t)
  : Steel (array uint8_t)
  emp (λ r -> null_or_varray r)
  (requires λ h0 -> ⊤)
  (ensures λ h0 r h1 -> not (is_null r) ⟹ length r ≥ size)
```

That is, as mandated by the C standard, requesting an allocation be made through `malloc` of size `size`, the returned value either is a null pointer or a pointer pointing to a memory block of at least the specified length `size`. Other C requirements regarding allocation and deallocation were described in Section 2.1.1: alignment requirements and their modelization are described in Section 3.5.3.

Similarly, a minimum specification of `free` expressed in Steel forming the basis of our corresponding functional correctness theorem is the following.

```
val free (ptr: array uint8_t)
  : SteelT bool
  (null_or_varray ptr)
  (λ b -> if b then emp else slarray ptr)
```

That is, the returned value specifies whether deallocation was successful or not. In turn, this value can be used by the C wrapper (more on this below), e.g. to determine whether a failure should be emitted to account for the deallocation failure and thus aborts the entire process. By default, this is the case, thus enforcing some basic strictness discussed in Section 2.1.4.4. Finally, this explains what happens in case of deallocation failures that were described as "deallocation process stops" in Section 3.1.1 and in Section 3.1.2: a failure can be emitted by the C wrapper depending on the returned boolean.

The need for C wrappers is justified by the following. When written in the KaRaMeL-compatible fragment, Steel code extracts to idiomatic C code. For instance, the Steel `malloc` signature above will yield the following standard C function prototype: `uint8_t* malloc(size_t size)`. This signature is more precise than that of the C standard (`uint8_t*` instead of `void*`); in practice, we add a wrapper that performs a cast, so as to match the expected ABI and thus provide a drop-in replacement for existing userspace memory allocators. The same reasoning applies for `free` and other memory management functions. As we will see in Section 4.1.1, these wrappers, also dubbed "C glue", only represent a small amount of code.

Once basic functional correctness theorems are established, ensuring that they remain valid or even refining them, e.g., while adding security mechanisms, can be done in an iterative way. In turn, this results in additional modeling: this is the object of next sections.

3.5.2 Specifying hardening features' correctness

As described in Section 3.1.3, StarMalloc provides several hardening mechanisms, such as segregated metadata, canaries, or the use of quarantine and guard pages. Formally establishing security guarantees about hardening mechanisms is an open question and out of scope for this work, we instead only rely on hardened_malloc's design choices. We however wish to prove their correctness, namely, that their implementation matches their expected design, and preserves the allocator's functionalities.

We briefly show how zeroing and guard pages impact our specifications; following the methodology outlined in Section 3.3, required code changes are often localized and well-contained.

Zeroing. To specify that allocations must be zero-filled, we extend the specification of `malloc` presented in Section 3.5.1 as follows, adding an extra postcondition in the `ensures` clause. The postcondition ensures that the contents of the returned pointer `r` in the final state `h1` only consists of zeros.

```
val malloc (size: size_t)
  : Steel (array uint8_t)
  emp (λ r -> null_or_varray r)
  (requires λ _ -> True)
  (ensures λ _ r h1 -> not (is_null r) ⟹ (
    let r_v : seq uint8_t = h1 (null_or_varray) in
    length r ≥ size /\
    // length r = Seq.length r_v, thus Seq.length r_v ≥ size
    (∀ (i:nat{i < size}). r_v[i] == 0)
  ))
```

As mentioned in Section 3.1.3, our free operation takes care of zeroing, and our malloc operation (and others) takes care of checking that memory is still zero. If the memory is corrupted, the allocator emits a failure and interrupts the client program execution.

Guard pages. We previously showed (Section 3.3) how to extend the `arraylist` data structure to support additional slab lists, including one for keeping track of guard pages. All that is left is to model guard pages using separation logic: we want to design a `vprop guard_slab_vprop` capturing that a slab corresponds to a guard page; this will be the predicate associated to a `Guard` status by the previously presented `dispatch` function.

```
let dispatch (sc: sizeclass)
  ...
  (i: US.t{US.v i < US.v md_max})
  : vprop
  =
  let status = Seq.index sizeclass_md_as_seq (US.v i) in
  let ith_slab = ... in
  let ith_slab_md = ... in
  match status with
  ...
  | Guard -> guard_slab_vprop sc ith_slab ith_slab_md
  ...
```

Importantly, the `guard_slab` predicate is abstract, and we only provide an introduction rule, as shown below. This induces a form of monotonicity: once a slab is marked as `guard_slab`, by virtue of abstraction, it can never be eliminated to retrieve a separation logic predicate on which the allocator could operate, or that it could transform to an `varray` to pass to the client. Under the hood, the introduction rule is implemented as a call to `mmap` with the `PROT_NONE` flag; this ensures that any access to a guard page will result in a `SIGSEGV` returned by the OS.

```
val guard_slab_vp
  (slab: array U8.t{length slab >= slab_length sc})
  (slab_md: array U64.t{length slab_md = bitmap_size})
  : vprop

val guard_slab_intro slab slab_md
  : Steel unit
  (varray slab ⋆ varray slab_md)
  (fun _ -> guard_slab slab slab_md)
```

Additionally, guard pages must appear following a specified pattern, as already hinted at in Section 3.1.3; typically, every `Config.guard_pages_interval` pages, where `guard_pages_interval` is a statically fixed, user-defined constant. When increasing `md_count`, whose purpose is to distinguish between slabs currently in use and so-far untouched slabs, the sizeclass metadata's update is specified the following way.

```
val extend_set_of_usable_pages
  (sizeclass_md: array (cell status){length sizeclass_md = md_max})
  (md_count_v: size_t{md_count_v + guard_pages_interval ≤ md_max})
  (idxs: Seq.lseq size_t 7)
  : Steel (erased (Seq.lseq size_t 7))
```

```
  (varraylist preds
    (split_l sizeclass_md md_count_v) idxs ⋆
  ...)
  (fun idxs' -> varraylist preds
    (split_l sizeclass_md (md_count_v + guard_pages_interval)) idxs' ⋆
  ...)
  (requires fun h0 ->
    let ptr0 = split_l sizeclass_md md_count_v in
    // varraylist selector of type seq (cell status)
    let gs0 = v_arraylist preds ptr0 idxs h0 in
    // 5 lists designed by first 5 indices partition metadata
    partition #status gs0 idxs[0..4]
  )
  (ensures fun h0 idxs' h1 ->
    let ptr0 = split_l sizeclass_md md_count_v in
    let ptr1 = split_l sizeclass_md (md_count_v + guard_pages_interval) in
    // varraylist selectors of type seq (cell status)
    let gs0 = v_arraylist preds ptr0 idxs h0 in
    let gs1 = AL.v_arraylist preds ptr1 idxs' h1 in
    // 5 lists designed by first 5 indices partition metadata
    partition #status gs1 idxs'[0..4] /\
    dataify gs1 == Seq.append (dataify gs0) (Seq.append
      (Seq.create (guard_pages_interval - 1) Empty)
      (Seq.create 1 Guard)
    )
  )
```

That is, partitioning invariants are preserved while the correct number of empty pages as well as one guard page are added at the end of the set of currently used pages.

Quarantine. Quarantine is handled similarly: the `quarantine_slab` predicate is abstract, so that introducing it as well as eliminating it and thus getting back a normal array require corresponding OS memory management. Indeed, depending on some configurable value, slabs put into quarantine may have their permissions entirely removed. In this case, permissions of a slab are restored when it is released from the quarantine. In addition to that, quarantined slabs are released in a FIFO order, which is guaranteed by the varraylist refinement specifications about the quarantine slabs doubly-linked list refined as a queue.

Canaries. Our current heap canaries implementation is basic and uses magic constant values on two bytes. These values are checked at deallocation: the slab allocator's deallocation function specifies that for deallocation to be successful, these values must match at deallocation time their original values.

```
val slab_free (ptr:array U8.t)
  : Steel bool
  (varray ptr ⋆ [...])
  (fun b -> (if b then emp else varray ptr) ⋆ [...])
  (requires fun _ -> [...])
  (ensures fun h0 r _ ->
    let s = asel ptr h0 in
    enable_slab_canaries_free ==>
      (r ==>
        length ptr >= 2 /\
        Seq.index s (length ptr - 2) == slab_canaries_magic1 /\
        Seq.index s (length ptr - 1) == slab_canaries_magic2
      )
  )
```

3.5.3 Syscalls modeling: both for correctness and performance

Memory allocators provide a layer on top of operating systems: they manage memory obtained through syscalls such as `mmap` or `sbrk`, see Section 2.1.1.3. To implement and verify a `mmap`-based memory allocator such as StarMalloc, it is thus necessary to model `mmap`'s behavior. In this work, there are two different uses of `mmap` to request memory from the OS: at initialization for the slab allocator and for each large allocation request as part of the large allocator. We lay the emphasis on the fact that we only specify a small subset of all features supported by `mmap`, relying on specialized axiomatizations. These F⋆ specialized axiomatizations have C functions counterparts, whose underlying arguments for `mmap` were carefully selected.

3.5.3.1 Slab allocator: initialization spec and correctness

In the setting of initializing the slab allocator, we rely on the axiomatization shown below.

```
val mmap_init_u8 (size: size_t)
  : Steel (array uint8)
  emp (fun r -> varray r)
  (requires fun _ -> size > 0 /\ size  mod  page_size == 0)
  (ensures fun _ r h1 ->
    let r_v = asel r h1 in
    length r == size /\
    // length r = Seq.length r_v, thus Seq.length r_v = size
    (forall (i:nat{i < size}). r_v[i] == 0) /\
    aligned r page_size
  )
```

SL pre/postconditions. `mmap_init_u8` is used to reserve a very large amount of memory that will form the very large memory mapping used for slab allocations. As this function is only used at initialization, it is assumed that it always return a valid array `varray` of bytes: otherwise, initialization of the allocator is aborted.[15]

Pure preconditions. Per the `mmap` specification, `mmap` fails when the requested size is equal to zero: this size thus is required to be greater than zero. Furthermore, as the mapping is used as a large number of pages, the size of the requested mapping is specified to be a multiple of the page size.

Pure postconditions. Again, as this function is used at initialization, it is assumed it does not fail and always returns an array of size greater that the requested length. Additionally, this signature does not model resource exhaustion at the OS level: in practice, if `mmap` returns `NULL` at runtime because the OS could not provide further memory, we raise an uncatchable fatal error and interrupt the execution of the allocator. Our axiomatization also states that the returned memory is zero-initialized; to guarantee this, our implementation calls `mmap` with the `MAP_ANONYMOUS` flag.

Alignment. One last point of interest relates to our handling of alignment, captured by the predicate `aligned r page_size`. As seen in Section 2.1.2.3, the C standard dictates that values returned by `malloc` must be suitably aligned, in practice on 16-bytes on modern x86_64. In addition to that, `aligned_alloc` must provide overaligned allocations whose alignments are specified by the user.

To formally capture alignment constraints, we rely on the predicate `aligned ptr` n, which represents that `ptr` is aligned on a n-bytes boundary; the postcondition of `mmap_init_u8` captures our assumption that calls to `mmap` always return a page-aligned memory. Indeed, no address hint `addr` at which the mapping should be created is provided to the kernel: per `mmap` specification, "if `addr` is NULL, the kernel chooses the (page-aligned) address at which to create the mapping".

This predicate is propagated throughout our code, and is eventually provided to the client as a post-condition of `aligned_alloc` (`aligned ptr n`) or `malloc` (`aligned ptr 16`). Our `malloc` returns values that are always aligned on a 16-bytes boundary, a stricter version of what some older ABIs (e.g., Linux x86) permit. To facilitate reasoning within StarMalloc, we mostly keep the `aligned` predicate abstract. Non-linear arithmetic is circumscribed to one-controlled lemma that goes from `aligned ptr page_size` to smaller alignment constraints, thus limiting issues with SMT solvers that are notoriously unstable when non-linear arithmetic is involved [134, 161].

[15] The corresponding error is emitted by the C wrapper executing the cast from the `void*` returned by the actual `mmap` call to the desired return type `uint8_t`.

3.5.3.2 Large allocator: precise axiomatization and performance

In the previous case, precise axiomatization was a matter of correctness, e.g., to define a precise specification of the `aligned_alloc` API. In this case, it is also related to time performance, as we will see.

In the setting of large allocations, we rely on the `mmap_u8` axiomatization shown below.

```
1  val page_rounding (size: size_t)
2    : Pure size_t
3    (requires True)
4    (ensures fun r -> r ≥ size)
5
6  val mmap_u8 (size: size_t)
7    : Steel (array uint8)
8    emp (fun r -> null_or_varray r)
9    (requires fun _ -> size > 0)
10   (ensures fun _ r h1 -> not (is_null r) ⟹ (
11     let r_v : seq uint8_t = h1 (null_or_varray) in
12     (forall (i:nat{i < length r}). r_v[i] == 0) /\
13     aligned r page_size /\
14     // mmap returns pages, not arbitrary-sized arrays
15     length r == page_rounding size
16   ))
```

Most of this specification is equivalent to the previous one, `mmap_init_u8`. The part that we are interested in corresponds to the line 15.

As presented in Section 2.1.2.2, the `realloc` function is part of the C standard and thus must be supported: it belongs to the set of APIs provided by StarMalloc. This function most of the time resizes (that it, either expands or shrinks) existing allocations.

Given that StarMalloc is actually composed out of two main suballocators, in the latter, there are four possible cases to consider: on one hand, depending on which allocator initially allocated `ptr`; on the other hand, depending on which allocator is responsible for the new allocation (in-place or not).

In this setting, we are only interested in the case where an initially large allocation remains a large allocation. If `new_size` ≤ `old_size`, `old_size` being a value retrieved from metadata (in this case, in the dedicated AVL tree) it is possible to shrink in-place the existing allocation, and this mostly amounts to a no-op. Otherwise, in a naive setting, each reallocation entails a call to `mmap` (as well as a call to `munmap`).

During various benchmarks that we will present in Section 4.1.2, we stumbled onto a quite specific case: a sequence of increasing reallocations, the new size being at each iteration the old size increased by a few bytes. Comparing StarMalloc at the

time with hardened_malloc showed us that it was actually possible to do better. Indeed, `mmap` returns pages, not arbitrary-sized arrays. Thus, when proceeding with a large allocation, the actual mapping size is equal to a rounding up to the nearest `page_size` multiple of the requested size using `page_rounding`.

This led us to this specification refinement, in association with very few code changes, that is, storing the actual mapping length in place of the requested length. This way, the in-place expanding optimization when `new_size` ≤ `old_size` intervenes in all relevant cases.

In this admittedly quite specific but realistic case[16], refining our syscall modeling straightforwardly led to time performance improvements, avoiding a large number of syscalls in some settings.

3.5.4 Other low-level axiomatization

While underspecification is the common thread through this entire section, additional modeling is necessary as part of StarMalloc's implementation. Once again, we do not set as a goal to be exhaustive: we only intend to showcase a few relevant examples deemed representative of the other axiomatized functions.

Zeroing. Ensuring that allocations provided to the client program are empty, that is, zeroed, is necessary in two distinct cases. On one hand, `calloc`'s specification requires it; on the other hand, zeroing is part of StarMalloc's security mechanisms. Regarding large allocations, they are performed through the `mmap` syscall, that guarantees to return pages filled with zeroes when used in a suitable manner, as we have seen. In this case, no additional zeroing is required. On the opposite, small allocations require additional handling regarding zeroing. In that case, to perform zeroing, `memset` seems suitable at first glance. Unfortunately, modern C compilers can introduce "silent bugs" when aggressively optimizing code, as Xu et al. [162] put it. As a consequence, we rely on the quite common good practice to use a suitable function resistant to compilers optimizations and end up with the following unsurprising axiomatization. The C zeroing function actually used is the zeroing function shipped as part of HACL⋆[17], that presents the advantage of being already battle-tested.

```
1 assume val apply_zeroing_u8 (ptr: array uint8) (length: size_t)
2   : Steel unit
3   (varray ptr) (fun _ -> varray ptr)
4   (requires fun _ -> length <= length ptr)
5   (ensures fun _ _ h1 ->
6      let v = asel ptr h1 in
7      (forall (i:nat{i < length}). v[i] == 0)
```

[16] The `redis` benchmark as part of mimalloc-bench, see Section 4.1.2.

[17] Accessible at the following URL: `https://github.com/hacl-star/hacl-star/blob/main/lib/c/Lib_Memzero0.c`.

```
8   )
```

Overflow-checking multiplication. The safety and correctness of StarMalloc are captured through the concise Steel signatures of the API functions exposed to client programs, e.g., `malloc` or `free`. They often have trivial preconditions, when this is not the case, we attempt to apply defensive programming. For instance, the `calloc` function aims to allocate an array of `n` objects of a given `size`; its correctness requires that the multiplication `n * size` does not overflow. To avoid API misuses, we define a wrapper that checks this at runtime and emits a failure in case of overflow, thus exposing a clean API with trivial preconditions to the user. This can be seen as input sanitizing of some sort.

```
1 assume val builtin_mul_overflow (x y: size_t)
2   : Pure size_t
3   (requires True)
4   (ensures fun r ->
5     FStar.SizeT.fits (v x * v y) /\
6     v r == v x * v y
7   )
```

We here keep the coercion function `v: size_t -> nat` explicit, to avoid any possible ambiguity. The `FStar.SizeT.fits` function indeed takes a `nat` as sole argument. The corresponding C code makes use of the compiler builtin `__builtin_mul_overflow`, supported by both GCC and clang/LLVM.

Fast suitable slot position retrieval. We did not dive into the bitmap implementation used to handle slab metadata, as it relies on expected fast bitwise operations such as shifts to perform metadata retrieval and update when provided with the position of the considered bit.

However, finding a suitable position in itself may be challenging. Let us consider the allocation process. Given metadata associated to a slab guaranteed not to be full, all that is required to proceed with slot allocation is to find a suitable position within the bitmap. In all cases, iterating over the set of suitable positions would be too slow: hardened_malloc actually relies on a suitable compiler builtin; StarMalloc also followed this approach and relies on a compiler builtin.

Let us recall that the bitmap layout across the four 64-bit integers depends on the number of slots, itself depending on the sizeclass. The supported set of size classes is the following one[18]: $S_{\mathtt{sc}} = \{16, 32\} \cup S'$, where S' is the set of multiples of 16 between 64 and 4096. As a consequence of that, there are several cases:

- the 16 size class: 256 slots, the bitmap uses fully all four 64-bits integers;

[18] Any subset of the integer interval $[64, 4096]$ may actually be supported: the restriction corresponds to the previously mentioned alignment requirement to 16 bytes for all allocations, see Section 2.1.2.3

- the 32 size class: 128 slots, the bitmap uses fully the first two 64-bits integers;

- sizeclasses ≥ 64 have at most 64 slots: one 64-bit integer is enough to store metadata.

In this last case, only some part of the integer x is actually used to store metadata: if n of the number of slots in the considered sizeclass, only the n least significant bits of x are used. In our setting, given one slab, the i-th bit of the associated slab metadata stored as a bitmap is set if the i-th slot is currently in use. Therefore, what is needed is an efficient function finding the first unset bit, starting from the least significant bits.

To this end, the function `f` that we axiomatized is defined for any suitable input x as `f`(x) = `__builtin_ctzll`$(\sim x)$, where `ctzll` stands for count trailing zeroes of a long long integer and ~ is equal to the bitwise not operator, inverting all bits of x is the aforementioned setting. That is, this function `__builtin_ctzll` returns the number of consecutive zeroes starting from the least significant bits, which, if the argument is not equal to zero, to a valid position of the first set bit. As we are looking for the first unset bit, $\sim x$ is used as argument instead of x.

The corresponding axiomatization is the following one.

```
1  // starting from least significant bits
2  val nth_bit_u64 (x: uint64) (n:nat{i < 64}) : bool
3
4  assume val f (x: uint64) (bound: erased nat)
5    : Pure U32.t
6    (requires
7      x < U64.max_int /\
8      bound <= 64 /\
9      (exists (k:nat{k < bound}). nth_bit_u64 x k = false
10   (ensures fun r ->
11     // this implies that r < bound
12     (forall (k:nat{k < 64 /\ nth_bit_u64 x k = false}). r <= k) /\
13     nth_bit_u64 x r = false
14   )
```

As ~ x is used as argument for the builtin in place of x, and due to the fact that this builtin has undefined result when `x` = 0, line 7 ensures that this case does not happen. Next two lines actually refine this precondition, further guarding this function's use. As the postcondition states that given all unset bits, the smallest position corresponding to such an unset bit is returned, it can be automatically inferred through the SMT that there exists an unset bit in the `bound` first bits (starting from the least significant ones).

3.5.5 StarMalloc's Trusted Computing Base (TCB)

Similarly to other verification projects, our TCB includes our verification toolchain, which consists of the F* proof assistant [163] and the underlying Z3 SMT solver [164], as well as the KaRaMeL compiler [141] that extracts F* code to C. To obtain executable code, we thus also need to assume the correctness of a C compiler.

The safety and correctness of StarMalloc are captured through the concise Steel signatures of the API functions exposed to client programs, e.g., `malloc` or `free`. They often have trivial preconditions: when this is not the case, we attempt to apply defensive programming. For instance, the `calloc` function aims to allocate an array of `n` objects of a given `size`; its correctness requires that the multiplication `n * size` does not overflow. To avoid API misuses, we define a wrapper that checks this at runtime, thus exposing a clean API with trivial preconditions to the user.

The last part of our TCB is our axiomatization of the interaction of the OS, i.e., our model of `mmap` and `munmap`. While the corresponding Steel signatures are designed to be simple and easy to review, the underlying C implementation, calling the syscalls with specific arguments, needs to be carefully audited. Doing so, we found a mismatch with our assumptions at the Steel level (a missing flag in `mmap`), which we fixed.

Chapter 4

Benchmarking and Deploying Memory Allocators

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
>
> Brian Kernighan and P. J. Plauger,
> *The Elements of Programming Style*, 2nd ed. (1978), p. 10

This chapter is partly adapted from the "StarMalloc" article published at OOPSLA'24 [1].

In this chapter, we aim to evaluate StarMalloc through various benchmarks to check whether it can be considered realistic in terms of performance and in terms of usability with real-world applications (Section 4.1). Given the results that we obtained in this setting, and during the underlying iterative process of improving StarMalloc's implementation, we observed several things. First, that understanding the performance gap between two similar allocators, in our case hardened_malloc and a then in-development version of StarMalloc, can be quite difficult. Second, that more generally, allocators exhibit a variety of behaviors depending on the considered benchmark as well as on the underlying execution environment. Unsatisfied with the tooling we were equipped with to improve StarMalloc's performance and our understanding of allocators behaviors, we developed a tracing prototype specialized for memory management evaluation relying on the Intel Processor Trace hardware feature, described in Section 4.2. While the development of our tracing prototype, named TranscrIPT remains work in progress, as is the analysis of the generated datasets, we present them shortly in Section 4.3.

4.1 StarMalloc: evaluation of the implementation

We now compare StarMalloc with other state-of-the-art memory allocators, including security-oriented allocators, e.g., DieHarder [117] and hardened_malloc [100], and regular allocators, such as the glibc allocator; the next-gen musl allocator [165], part of the musl libc [91]; and the Scudo hardened allocator [166], Android's default allocator.

Doing so, we aim to answer the following two questions: i) is our verification methodology effective enough to retrieve the performance of our initial target, hardened_malloc, and ii) is StarMalloc usable as a drop-in replacement in existing, widely used applications.

Before presenting our evaluation, we first provide an overview of our implementation, see Section 4.1.1. We then present our evaluation, that relies on two sets of benchmarks. First, to provide a fine-grained performance comparison with other allocators, we reuse mimalloc-bench [167], a comprehensive and popular suite which contains both allocator stress-tests and real-world applications (Section 4.1.2). To evaluate the real-world readiness of our allocator, we then describe our integration of StarMalloc within the Firefox browser, which required an extensive set of allocator features, see Section 4.1.3.

4.1.1 Implementation overview

Proof idioms and data structures. As part of the development of StarMalloc, we defined and implemented a range of generic libraries which we instantiated to fit our needs. We already presented several of them in the previous sections; they include data structures (statically allocated doubly linked lists, efficient bitmaps, AVL trees, top-level const arrays, FIFO queues), proof idioms and generic separation logic combinators (`starseq`, extensions and helpers for `vrefine` and `vdep`), concurrency primitives (read-only arrays encoded using dynamically allocated invariants), and modeling of the interaction with the OS (axiomatization of memory-related syscalls, e.g., `mmap`). These libraries required significant engineering and modeling effort, and represent a large part of our development, totalling more than 13,000 lines of F* code. While crucial to the development of StarMalloc, they are a contribution of their own, which significantly decreases the effort required for future systems verification projects in F*. We organized our codebase to ensure they are entirely separate from the code of StarMalloc, and hope to upstream them to the Steel standard library in the future to benefit the community.

Methodology's scalability. A lot of work related to StarMalloc was foundational, and aimed at understanding how to successfully model, verify, and extract a low-level view of memory within the framework. Thanks to the proof methodology presented in Chapter 3, which provides tight abstraction boundaries between the various components of StarMalloc, iterations on the verified codebase are easier to perform. Case in point, a first batch of security mechanisms has been implemented in less than a week, without affecting the rest of the code. The tight abstractions

Component	LoC	Total LoC
Reusable Libraries		13659
Bitmaps	1,520	
Arraylist	3,785	
AVL	4,332	
Misc	4,022	
StarMalloc		28,059
C glue		330
Total		42,048

Table 4.1: Lines of F⋆ code for each component of StarMalloc, excluding whitespace and comments. Misc includes `starseq`, other helpers related to `vdep` and `vrefine`, plus various library functions not found in F⋆'s standard library.

have the added benefit of preserving a lot of intermediary verification files, meaning that one can efficiently rebuild and re-extract StarMalloc when iterating on one sub-component.

Table 4.1 provides figures about our implementation. StarMalloc currently consists of about 42,000 lines of F⋆ code, extracting to close to 6,000 lines of C code (excluding whitespace and comments); our reusable libraries form a large part of this. StarMalloc also contains a small amount of C glue code, needed, e.g., to match our axiomatization of OS syscalls with actual code calling `mmap`. Verifying the whole project requires about 15 minutes on a modern machine, leveraging concurrent builds. Code is available online [2].

Supported APIs. In Section 2.1.2, we have presented the commonly expected APIs a memory allocator must provide. StarMalloc supports the following functions among those that were presented: `malloc`, `free`, `realloc`, `calloc`, `posix_memalign`, `aligned_alloc`, `memalign`, `malloc_usable_size`, as well as `free_sized` and `free_aligned_sized` without hardening (that is, plain stubs to `free`)[1].

The absence of any of these functions can lead to surprising runtime errors. For instance, if `malloc_usable_size` is not provided but called by an application, function symbols will typically resolve to the standard C library implementation, leading to incoherent results when allocations were performed through StarMalloc. Applications rarely specify the complete set of APIs they rely on; this led to issues when trying to experimentally compare StarMalloc's performance to other hardened state-of-the-art allocators (Section 4.1.2) such as FreeGuard [168] or Guarder [169] which do not expose all required APIs for our benchmarking suite.

To circumvent this issue, we performed extensive debugging and sleuthing, to make sure StarMalloc provides all standard, widely used extensions such as `malloc_usable_size`, hence offering a complete API sufficient for use with the Firefox

[1]The C23 specifies that this is acceptable for conforming implementation. While adding checks about the size and the alignment would be reasonable, distinguishing between regular allocations and overaligned ones would require to duplicate size classes, which would be more involved. We leave this as future work.

browser (Section 4.1.3). For other functions that we identified but do not implement (yet), such as `valloc` or `pvalloc`, we provide C stubs that raise a fatal error when called, avoiding silent fallbacks to the C standard library. These functions seem obsolete, and none of the applications we tested StarMalloc with use them.

4.1.2 Performance Evaluation

Experimental Setup. We run experiments on a 12-core (4 performance cores, 8 efficient cores), 16-thread machine with an Intel(R) Core(TM) i5-13600H @ 4,80GHz processor and 32GB memory, using Linux 6.6 and glibc 2.40. To ensure reproducible results, we use the "performance" scaling governor (meaning all cores pegged at maximum speed without dynamic CPU scaling), and increase `vm.max_map_count` from 65,530 to 1,048,576 to ensure allocators using a large number of memory mappings, such as hardened_malloc and StarMalloc for guard pages, work properly. Geometrical mean of 5 runs is used. Aiming for an apples-to-apples comparison, we use the same configuration as hardened_malloc when possible, namely, the guard pages interval set to 2 and the number of arenas set to 4, with quarantine and zeroing enabled.

mimalloc-bench. Our performance evaluation relies on the mimalloc-bench testsuite, also used by recent works on allocators [99, 170, 171, 172, 173]. It consists of two categories of benchmarks. The first series are real-world applications, such as the Lean compiler, the Z3 SMT solver, or the Redis key-value database. The second series of benchmarks stress-tests allocators via specific allocation patterns which are not representative of real-world applications, but that allow one to pinpoint inefficiencies in allocator designs or implementations. As part of this work, we improved mimalloc-bench by allowing the use of the testsuite with recent compiler toolchains, and fixed several compilation issues and bugs in benchmarking scripts; we contributed several patches that were merged upstream.

The most meaningful comparison is between StarMalloc and hardened_malloc. Figures about other allocators help to situate hardened_malloc and StarMalloc in the allocator landscape, compared to both widely used but not necessarily security-oriented allocators (glibc and the nextgen implementation for musl libc [165]), and to security-oriented allocators from the academic literature (DieHarder [117]) as well as from the industry (Scudo [166]). We present experimental results in Table 4.2 and Table 4.3. We attempted to include Guarder [169] or FreeGuard [168] in our evaluation, but did not manage to make them work with our benchmarks. Upon inspection of the code, this seems due to publicly available versions lacking support for several widely used functions, such as `aligned_alloc` or `malloc_usable_size`.

Total Execution Time. A meaningful way to measure an allocator's performance is to measure the client program's total execution time. On the first series of benchmarks (real-world applications), the performance overhead compared to hardened_malloc is within 0.79x to 1.29x. For the second series (stress-tests), we

observe that StarMalloc performs within 0.69x to 1.12x of hardened_malloc on all tests, except for one outlier at 1.36x (`rptestN`).

Broadening our scope to look at other allocators, we remark that our performance and hardened_malloc's is further apart from non-security-oriented allocators like glibc. This is the cost of defensive security measures; ultimately, application developers will have to pick their balance between security and performance. Nevertheless, StarMalloc outperforms mng and Scudo for some benchmarks. More broadly, we observe a greater variety of perfomance profile across allocators, with greater variation on micro-benchmarks than on realistic workloads. These results suggest that no single choice of optimizations is the "right" one for every workload. Additional benchmarks (not listed here) on different processors show that underlying hardware can also significantly influence the final results.

Memory Usage. We measure the Resident Set Size (RSS), with swap disabled to get an accurate reading of the memory footprint of a running program. RSS only counts pages that are actually used, as opposed to the *virtual* memory usage that also contains reserved pages; this is what we want. We observe a wide variety of behaviors; in several cases, StarMalloc outperforms hardened_malloc, while in others, its memory footprint is larger. We posit that this variance is due to minor implementation differences. We remark that glibc's allocator, too, exhibits suboptimal behavior compared to hardened_malloc on several test cases; this again suggests that no design is ideal for all applications.

Differences between StarMalloc and hardened_malloc. hardened_malloc supports a large set of security mechanisms for both large allocations and slab allocations [100]. As part of StarMalloc's iterative verification efforts, we focused on the slab allocator, arguably the most commonly used one and more complex than the large allocator. As a consequence of that, we did not implement security mechanisms for large allocations (guard regions around allocations and delayed unmapping after permission removal).

Regarding security mechanisms for the slab allocator, we support most of them, albeit sometimes with minor differences; all mechanisms requiring randomization are unsupported at the moment, e.g., randomized slot selection within a given slab. While zeroing and guard pages are implemented identically, heap canaries and quarantine implementations differ from that of hardened_malloc. Canaries are implemented using constant magic values on two bytes on StarMalloc side, while cryptographic values (using per size class random state) of eight bytes are used by hardened_malloc. Regarding quarantine, as outlined in Section 3.1.3, StarMalloc implements a quarantine of slabs. hardened_malloc actually implements a finer-grained quarantine with additional metadata for each slot.

4.1.3 Integration into Firefox

To assess whether StarMalloc can be deployed as a drop-in replacement into real-world projects, we integrated it into the Mozilla Firefox web browser, raising

Benchmark	**st**	hm	dh	ff	iso	mi-sec	mng	scudo	sg	sn-sec	glibc
barnes	**1.00**	1.00	1.05	1.01	1.02	1.00	1.02	1.00	1.00	1.00	1.00
cfrac	**0.86**	1.00	1.79	0.70	1.04	0.59	0.71	0.63	0.49	0.48	0.49
espresso	**1.10**	1.00	1.35	0.92	1.05	0.77	1.17	0.80	0.78	0.70	0.78
gs	**1.20**	1.00	0.99	1.71	0.64	0.47	0.78	0.51	0.44	0.51	0.44
larsonN	**0.81**	1.00	58.11	3.27	41.76	0.13	6.45	0.74	0.06	0.05	0.06
larsonN-sized	**0.79**	1.00	55.74	3.22	40.79	0.13	6.34	0.73	0.06	0.05	0.06
leanN	**0.93**	1.00	7.00	0.72	8.39	0.59	2.02	0.70	0.64	0.55	0.63
lua	**1.26**	1.00	ERR	1.07	1.00	0.81	1.03	0.82	0.79	0.80	0.79
mathlib	**0.99**	1.00	ERR	0.79	1.66	0.68	0.86	0.71	0.70	0.65	0.70
redis	**0.93**	1.00	3.09	0.68	1.53	0.80	0.76	0.62	0.58	0.52	0.57
rocksdb	**0.92**	1.00	ERR	0.93	ERR	0.84	ERR	0.86	0.80	ERR	0.80
z3	**1.29**	1.00	1.91	1.07	1.40	0.77	1.03	0.77	0.74	0.74	0.71
alloc-test1	**1.07**	1.00	2.18	0.81	1.12	0.68	0.91	0.62	0.60	0.52	0.60
alloc-testN	**1.05**	1.00	10.52	0.15	6.29	0.06	4.40	0.07	0.06	0.05	0.06
cache-scratch1	**0.98**	1.00	1.02	1.02	1.02	0.99	1.01	0.99	0.99	0.98	0.98
cache-scratchN	**1.00**	1.00	1.11	1.11	1.13	1.02	1.08	1.02	0.96	0.96	0.96
cache-thrash1	**1.00**	1.00	1.01	1.01	1.01	1.00	1.01	0.99	0.99	0.99	0.99
cache-thrashN	**1.04**	1.00	1.08	1.13	1.13	1.04	1.04	1.00	1.02	1.04	1.04
glibc-simple	**0.92**	1.00	1.71	0.59	1.03	0.39	1.07	0.54	0.51	0.29	0.50
glibc-thread	**1.12**	1.00	12.26	0.24	7.80	0.05	5.68	0.06	0.04	0.03	0.04
malloc-large	**1.00**	1.00	1.76	1.00	0.30	0.10	1.00	1.00	0.14	0.29	0.14
mleak10	**0.96**	1.00	1.51	11.19	1.44	0.96	1.33	1.15	0.93	1.12	0.96
mleak100	**1.02**	1.00	1.14	11.41	1.13	1.06	1.09	1.16	0.98	1.16	0.98
mstressN	**0.80**	1.00	9.82	1.43	7.72	0.45	4.47	0.71	0.44	0.37	0.44
rbstress1	**0.87**	1.00	3.10	ERR	ERR	0.77	0.83	0.82	0.81	0.75	0.80
rbstressN	**0.96**	1.00	1.43	ERR	ERR	0.84	0.90	0.83	0.83	0.78	0.83
rptestN	**1.36**	1.00	37.4	5.30	7.82	0.40	4.01	1.14	0.29	0.44	0.29
sh6benchN	**0.69**	1.00	22.12	0.18	10.86	0.03	4.99	1.06	0.05	0.02	0.05
sh8benchN	**0.76**	1.00	14.93	0.21	9.03	0.04	2.84	1.40	0.05	0.02	0.05
xmalloc-testN	**0.85**	1.00	10.32	0.24	5.44	0.06	2.32	3.79	0.12	0.04	0.12
gmean	**0.97**	1.00	3.76	1.00	2.47	0.39	1.59	0.74	0.37	0.32	0.37
min	**0.69**	1.00	0.99	0.15	0.30	0.03	0.71	0.06	0.04	0.02	0.04
max	**1.36**	1.00	58.11	11.41	41.76	1.06	6.45	3.79	1.02	1.16	1.04

Table 4.2: Execution time of various programs, measured against the following allocators: dieharder ("dh", [117], revision 640949f), FFmalloc ("ff", [174], revision 2f24ecf), hardened_malloc default version ("hm", [100], revision 995ce07), isoalloc ("iso", [175], revision 2670d5f), mimalloc-secure ("mi-sec", [99], revision b66e321), nextgen malloc implementation for musl libc ("mng", [165], revision 2ed5881), scudo ("scudo", [166], revision 1654d7d), slimguard ("sg", [176], revision 7d9139a), snmalloc-secure ("sn-sec", [95], revision dc12688), StarMalloc ("st", this work), and the glibc allocator ("glibc", version 2.40). All times are normalized and presented relative to hardened_malloc. ERR corresponds to crashes during benchmarking.

Benchmark	**st**	hm	dh	ff	iso	mi-sec	mng	scudo	sg	sn-sec	glibc
barnes	**0.97**	1.00	1.07	1.56	1.08	0.96	0.96	0.99	0.96	0.97	0.95
cfrac	**1.01**	1.00	1.06	35.51	2.37	0.49	0.36	0.71	0.41	0.47	0.40
espresso	**2.13**	1.00	0.37	1.85	2.38	0.26	0.13	0.24	0.13	0.55	0.12
gs	**1.69**	1.00	1.29	2.98	1.59	0.96	0.82	0.87	0.81	0.95	0.81
larsonN	**1.55**	1.00	1.03	2.83	1.11	1.35	0.50	0.54	1.18	2.24	1.16
larsonN-sized	**1.55**	1.00	1.03	2.83	1.10	1.35	0.50	0.54	1.16	2.25	1.17
leanN	**1.95**	1.00	1.50	8.39	2.65	1.05	0.73	0.90	0.82	1.16	0.82
lua	**1.81**	1.00	ERR	1.26	1.55	0.96	0.75	0.90	0.78	0.91	0.78
mathlib	**1.42**	1.00	ERR	2.47	4.03	0.95	0.68	0.80	0.77	0.87	0.75
redis	**0.45**	1.00	1.00	2.54	2.03	0.55	0.42	0.50	0.39	0.50	0.40
rocksdb	**1.60**	1.00	ERR	2.27	ERR	0.99	ERR	0.94	0.95	ERR	0.95
z3	**1.18**	1.00	0.95	2.46	1.92	0.78	0.65	0.71	0.63	0.80	0.63
alloc-test1	**0.89**	1.00	0.91	33.28	4.16	0.65	0.59	0.72	0.63	0.65	0.63
alloc-testN	**1.55**	1.00	0.42	33.16	1.10	0.65	0.30	0.36	0.35	0.45	0.35
cache-scratch1	**1.05**	1.00	1.23	16.26	2.31	0.85	0.81	0.86	0.81	0.85	0.78
cache-scratchN	**0.99**	1.00	1.22	15.83	2.27	0.86	0.77	0.85	0.82	0.84	0.79
cache-thrash1	**1.03**	1.00	1.20	16.08	2.31	0.85	0.80	0.84	0.80	0.84	0.78
cache-thrashN	**0.99**	1.00	1.21	15.92	2.30	0.84	0.83	0.84	0.83	0.85	0.78
glibc-simple	**1.26**	1.00	0.82	5.43	2.44	0.25	0.22	0.58	0.22	0.31	0.22
glibc-thread	**2.38**	1.00	0.17	0.84	0.80	0.69	0.06	0.11	0.07	1.12	0.07
malloc-large	**1.00**	1.00	1.01	1.18	4.63	1.39	1.00	1.00	1.30	1.64	1.30
mleak10	**0.72**	1.00	1.07	13.04	2.36	0.40	0.32	0.69	0.34	0.44	0.34
mleak100	**0.61**	1.00	0.89	55.73	2.26	0.39	0.27	0.57	0.31	0.39	0.30
mstressN	**2.04**	1.00	1.02	1.21	2.48	2.32	0.84	0.92	1.60	1.61	1.58
rbstress1	**0.96**	1.00	2.38	ERR	ERR	1.52	1.22	1.16	0.97	1.00	0.98
rbstressN	**0.80**	1.00	1.93	ERR	ERR	2.00	1.00	0.94	1.31	1.76	1.38
rptestN	**1.46**	1.00	0.95	1.17	1.18	0.95	0.41	0.44	0.47	1.21	0.52
sh6benchN	**1.05**	1.00	1.53	7.38	5.09	0.91	0.90	1.23	1.10	1.03	1.09
sh8benchN	**1.79**	1.00	1.17	1.18	2.23	1.21	0.92	0.89	1.61	1.18	1.61
xmalloc-testN	**1.98**	1.00	0.33	1.33	0.31	1.09	0.14	1.04	1.24	1.92	1.26
gmean	**1.24**	1.00	0.95	4.63	1.93	0.84	0.51	0.69	0.65	0.90	0.65
min	**0.45**	1.00	0.17	0.84	0.31	0.25	0.06	0.11	0.07	0.31	0.07
max	**2.38**	1.00	2.38	55.73	5.09	2.32	1.22	1.23	1.61	2.25	1.61

Table 4.3: Resident Set Size (RSS) memory usage for various programs, measured against the same set of allocators as Table 4.2. Numbers are, as before, normalized and presented relative to hardened_malloc. ERR corresponds to crashes during benchmarking.

interesting challenges: first, the replacement allocator must provide non-standard extensions of `malloc` and `free` [177]; second Firefox, and hence the underlying allocator, must work in a highly concurrent setting, supporting many threads and processes in parallel.

As a matter of fact, to improve performance and security, Firefox ships by default is own memory allocator, a fork of jemalloc called mozjemalloc [178]. We note that this is also the case of the Chromium browser, that ships its own memory allocator called PartitionAlloc [179], highlighting that applications can implement their own memory allocator and that the actual memory management software stack is more complex that the one we sketched in Section 1.1. A consequence of Firefox using its own allocator instead of the system one (i.e., the libc's one) is that benchmarking our memory allocator using this browser as a workload cannot be done in a straightforward manner.

To explain why, let us recall one platitude: programs are not always required to reinvent the wheel and can use functions belonging to external libraries; in this setting, functions are called *symbols*. Programs using external libraries have two ways of doing so. On one hand, they can include such libraries statically as part of the compilation result, at the risk of increased program size, thus using *static libraries*. On the other hand, such programs can use *dynamic libraries* and assume that the required libraries will be accessible at runtime. In this case, they defer until the last moment the act of determining where external functions are located in memory (*symbol resolution*) so that they can be executed.

As the libc is providing basic functions and assumed to be available, it is commonly used as a dynamic library in standard Linux environments. In turn, as the memory allocator is part of the libc, this can be leveraged to specify at runtime the allocator that should to be used, when the considered libc supports doing so [93]. Using a variable called `LD_PRELOAD`, one can specify to the program's execution environment (actually, the operating system's *dynamic loader*) to prioritize one library over others when loading external functions, e.g., in our setting a memory allocator library over the libc. This technique is called *symbol overload* [50, section 15.1] and is the one that the mimalloc-bench benchmarking suite relies on.

Coming back to Firefox's case, the built-in memory allocator is used by default as a static library, thus rendering the `LD_PRELOAD` trick unusable. As a matter of fact, further demonstrating that StarMalloc is a realistic memory allocator that can be used in real-world workloads, using Firefox was prompted by the existing memory allocator literature using Firefox as an evaluation benchmark regarding time and memory performance [180, 181]. We reused existing methods [181, 180] for recompiling Firefox, using the `-disable-jemalloc` flag, leveraging Firefox's support for replacing its memory allocator. There exists some documentation regarding the required set of primitives to be supported by a memory allocator replacing mozjemalloc [177]. In particular, as outlined in Section 4.1.1, StarMalloc supports most of the seemingly required functions except statistics-related functions, and ensure an error is emitted should other expected yet unsupported primitives be used through the use of dedicated C stubs.

To evaluate the user-facing impact of using StarMalloc, we used JetStream2 [182],

a "benchmark suite focused on the most advanced web applications", also used by existing memory allocator literature, either through Firefox [183][2] or through JavaScript engines [174]. We integrated StarMalloc with Firefox 129.0.2 and successfully ran the entire benchmark suite, demonstrating a score overhead of 0.98x and 1.05x compared to a version compiled with hardened_malloc and glibc respectively. We note that the use of dynamic libraries instead of static ones may incur slight performance differences such as a slight overhead related to symbol resolution. Thus, we only present benchmarks for non-statically linked allocators using the same compiled version of Firefox, though results were similar, as also demonstrated by other benchmarking experiments [184].

We emphasize that JetStream2 does not specifically test the allocator's performance, but rather a comprehensive set of features that must be provided by modern web browsers. The successful execution of these benchmarks therefore indicates that StarMalloc indeed provides all the features required by the Firefox browser, while the score differences, despite not being equivalent to a performance overhead, suggest its applicability to real-world usecases.

4.1.4 Discussion

Reaching competitive results for StarMalloc's verified implementation using hardened_malloc as a baseline was challenging beyond verification-related hurdles. We observe that gaining understanding of memory allocators implementations and their behaviors across various benchmarks was of the utmost importance. On one hand, carefully structuring StarMalloc was done through higher-level understanding of hardened_malloc's implementation. On the other hand, "bridging the gap" between the performance of StarMalloc and hardened_malloc required us to gain understanding of some optimizations on hardened_malloc's side that lead to significantly improved results on some benchmarks. In this last case, understanding why a specific benchmark strains implementations, e.g., leading to crashes or large overhead, sometimes was a prerequisite to understand the need for some specific optimization.

More specifically, we observe two things. First, while iterating on an incomplete proof, we note that verification guided debugging: the more a portion of code is exempt from assumptions using `sladmit`s (briefly presented in Section 2.2.3), `admit`s or `assume`s, the less it is inclined to exhibit unexpected behaviors; that is, in the general case, when specifications are correct. The process of verifying a function itself generally was the following: first, proving only the correctness of the memory shape (and the first couple of spatial pre- and postconditions), using `sladmit`s when needed[3]; then, moving onto proving the correctness of the memory content (and the second couple of pre- and postconditions), using `admit`s when needed; finally, refining specifications of the function, using leftover `assume` eventually progressively removed. In such cases, Steel was first used as a (most of the time) quite pleasant

[2]We note pSweeper not strictly is a memory allocator but rather a runtime defense that completes memory allocators.

[3]A noteworthy limitation is that using two `sladmit`s in the same function is not supported by the automatic frame inference.

low-level programming language avoiding many pitfalls to the programmer.

Second, debugging mostly amounted to finding relevant traces of memory management functions. In that case, side effects from the rest of the program are not relevant, as the considered program can be easily checked to be correct using other allocators. In turn, isolating memory management functions calls helps determining the source of the unexpected behavior (e.g., a crash or a performance overhead).

Initially aiming at developing basic corresponding tooling, we found ourselves in a situation similar to that described by Firefox developers investigating memory management performance [184]. One of their core ideas it to use a tool that can log all Firefox internal memory management calls so that any issue that may be related to it can be reproduced by only replaying all corresponding memory management calls, using a tool called logalloc [185]. This idea is in fact already present in the survey on dynamic memory allocation by Wilson et al. [49], that proposes to collect such traces so that they can be used for "simulation" purposes. While it is fair to say that we did not look down on so-called print-debugging to implement something similar for StarMalloc, we were soon limited by the fact that our basic tooling only supported StarMalloc and faced large performance overhead when used on demanding benchmarks.

This led us to develop specialized tooling, including a prototype of a generic tool using baked-in CPU hardware features that we describe in the rest of this chapter.

4.2 Background on profiling and Intel Processor Trace

The need to gain insight into what actually happens when executing programs generalizes by far our need as verified implementers to gain better understanding of memory allocators. As the 1994 ATOM paper puts it [186], "Program analysis tools are extremely important for understanding program behavior. [...] Software writers need tools to analyze their programs and identify critical pieces of code." The need for such analysis tools also predates the wide availability of computers such as personal computers. Indeed, Bernecky [187] gives example of such tools that can be dated as back as 1974, namely the use of the PSW[4] on IBM S/370 machines to produce per-program processor time statistics.

At this point, it is actually needed to distinguish between two different activities: on one hand, debugging; on the other hand, profiling. Debugging can be defined the following way: it is "the diagnosis of mistakes in programmes (sic)" per the title of a 1950 paper [188], when such programs that are "presented to the machine are not those required to obtain the results sought". This paper considers mistakes on the venerable EDSAC[5] at the Cambridge Mathematical Laboratory in the late 1940s.[6]

[4]Program Status Word

[5]Electronic Delay Storage Automatic Calculator

[6]We note that, at the time, hardware reliability was an important concern, as they distinguish such mistakes from those "resulting from faults in the machine itself". Today, the hardware is considered as an "unusual suspect" [189], something that Thomas Nicely also described when narrating his investigations in 1994 about the FDIV Pentium bug. [190]

Among those results produced by the machine, we may distinguish between the results themselves, that is, the output, and the conditions in which they occurred: the time spent producing them, the amount of energy used to produce them, etc. Profiling corresponds to this second set of properties. Bernecky [187] mentions two different definitions of profiling. The 1988 Webster's dictionary definition is cited in the paper; according to it, a profile is "a set of data often in graphic form portraying the significant features of something". Authors then propose a second context-specific definition: "for our purposes, profiling is the analysis of a running computer program in order to determine its actual, rather than predicted, behavior. Profiling may be performed manually, or automatically, with the aid of hardware or software. The data collected by a profiling activity [...] typically will allow determination of instruction mix, storage reference patterns, and instruction reference patterns".

In this section, we lay out a brief history of hardware CPU profiling techniques, from sampling to tracing in Section 4.2.1, and then proceed with presenting the Intel Processing Tracing hardware feature in Section 4.2.2, used in our memory allocators profiling prototype, presented in Section 4.3.

4.2.1 A brief history of hardware profiling, from sampling to tracing

Hardware features baked into CPUs allowing one to proceed with performance analysis greatly evolved since the 1990s with Intel Pentium CPUs and their hardware performance counters [191, 192]. Such features actually now form a very diverse set, with modern CPUs generally supporting more advanced features. As Weaver [193] presents it, a common way to perform profiling is to use *sampling*, that is, to interrupt periodically the CPU to record part of its internal state, and from this extrapolate what the "full system behavior" actually is. They also presents the fundamental tradeoff at stake in a simple setting between overhead and accuracy, depending on the sampling frequency: sampling more frequently leads to more accurate results as well as a higher overhead. Modern CPU hardware features try to help regarding this matter.

Profiling hardware features aim at allowing low-level details of the behavior of executed programs on the underlying system. As such, as investigated by Sasongko et al. [194], even if AMD and Intel x86_64 CPUs mostly share the same ISA[7], there are important differences in terms of supported baked-in profiling features depending on the vendor. Let us have a look at the corresponding set of features for Intel CPUs, most relevant in our case as we used an Intel hardware feature as the basis for our prototype described in Section 4.3.

Leveraging previous work [193], we distinguish between the following features.

- Sampling through per-sample interrupts. Fixed and programmable performance counters are implemented as part of PMUs[8] in CPUs. On one hand, fixed counters always count the same sort of events, such as the number of

[7]Instruction Set Architecture
[8]Performance Monitoring Units

retired instructions [195]. On the other hand, programmable counters can be used to measure user-defined classes of events. Using periodical interrupts, counting events can be made into sampling [196]. Underlying counters actually are MSRs[9]: their number and width can vary, depending on the underlying CPU model.

- Low-latency sampling through Intel PEBS[10]. This has two advantages over bare performance counters. On one hand, theoretically reduced overhead: samples are stored in a dedicated ringbuffer, interrupts are issued only when it is full [197, 194]. On the other hand, latency (in terms of instructions) is lower, thus avoiding so-called *skids* [195], that is, mistakes when tracking instruction causing events (due to microarchitectural details) [198].

- Control-flow partial reconstruction through Intel LBR[11]. Instructions causing a change in the instruction pointer (beyond usual calls and branches: interrupts, traps, faults) are logged in a ringbuffer. This allows computing a complete "hot code" call graph (restricted to "hot code" as this still is sampling) and thus profile-guided optimization (PGO); as well as debugging crashes, among other things [199, 200, 197].

- Control-flow tracing using Intel Processor Trace (IPT / Intel PT). This technology has been more recently deployed (since Intel Core 5$^{\text{th}}$ generation "Broadwell" [201, 202]), and does not rely on sampling anymore. It is a CPU tracing feature that logs any branch instruction details required for exact reconstruction of the control-flow and compresses it on-the-fly. The basic idea is that the minimum information required to reconstruct the entire control-flow is stored: for each conditional, one bit indicating whether the branch was taken is stored. Precise timing information can also be stored within the compressed trace. The trace output can then be decoded using the executed binary: the resulting final file can be very large, as we will see. In most cases, overhead remain small, around 5% according to several observers [203, 204].

On Linux, developers can access most of this information through the `perf` Linux profiler [205], also sometimes called `perf_events` [206]. It can be seen as the profiling swiss-army knife and exposes many Linux sampling and tracing features, though not all of them.

4.2.2 Using Intel Processor Trace in practice

All data collected using IPT is stored under the form of highly compressed binary packets that can later be decoded, including "taken-not-taken" `TNT` packets storing control-flow information. Assuming the executed program binary still is available, decoding allows one to retrieve the exact control-flow of underlying executed software. Crucially, it is not needed to recompile this software, e.g., to instrument it.

[9] Model-Specific Registers
[10] Processor Event-Based Sampling, or sometimes Precise Event-Based Sampling
[11] Last Branch Records

This two-step process takes the following form using `perf`. First, data is collected with `perf record` and stored by default in the current directory as a `perf.data` file. The general syntax is the following:

```
$ perf record -e intel_pt/$events_config/$modifiers -- cmd
```

where `$events_config` allows one to specify which events packets should be produced and/or collected, such as (possibly very) precise timing packets, such as `CYC` packets[12]; `$modifiers` can be used to restrict tracing, e.g., to userspace or kernel mode. As an example, a userspace-only trace of the execution of the `uname` program printing information about the system can be captured using:

```
$ time perf record -e intel_pt//u -- uname
```

yielding the following output:

```
Linux
[ perf record: Woken up 4 times to write data ]
[ perf record: Captured and wrote 0.085 MB perf.data ]

real    0m0.027s
user    0m0.008s
sys     0m0.011s
```

Second, collected data can be decoded with `perf script` or read plain using `perf report -D`, displaying raw `TNT` packets. As collected data is highly compressed, large compressed traces can lead to very large uncompressed data: decoding the `uname` trace for calls leads to a file of several MiBs; for branches, of dozens of MiBs.[13] As we will see shortly, the trace provides various information, including for each instruction the name of the underlying program, the thread id, the CPU core used for execution of the instruction, timing information as well as the executed assembly instruction. A higher-level view such as the call trace can also be obtained using `perf script -call-trace`. All in all, a large amount of information is available for later analysis, which one can filter.

Another feature of IPT that gives it a lot of flexibility is the `ptwrite` instruction. This instruction can be used to define additional "software-defined events", as Chen et al. [207] put it; in short, it allows one to include custom additional data in IPT traces [208], at most 64 bits for each `ptwrite` instruction. Using it is fairly simple: all is required for the recording step is enabling `PTWRITE` packets logging corresponding to executed `ptwrite` instructions. This corresponds to setting `ptw=1` as part of the aforementioned `$events_config`, e.g., the following way.

```
$ perf record -e intel_pt/ptw=1/u -- ./a.out
```

As an example let us assume that the target program is the following one that only logs the value 42 (`0x2a`) using the `ptwrite` instruction through inline assembly.

[12] These can be configured so that a timestamp `CYC` packet is sent whenever another packet is sent, with a configurable minimum interval expressed as a number of CPU cycles. This number can be set to be 1.

[13] Using `perf script -itrace=c` and `perf script -itrace=b` respectively.

```c
#include <stdint.h>
void log_ipt(uint64_t x) {
  __asm__ ("ptwriteq %rdi");
}
int main(){
  log_ipt(0x2a);
  return 0;
}
```

The corresponding output when specifying decoding to only filter `PTW` events[14] is the following.

```
a.out  220113 [006] 33450.170884:  ptwrite:  [...] payload: 0x2a
```

From left to write, this includes the program currently executed, its thread id (tid), the CPU core, the time in seconds since the underlying machine boot, and the actual logged value in hexadecimal, equal to 42 here.

Hardware support for the `ptwrite` instruction is more recent than support for IPT (only since Intel Core 12$^{\text{th}}$ generation "Alder Lake" [209, Chapter 1.3]). It should however be noted that `perf` supports software-emulated `ptwrite` instruction [202].

4.3 TranscrIPT: hardware-based memory management tracing

When improving StarMalloc's implementation in order to try reach parity with hardened_malloc in terms of performance, it was sometimes unclear as to where last-mile efforts regarding the performance gap should go. We have seen in Section 4.1.4 that keeping a precise trace of memory management related functions calls was considered relevant and used by Firefox thanks to a dedicated tool called logalloc.

However, beyond the fact that this tool only is applicable to Firefox, this tool has several limitations. On one hand, to the best of our knowledge it does not keep timing information, e.g., how much time is required for `malloc` to provide the user client with a memory allocation. On the other hand, it uses a single mutex to serialize logs, likely incurring performance overhead as well as distorted traces.

In this section, we present a tool named TranscrIPT that aims at providing one with precise traces of memory management functions on real-world workloads. From a high-level perspective, we aim at leveraging collected data to answer questions such as the following ones. How does allocator X compare to allocator Y regarding subpage overaligned allocations? regarding the reallocation of subpage allocations into subpage allocations?[15] How does the execution time of deallocation function

[14] This can be done using `perf script -itrace=w`.

[15] Let us recall that we observed that this pattern can lead to performance issues, see Section 3.5.3.2.

vary given the size of the memory block specified to be deallocated? Assuming collected traces can be replayed, the following questions could also be investigated. Given one benchmark, is the difference regarding time execution only related to memory management or an outcome of side effects from the rest of the programs, such as cache-related issues? On the opposite, is the lack of difference regarding time execution only due to the fact that memory allocators are not strained by the considered benchmark?

Our design constraints thus are the following. First, our tool must be generic and support any pair of memory allocator and benchmark without requiring manual patches of corresponding implementations nor recompilation. Second, collected traces should be complete and include all memory management related calls with complete arguments as well as their execution time. Finally, aiming at collecting precise representative traces that do not significantly stray from normal execution behaviours, this tool must target a very low overhead in all possible cases.

Research questions that we aim to answer are the following.

RQ1. Using state-of-the-art hardware features, is this set of constraints satisfiable? Put simply, is this feasible?

RQ2. Can collected traces be analyzed in practice?

RQ3. Does our tool actually improve our understanding of memory allocators?

RQ4. Does our tool compare favorably with other tools, e.g., are collected traces more precise?

We present preliminary work towards answering these questions. It is likely that our current prototype will as we make progress i) be further refined ii) only be one way to get data among a collection of tracing and profiling scripts. Corresponding code is made available online [3].

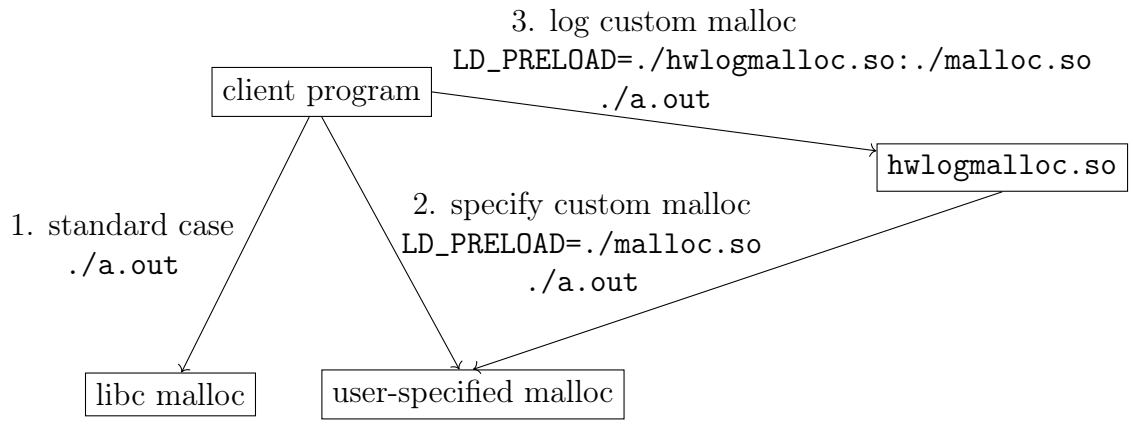
We first present TranscrIPT's current design in Section 4.3.1 and then corresponding preliminary results in Section 4.3.2.

4.3.1 Design and implementation

The resulting prototype in its current form leverages Intel Processor Trace and its extension based on `ptwrite` instructions logging. TranscrIPT's main technical component is a (dynamic) library named `hwlogmalloc.so` intercepting memory management functions. Its role is to insert carefully tailored `ptwrite` instructions (more on this below) used for two purposes: to store log arguments and returned values as well as delimit actual memory management calls so that the corresponding execution time of each call can be computed. One additional constraint is that corresponding encoding of these calls should be unambiguous for later offline analysis. Using linking tricks, actual memory management calls are redirected to a user-specified memory allocator, see Figure 4.1.

Using this library and a carefully-tailored `perf record` invocation, a trace is produced. Decoding the collected trace can then be done through `perf script`. As the collected amount of information can be very large, the resulting compressed file is never entirely decompressed for analysis. Instead, due to the fact that IPT

Figure 4.1: Interception of user-specified memory allocator by `hwlogmalloc.so`.



compression encoding favors compression time rather than decompression time, decompression time can be large. Thus, a first analysis pass decompresses the trace, parses it and compresses the result using an algorithm favoring decompression speed, all of this on-the-fly, so that the result is stored as a compressed file that can efficiently be read.

We just presented a high-level picture of the process. Now, let us first give more details about the encoding. To limit overhead, the number of `ptwrite` instructions and thus of `PTWRITE` packets must itself be limited; in addition to that, consecutive `ptwrite` instructions can lead to performance overhead. Thus, we try to use exactly two `ptwrite` instructions for each memory management call, respectively at the beginning and at the end of the execution of the function.

As an example, let us present the encoding for `malloc`, `free`. It relies on one low-level assumption, that in x86_64 Linux environments, pointers only are 48 bits wide [50]. This way, the following encoding can be safely used in practice, with following naming conventions. The `void*` returned pointer by `malloc` is named `ptr`, corresponding to a `size_t` specified size named `size`; the pointer to be deallocated by `free` is also named `ptr`. Let us also recall that in C, `1UL << n` is used to define an integer such that only its n-th bit is set; x `|` y denotes the bitwise or of x and y.

function	packet 1 (header)	packet 2
`malloc`	`(1UL << 63) | ((uint64_t) size)`	`(uint64_t) ptr`
`free`	`(1UL << 62)`	`(uint64_t) ptr`

The first packet is used as a header to distinguish between different functions, while storing additional information. In the case of `malloc`, using the fact that pointers do not use all 64 bits, the integer corresponding to requested allocation size is modified such that the most significant bit is set. Another implicit assumption making this encoding unambiguous is that the trace stores the thread identifier, relying on the fact that a single thread cannot execute two functions at the same time. Thus, once an unambiguous header has been parsed, the second packet can be parsed

accordingly.

Other assumptions can be made to encode in a very compact way other memory management functions. First, regarding requested memory allocation sizes, memory allocators limit valid allocations to `PTRDIFF_MAX`, the maximum representable `ptrdiff_t` integer, so that pointer difference within provided memory allocations are restricted to defined behaviors.[16] In fact, a more efficient way of encoding sizes is to assume that they fit on n bits, e.g., with $n = 30$, using another tool or runtime checks to ensure that the considered application does not request memory allocations whose size cannot be represented using n bits.

Second, regarding pointers, assuming they are 16-bytes aligned, the 4 least significant bits are equal to zero: a pointer thus can be represented using 44 bits. Finally, considering memory management functions presented in Section 2.1.2 and commonly used as seen in Section 4.1, there exists more than 8 memory management functions of interest and less than 16. Thus, 4 bits are needed for the header.

Using such assumptions, all arguments of memory management functions as well as their return value can be encoded using two `ptwrite` packets. As an example, considering a `void* new_ptr = realloc(ptr, new_size)` memory management request, where `ptr` is of type `void*` and `new_size` of type `size_t`, the amount of information to be stored is the following.

- 4 bits are required for the header;
- 44 bits are required for `ptr`;
- 30 bits are required for `new_size`;
- 44 bits are required for `new_ptr`.

The total amount of information to be stored in bits thus is 122 bits, that fit on two 64-bit `ptwrite` operands.

Current version of TranscrIPT only log the following functions: `malloc` and `free`. Supporting other memory management primitives is left as straightforward future work using the presented encoding.

The decoding step parses resulting trace of `PTW` packets, and ensures that the resulting final file can efficiently be read and analyzed through the use of a compression algorithm featuring correct decompression speed for practical reasons.

4.3.2 Preliminary experimental results and perspectives

We use TranscrIPT on various benchmarks that are part of the mimalloc-bench benchmarking suite (see Section 4.1.2). This includes allocation-intensive benchmarks such as the `cfrac` and `espresso` sequential benchmarks; as well as multi-threaded thread tests such as `sh6bench`. We verified using another tool named SystemTap providing us with the number of each memory management operation that these benchmarks mostly feature `malloc` and `free` memory management operations. Finally, we were cautious to only require `IPT` through `perf` to trace relevant packets. In practice, we use `ptw=1,fup_on_ptw=0,branch=0` modifiers for the `perf record` invocation.

[16] See `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=118220`.

The used experiment settings are very similar to those used in Section 4.1.2. Relying on these, we initially had trouble obtaining reliably complete traces, which manifested itself in the form of varying sizes of resulting collected compressed traces. Collected traces could then be parsed and checked to be consistent or not with statistics collected using SystemTap. As an example, traces collected using the glibc allocator on the `cfrac` benchmark can vary from dozens of MiBs to roughly 5GiBs.

To improve the reliability of our setup, we refined our experiments settings in two ways. On one hand, benchmarks were run with high privileges, using the `root` user. This improved our results, we thus suspect that some priority-related issue is at stake and did not investigate it further. On the other hand, completeness of the trace can be related to limited memory bandwidth. Let us recall that the `cfrac` benchmark only takes a few seconds to execute. To improve on this, we enforce the collection of data to happen in an in-RAM filesystem, thus reducing I/O overhead. This, in turn, requires a corresponding large RAM size.

Using these refined settings, we observe that the overhead induced by the use of TranscrIPT generally ranges from 1.5x to 2x, sometimes less, especially when using other allocator than the glibc one, such as hardened_malloc. We posit this is due to the fact that an allocator with lower throughput (that is, performing, less memory management operations per second) yields less pressure on the IPT feature.

One of the current limitation is that long-running benchmarks (such as `sh8bench` using hardened_malloc) yield very large traces that do no fit within the RAM of the considered system. Overall, while much work is needed to improve our TranscrIPT memory management tracing prototype, we consider these first results as encouraging. We note that an additional recent extension of IPT allows one to entirely disable TNT packets [209, p.41] even though these form the basis of the control-flow reconstruction. In our setting, this could be especially interesting to reduce TranscrIPT's overhead as well as the size of collected traces, as acknowledged by the documentation [202].[17]

We discuss other already existing tools in Section 5.2.

4.3.2.1 Perspectives

Presented preliminary results may give the misleading impression that traces collected using TranscrIPT can only be used to measure execution time, while doing so in an admittedly very precise way. In this section, we lay out several possible research directions extending or repurposing TranscrIPT.

First and foremost, a first reasonably straightforward extension of this work would allow one to measure memory performance as well. Indeed, the fact that collected data includes sizes as well as addresses of allocated memory blocks could be leveraged to reconstruct the memory layout throughout the entire underlying considered program's execution. Doing so could rely on knowledge about the page size of the considered system or even on additional tracing, e.g., of `mmap`-like syscalls.

[17] However, we could not find documentation indicating which commercially available CPUs support this feature.

This last point could be achieved using a tweaked version of the underlying libc, recompiled to instrument syscalls with `ptrwite` instructions.

In addition to that, given one allocator, collected data could be used to assess the quality of some of its security mechanisms. Indeed, addresses could be used to determine the reuse frequency of memory blocks and thus the degree of efficiency of security mechanisms aiming at lowering (virtual) addresses reuse, such as quarantines. The isolation between size classes could also be evaluated in a similar manner.

To conclude by giving a broader perspective, we note that the considered hardware-based technique to collect the entire memory management APIs calls could actually be leveraged beyond the context of C userspace memory allocators. Indeed, on one hand, other programming languages memory management primitives such as C++ `new` and `delete` operators (that come in typed versions) can be implemented as wrappers on top of C `malloc` and `free`. In this setting, the isolation between allocation of different types could be investigated, again by analyzing collected addresses. On the other hand, the Intel Processor Trace feature can also be used to trace kernel events. Thus, similar techniques could hypothetically be applied to the Linux kernel memory allocator, itself under scrutiny regarding its performance and security properties [149, 210].

Chapter 5

Related work

5.1 Systems verification

Verification methodologies abound for low-level programming, and given their critical aspect, memory allocators have long been considered an interesting target for low-level formal verification. We now focus on those low-level verification methodologies that have been specifically applied to allocators. Whenever possible, we highlight differences both in methodology and in verification artifact. We recall that our mixture of dependent types, separation logic, and semi-automated SMT-based verification in Steel allows us to meta-program and verify an allocator that extracts to 6.0kLoC, and features the wide array of defensive measures expected of a modern, secure allocator.

Wickerson, Dodds, and Parkinson [211] use rely-guarantee [212], in combination with separation logic, to verify the Unix Version 7 (1979) memory manager, while reasoning about concurrency. This early memory manager consists of a linked list of *all* the blocks. Finding a free block requires linearly traversing all the blocks; block headers consists of a single memory word pointing to the next block, with the low bit encoding whether the current block is available. The verification is pen-and-paper, and as such relies on a manual encoding of that allocator into the framework.

Using the Isabelle/HOL [213] proof assistant, Sahebolamri, Constable, and Chapin [214] verify a simple sequential memory allocator where a linked list intertwined with data keeps track of allocation units. The methodology relies on AutoCorres [215], which was developed as part of the seL4 microkernel verification effort [71]. This tool takes C code and produces a higher-level monadic specification, along with a proof of correctness of the translation. Owing to the (deliberately) simplistic design, it suffices to directly establish invariants over the result of the translation, without any particular framework or proof style.

Appel and Naumann [216] verify an array-of-bins malloc/free system using the Verified Software Toolchain [217]. VST allows reasoning over a deep embedding of C in Rocq (formerly Coq), via the Compcert semantics. This allows for precise modeling of C, at the price of possibly more involved proofs. VST relies on a concurrent separation logic, but unlike Steel, does not use an SMT solver and thus

relies entirely on tactics for automation. This allocator fits in 108 lines of C, is only proven correct in the sequential setting, and is intended for use with verified clients, meaning it has no defensive security mechanisms. They do, however, model resource awareness and exhaustion.

Sammler et al. [218] verify C code using the Iris framework [219, 220, 221], a recent higher-order concurrent separation logic framework in Rocq. An ownership discipline, applied on top of C code, allows translating C programs into Iris, and thus proving their correctness against user-provided annotations. Further properties can be derived in Rocq by building upon this first layer. The authors verify a thread-safe allocator (68 C LoC), and a page allocator from a hypervisor (191 LoC). The latter is their largest case-study.

Pulte et al. [154] propose CN, a new separation logic and refinement type system that captures almost the entire ISO C standard; the authors claim it allows reasoning about code as written by the programmer. Like ours, their system mixes manual proofs with automated SMT-based reasoning. Source C code is translated into a core language, and the specification language is designed so as to make sure user-provided refinements remain in a logical fragment that can be decided via SMT. Anything outside of this fragment (including non-linear arithmetic) needs to be isolated in noninterpreted functions and reasoned about manually in Rocq. Unlike the system we used, their system does not support concurrency. Furthermore, we leverage meta-programming to generate different variants of our implementation; it is unclear if CN would support an equivalent, C-native implementation based on macros. Their flagship example is a buddy allocator from Android's pKVM, which comprises 364 lines of C code, excluding comments; its verification required several modification to its C code.

Lattuada et al. [222] leverage Verus [223] as a system verification language to tackle the verification of several case-study systems. Their verification methodology heavily rely on SMT-driven proof automation, leveraging the use of linear types to simplify verification conditions sent to the SMT solver. While separation logic is not used, Verus supports the use of linear ghost permissions to express ownership of (raw) heap pointers. In addition to that, authors add to Verus as part of their systems verification effort a dedicated approach called VerusSync to reason about concurrency. To this end, they base it on resource algebras, noting that this concept has been used in concurrent separation logic implementations. As part of their case-study systems, they port existing verified systems to Verus to enable comparison with the Dafny [224] and Linear Dafny [225] verification frameworks against which they compare favorably regarding verification time of the resulting artifacts. Most importantly, their largest case study is a verified memory allocator based on mimalloc [99], providing `malloc` and `free` functions and relying on OS interfaces such as `mmap`. The resulting memory allocator called Verus-mimalloc does not support `realloc` nor aligned allocations and only support a subset of the mimalloc-bench benchmarking suite. Verus-mimalloc implementation totals 14k LoC of proofs that can be reverified in around a minute and results in 3.1k LoC of executable Rust code.

Other works [226, 227] verify *simplistic* memory allocators, but to the best of our

knowledge, none verify a state-of-the-art, competitive, real-world allocator. Zhang et al. [228] verifies a non-trivial allocator design, but only a *specification*, not an actual implementation.

Replacing the system libc's allocator is a challenge, and requires implementing a wide API surface (both POSIX *and* non-standard but widely used APIs) along with several advanced mechanisms for performance. Furthermore, having a security-oriented allocator requires implementing a vast amount of features, which increases the verification burden. This explains why none of the allocators we reviewed claim to act as a *drop-in* replacement for the system allocator, and study implementations that are at best an order of magnitude smaller than the code we verify.

Beyond userspace allocators, previous verification projects tackled verified operating systems and microkernels, and thus kernel space memory management [229, 230, 72, 231]. Being in-kernel, these allocators have different constraints. First, kernels have many static data structures so these allocators do not hit many small heap-based allocations; second, kernels oftentimes have global locks for data structures [232], meaning concurrency isn't as much of a concern for the allocator. Furthermore, in the context of verified OS kernels like above, additional proofs ensure that the in-kernel allocator does not need to be defensive against other parts of the kernel, which means even in this context, a full-fledged defensive general-purpose security-oriented allocator was not typically the object being studied.

On adjacent area of related work is garbage collectors (GCs). Numerous runtime systems and GCs have been studied and verified, starting with Doligez and Leroy [233] and Doligez and Gonthier [234], all the way to CakeML [235]. While verifying a GC involves similar data structures as those of an allocator like StarMalloc, the verification challenge is different.

Zakowski et al. [236] focus on verifying the mutator in conjunction with the allocator, and showing that the operation of the two maintains the proper invariants in a concurrent setting. The authors adopt a modular verification methodology too, but that is specifically tailored to their implementation of rely-guarantee reasoning in Rocq. Like us, they proceed incrementally with multiple layers of abstraction to make verification tractable. The description of the algorithm focuses on the mark-and-sweep phases, and the only data structure mentioned in the paper is a free list for the pool of unused references. Finally, we remark that this is a non-executable specification, which does not yield or is formally linked against executable code.

Sandberg Ericsson, Myreen, and Åman Pohjola [235] operate within CakeML, meaning that their proofs are carried in HOL4 over actual SML code. The GC of CakeML relies on a moving approach where fresh allocations within a heap rely on a bump-pointer allocator, and where compaction happens when the old heap is copied into the new heap. As such, the challenge lies more in the correctness of the traversal and reachability analysis, rather than the allocation primitive itself.

Shamsu et al. [237] present a verified garbage collector that can serve as a replacement for OCaml's GC. Authors prove their GC correct using Low* [141], a language for low-level programming and verification shallowly embedded in F*. To render their verification effort tractable, authors adopt a layered verification methodology, separating proof obligations in these different layers. First layer provides abstract

graphs and depth-first search reachability specifications. Second layer consists of a purely functional implementation of a mark and sweep GC operating on a specification of the OCaml heap reflecting actual memory layout. Although functional, this layer already uses machine integers, thus getting closer to the actual final implementation while tackling bitwise operations proof obligations. Third and final layer implements an imperative mark and sweep GC whose functional correctness is rooted in the specifications of the functional GC: this last layer is proven to respect these. Extraction to C code is done using KaRaMeL. The resulting artifact is integrated with OCaml 4, then evaluated on a variety of benchmarks in comparison with the OCaml GC, used as a baseline, demonstrating competitiveness.

One last area of related work is other languages well-suited for memory management systems verification. Ebner et al. [238] describe PulseCore, a new higher-order, dependently-typed concurrent separation logic shallowly embedded in F*. It can be seen as a natural continuation of SteelCore [138], aiming at refining it by removing one axiom deemed brittle and in particular offers more expressiveness through higher-order invariants, whereas SteelCore only supported first-order invariants. Intended to serve as a semantic foundation for the Pulse proof-oriented programming language (not described in the same paper), they present several libraries verified using Pulse, including an OCaml5-style task pool showcased with some task-parallel programs.

Cronburg and Guyer [239] describe FloorPlan, a domain-specific language for describing the layout of memory management structures. FloorPlan accounts for blocks, bitmaps, linked lists, and aims to automate the generation of these memory management systems rather than having programmers write error-prone code by hand. The chief use-case is runtime systems for managed languages, but this could be repurposed for StarMalloc. FloorPlan, however, produces *unverified* Rust code, meaning it is not directly applicable. One could conceivably add a verified Steel backend for FloorPlan, or conversely, switch the verification technology to, e.g., use Aeneas [240] or Verus [223] and directly verify the Rust code. Both remain future work.

5.2 Benchmarking memory allocators

There exists a broad literature regarding memory allocator design, investigating how to improve memory allocation state-of-the art performance regarding various constraints, e.g., execution time, memory usage or scalability [52]. We note that there exists literature regarding the detection of inefficient memory use by applications such as the Hound runtime by Novark, Berger, and Zorn [241], that we consider out of scope here.

Corresponding benchmarking efforts account for these various constraints affecting memory allocators design. However, improving an implementation to meet benchmarking-fixed performance thresholds in comparison with other allocators requires one to gain understanding of both considered benchmarks and allocators, which is costly. In turn, this calls for tools helping one to improve this understand-

ing of memory allocators behaviors when benchmarked as well as in the general setting. In this section, we focus on such tools that have been specifically applied to memory allocators. We recall that our TranscrIPT prototype provides one with generic, low-overhead and complete tracing of memory management related functions, making precise offline analysis possible, admittedly at the cost of large traces.

As already noted by Ball and Larus [242], tools targeting program analysis by collecting data to this end can be divided into two sorts: profiling tools and tracing tools. We consider fundamental distinction also relevant to our setting of analyzing memory allocators behaviours on various programs, as corresponding incurred overheads largely differ. Thus, we first consider profiling tools and then tracing tools, more specifically focusing in both cases on their applicability to memory allocators and benchmarks understanding.

Profiling. Profiling tools shine at providing coarse data about programs, such as aggregated data (e.g., mean, minimum and maximum, counts and histograms) or sampled data that does not require to store the entire trace.

There exists various tools in this area that can be used for the purpose of memory allocators profiling. A first form of profiling can be done through flamegraphs, determining through statistical sampling which parts of the code are using most of a given sort of memory resources (e.g., CPU time or memory). Tools supporting general profiling include Linux's `perf` and `eBPF` for CPU time resources. Regarding memory profiling, there exists various heap profilers; let us cite the Valgrind Massif heap profiler [243] and the PROMPT memory profiling framework aimed at developing fast memory profilers given specified user events, developed by Xu et al. [244]. We insist on the fact that in our setting, we focus on memory allocators performance, not that of underlying programs memory efficiency: flamegraphs can be here mostly leveraged to determine where most time is spent inside a given memory allocator implementation.

Another form of profiling can be done through the collection of the number of function calls over time. In our setting, this can be interesting to log memory management related calls to the C standard library, e.g., `malloc` and `free`, as well as to the operating system, e.g., the `mmap` syscall. This includes tools such as ltrace, DTrace, SystemTap, bpftrace and lttng. To the best of our knowledge and after preliminary experiments, the four five tools cannot be used in a scalable manner to record entire traces of memory management calls, but can only provide efficiently aggregated data such as the number of calls or histograms of arguments and returned values. We however note that `lttng` would deserve further investigation, even though it does not to the best of our knowledge support the on-the-fly compression of collected records, thus likely incurring prohibitive memory usage. Despite what their name suggests, they mostly seem suited to trace the origin of events (and thus are especially useful to monitor entire systems) rather than to provide complete execution traces of considered programs.

Finally, precisely measuring memory use over time, although this can seem simple at first glance, can be in itself a complex endeavor. Memory allocators often provide as builtins statistics about the client program current memory usage. This can

also be done through builtin memory allocator support, such as the Google heap profiler shipped with the tcmalloc allocator. However, programs may be provided with memory through various means, e.g., directly from the OS or by other memory allocators in case of partial replacement of the libc's allocator. The mstat tool [245] takes advantage of Linux-specific API to track memory consumption of a given set of processes over time and was created as part of the development of the Mesh compacting memory allocator by Powers et al. [181]. We note that it does not depend on the underlying memory allocator.

Zhou et al. [246] tackle the task of comprehensively characterizing the TCMalloc memory allocator, used in production by applications running on the large servers fleet in use at Google. As such, this allocator is equipped with specific features, leveraging the kernel support for fine-grained memory management of contiguous physical pages for the sake of performance as well as optimizing cache use for the underlying hardware architecture. Authors leverage fleet through randomized per-machine brief yet globally continuous profiling to get precise statistics used to perform a quantitative analysis of TCMalloc behaviors as well as memory allocation patterns of the considered applications. Comparing the results with those that can be obtained by using commonly used benchmarks, they argue that the latter are insufficient for their use. Leveraging their analysis, they implement a memory allocator optimization based on the observation that the lifetime of an object provided to client programs is correlated to its size, resulting in both execution time and memory performance improvements.

Zhou et al. [247] propose a generic memory allocator profiler aimed at detecting slowdowns incurred by the memory allocator implementation itself and identifying their causes. To this end, authors rely on one main assumption: that when the memory allocator provides user clients with one memory block, this may impact performance during the whole lifetime of this allocation. In this setting, they consider both costs associated with memory management operations or due to memory accesses, e.g., inducing various cache-related issues. All memory management operations are intercepted. All allocations addresses along with corresponding sizes are stored in a hash table and never removed, to distinguished between fresh addresses and allocations reuses. Average execution times of such operations are stored, such that small allocations are distinguished from big ones and fresh allocations from reused ones; execution time for each sort is then compared with that of a reference allocator. In turn, this information is used to maintain additional information about the utilization rate of each page used for allocations, as well as corresponding cache lines. Through hardware-based sampling of PMUs (likely Intel PEBS, although it is not explicitly mentioned), precise information about cache misses is collected. Separate information is kept for each thread using thread-local storage, and a precise report is emitted at the end of the considered program execution, diagnosing the cause of any measured slowdown. Authors report that 98% of allocators-induced slowdowns larger than 5% are identified, at the cost of a 10% overhead, identifying issues in existing allocators.

Tracing. Obtaining complete traces of memory management-related calls has the advantage of flexibility, as collected traces can be leveraged in various ways. This however comes at the cost of higher overhead. We focus here on using such traces to gain more understanding of memory allocators implementations and why specific benchmarks strain some of them.

One first direct application of complete traces is that memory management operations can be replayed to provide reproducible benchmarks. Zorn and Grunwald [248] evaluate in a sequential setting the accuracy of synthetic models of allocation behaviours with respect to the evaluation of allocators using actual traces. They conclude that randomly generated sets of events based on considered synthetic models only are suitable to evaluate simple programs or simple memory allocation algorithms.

Chilimbi, Jones, and Zorn [249] also aim at measuring the performance of allocators in a representative manner. Pointing the limits of using short-running sequential benchmarks, they call for the use of large allocation traces based on real-world multithreaded workloads. In turn, they remark that the sharing of such traces is impractical due to the lack of standardized format. To solve this, they propose a trace format to record memory management operations while compressing the corresponding trace. They evaluate variations over their design regarding trace size and required processing time.

Such comprehensive tracing can also be leveraged for offline analysis. Perks et al. [250] present WMTools, a toolkit aimed at understanding memory inefficient use by applications. To this end they implement the tracing of memory events along with the call site for allocations, collected data is compressed on-the-fly before being stored in a file. Authors insist on the flexibility left for offline analysis. In addition to that, they remark that additional runtime overhead would be incurred if the tracing tool was performing the analysis at runtime. Furthermore, authors evaluate their tool against the Massif heap profiler part and the Memcheck error detector both part of Valgrind. Corresponding evaluation demonstrates the competitiveness of WMTools with an associated overhead ranging from 1x in most cases to 12x in some cases, noting obtained results regarding Massif overhead are in line with the documentation, that indicates an overhead range from 5x to 100x.

Delorie, O'Donell, and Weimer [251] refine the `mtrace` malloc tracing tool part of the glibc and present a new tool introducing less overhead [252] that can be used to intercept all memory management calls to the libc for whole-system benchmarking. The resulting trace is a compact binary record containing allocations addresses and sizes as well as the call type and the thread id. This trace is progressively written to a file for further use at runtime. Authors leverage their tool to compare memory usage when using different memory allocators as well as tweaking memory allocators tunable parameters.

Ji et al. [253] adopt an approach that they themselves describe as "holistic". They include HPC (scientific computing) as part of a diverse set of programs to be investigated, noting that HPC applications have been less analyzed and characterized by previous work. Most importantly, their goal is to provide a fine-grained analysis of these. In particular, their study include object and lifetime distribu-

tion, the evolution of the total memory footprint, memory accesses patterns such as read-write ratio temporal and spatial locality as well as the sparsity of data. To this end, they develop a tool that implements two consecutive passes: first one is an online pass that aims at being fast, with a reported overhead ranging from 1x to 42x and a median of 1.2x. This pass intercepts every memory management operation to collect pointers as well as sizes of objects and group allocations by the stack of the caller. The second pass is an offline pass that can be very costly (reported to have an overhead up to 1000x). In particular, their goal is to map memory allocation blocks to variables in the programs source code, to help with further manual analysis. Interestingly, they state as part of their results that no strong correlation is observed between variable size and lifetime, what stands in opposition to the key observation made by Zhou et al. [246]. This suggests once more that there exists a very large set of possible behaviours for programs, depending on the function they perform.

Finally, we note that Hertz and Berger [254] leverage tracing of memory management operations to quantify the performance of precise garbage collage collection against explicit memory management.

Broader perspective on hardware tracing features. One last area of related work is other existing hardware tracing features. We remark that the Intel PT feature is not the only existing low-overhead hardware-based tracing feature. Indeed, while there does not exist equivalent hardware tracing support on AMD CPUs to the best of our knowledge, there exists other similar features on different hardware. First, this includes an IPT equivalent on ARM's side: ARM CoreSight [255]. Furthermore, Apple silicon also comes equipped with a "Processor Trace instrument" [256].

Finally, we note that the Intel PT feature has been used with the aim of achieving various goals. As a matter of fact, not exhaustively, it has been used for (reverse) debugging [257], that is, to determine the root cause of software failures; for low-overhead data race detection [258]; to improve security through the enforcement of control-flow integrity policies [259, 260, 261]; as well as for fuzzing [262, 263]. We believe this indicates the broad capabilities of such hardware-based low-overhead tracing techniques.

Chapter 6

Conclusion

In this thesis, we aimed at formally verifying hardened memory allocators so that the resulting code artifact could be used as a drop-in replacement of unverified software. On the path to this journey, we set two additional goals. On one hand, to develop a verification methodology extending Steel's methodology relying on dependent types, SMT-driven verification and modular abstractions to obtain scalable, iterative verification. On the other hand, to obtain a competitive memory allocator regarding similar hardened memory allocators.

Overview of our results. We developed StarMalloc, a formally verified hardened memory allocator inspired by the already existing security-oriented memory allocator hardened_malloc. Although StarMalloc does not exhaustively support all of hardened_malloc's security features, StarMalloc supports a representative subset of these. We note that these security mechanisms are commonly used by other hardened memory allocators [151] on top of its hardened_malloc-inspired intrinsic security-focused architecture. These security mechanisms complete our implementation of the entire expected API surface of a memory allocator, whose specifications reflect the enforcement of requirements mandated by the C standard.

As part of our verification efforts, we leveraged and extended in several ways the methodology baked in the Steel "formulation" of Concurrent Separation Logic triples through the division and specialized handling of proof obligations. First, we developed specialized abstractions in the form of combinators. Second, we carefully tailored separation logic predicates so that corresponding proof obligations were assigned to the most suitable proof "backend", leveraging separation logic predicates for ownership and SMT automation for complex reasoning on memory content. Finally, we reused existing techniques [158, 159, 160] leveraging compile-time reduction through normalization for the sake of configurability, yielding a memory allocator supporting various environments through configurable size classes and security mechanisms.

Most importantly, we note that the adopted verification methodology enabled us to scale up to the point of developing a realistic memory allocator, resulting in a C code artifact of 6.0kLoC corresponding to 42kLoC, a third of which are reusable libraries. Furthermore, its support for iterative development allowed us to optimize

StarMalloc's implementation until exhibiting competitive performance on a large set of benchmarks, using hardened_malloc as the baseline, as well as demonstrating its applicability to real-world scenarios such as its integration in the Firefox web browser.

Insights for systems verification. As Charguéraud [62] puts it, it is "hard to believe that Separation Logic has not been around [... since] program verification exists". In their retrospective paper, Brookes and O'Hearn [126] express their astonishment regarding how numerous unforeseen extensions have been developed, accounting for various concurrent programming invariants.

In our setting, using a higher-order CSL allowed us to reason on various such invariants (mutexes, frozen arrays for initialization), in addition to specify fine-grained ownership transfers as well as proving correct a disparate set of low-level data structures. All in all, joining the voices of many other, we firmly believe that CSL is well-suited to systems verification.

We lay the emphasis on the fact that the separation of proof obligations discharged by distinct specialized proof backends, e.g., the automatic frame inference tactic or the Z3 SMT solver, was of tremendous importance. Indeed, this enforces a proof discipline that can be leveraged in two ways. On one hand, properties can be encoded by targeting the most suitable backend, helping with iterative development. On the other hand, this provides the programmer with a high-level view over the current proof state as well as in the case of Steel with a twofold "sliding admit" strategy [135] acting as a double-deck sieve for proof refinements.

We think that this could be further improved by additional proof backends. Indeed, in some cases, we however faced some well-known issues related to SMT handling, e.g., instability when tackling proofs involving non-linear arithmetic. In such cases, while benefiting from automation was essential, losing fine-grained control over proof handling represented a major hurdle. This sometimes led us to very verbose, manual proofs involving opaque predicates as well as very precise NLA lemmas encoded in the form of type refinements exactly matching goals. It seems that a tactic such as the `grind` tactic provided in the Lean theorem prover [264] would have exactly met our needs.

Limitations and future work. In some cases, we observed that Steel's generation of Verification Conditions was likely suboptimal. In such settings, the SMT solver is able to successfully discharge in a few seconds at most what we assume are SMT queries that occupy several gigabytes of RAM and seemingly requiring up to minutes of CPU work for their generation. Given memory bandwidth limits, in these contexts, it seems likely that the SMT solver only performs a very limited traversal of considered queries. Although great care has been put in the design of Steel's VC generator [137], we believe this indicates it could be further improved.

Bibliography

[1] Antonin Reitz, Aymeric Fromherz, and Jonathan Protzenko. "StarMalloc: Verifying a Modern, Hardened Memory Allocator". In: *Proc. ACM Program. Lang.* OOPSLA2 (Oct. 2024). DOI: 10.1145/3689773.

[2] *StarMalloc: a formally verified, modern, hardened memory allocator*. URL: https://github.com/Inria-Prosecco/StarMalloc.

[3] *TranscrIPT: hardware-based memory management operations tracing*. URL: https://github.com/Inria-Prosecco/StarMalloc/tree/tracing/data.

[4] Jan Peleska, Anne E. Haxthausen, and Thierry Lecomte. "Standardisation Considerations for Autonomous Train Control". In: 2022, pp. 286–307. DOI: 10.1007/978-3-031-19762-8_22.

[5] *Commission Regulation (EU) 2018/401 of 14 March 2018 amending Regulation (EU) No 139/2014 as regards the classification of runways*. 2018. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32018R0401.

[6] Harshad Sathaye, Domien Schepers, Aanjhan Ranganathan, and Guevara Noubir. "Wireless Attacks on Aircraft Instrument Landing Systems". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 357–372. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/sathaye.

[7] *Regulation (EC) No 661/2009 of the European Parliament and of the Council of 13 July 2009 concerning type-approval requirements for the general safety of motor vehicles, their trailers and systems, components and separate technical units intended therefor*. 2009. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32009R0661.

[8] N. G. Leveson and C. S. Turner. "An Investigation of the Therac-25 Accidents". In: *Computer* 26.7 (1993), pp. 18–41.

[9] Mark Halper. *How Software Bugs led to 'One of the Greatest Miscarriages of Justice' in British History*. 2025. DOI: 10.1145/3703779.

[10] Doug Meil. *The U.K. Post Office Scandal: Software Malpractice At Scale*. 2024. URL: https://cacm.acm.org/blogcacm/the-u-k-post-office-scandal-software-malpractice-at-scale/.

[11] Julia Kollewe. *Post Office scandal may have led to more than 13 suicides, inquiry finds*. 2025. URL: https://www.theguardian.com/uk-news/2025/jul/08/post-office-scandal-inquiry-horizon-it-scandal.

[12] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. *Ariane 5 failure – Report by the inquiry board*. 1996. URL: https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[13] Gerard Le Lann. *The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems*. Research Report RR-3079. Projet REFLECS. French National Institute for Research in Computer Science and Control (INRIA), 1996. URL: https://inria.hal.science/inria-00073613.

[14] Matt Miller. *Trends, challenge, and strategic shifts in the software vulnerability mitigation landscape*. Microsoft Security Response Center (MSRC). 2019. URL: https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[15] The Chromium security team. *Memory safety*. 2020. URL: https://www.chromium.org/Home/chromium-security/memory-safety/.

[16] Maddie Stone, Jared Semrau, and James Sadowski. *A review of zero-day in-the-wild exploits in 2023*. Mandiant and Google's Threat Analysis Group. 2024. URL: https://blog.google/technology/safety-security/a-review-of-zero-day-in-the-wild-exploits-in-2023/.

[17] US National Vulnerability Database. *NVD – CVE-2014-0160*. Heartbleed bug. 2014. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-0160.

[18] US National Vulnerability Database. *NVD – CVE 2021-22156*. BadAlloc bug. 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-22156.

[19] Sam Frizell. *Report: Devastating Heartbleed Flaw Was Used in Hospital Hack*. 2014. URL: https://time.com/3148773/report-devastating-heartbleed-flaw-was-used-in-hospital-hack/.

[20] US Cybersecurity & Infrastructure Security Agency (CISA). *Multiple RTOS (Update E) | CISA*. List of RTOSs, SDKs and libraries affected by BadaAlloc. 2021. URL: https://www.cisa.gov/news-events/ics-advisories/icsa-21-119-04.

[21] Microsoft Security Response Center (MSRC). *"BadAlloc" – Memory allocation vulnerabilities could affect wide range of IoT and OT devices in industrial, medical, and enterprise networks*. 2021. URL: https://msrc.microsoft.com/blog/2021/04/badalloc-memory-allocation-vulnerabilities-could-affect-wide-range-of-iot-and-ot-devices-in-industrial-medical-and-enterprise-networks/.

[22] Jean-Yves Marion. "Ransomware: Extortion Is My Business". In: *Commun. ACM* 68.5 (Apr. 2025), pp. 36–47. ISSN: 0001-0782. DOI: 10.1145/3697829.

[23] Andy Grenberg. *The Untold Story of NotPetya, the Most Devastating Cyberattack in History | WIRED*. 2018. URL: https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/.

[24] US National Vulnerability Database. *NVD – CVE 2017-0143*. EternalBlue bug. 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-0143.

[25] Ali Islam, Nicole Oppenheim, and Winny Thomas. *SMB Exploited: WannaCry Use of "EternalBlue"*. Mandiant. 2017. URL: https://cloud.google.com/blog/topics/threat-intelligence/smb-exploited-wannacry-use-of-eternalblue/.

[26] The US White House. *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. 2024. URL: https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.

[27] US National Security Agency. *Software Memory Safety*. 2022. URL: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.

[28] US Cybersecurity & Infrastructure Security Agency (CISA). *Secure by Design. Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software*. 2023. URL: https://www.cisa.gov/sites/default/files/2023-10/SecureByDesign_1025_508c.pdf.

[29] US Cybersecurity & Infrastructure Security Agency (CISA). *The Case for Memory Safe Roadmaps*. 2023. URL: https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf.

[30] US Cybersecurity & Infrastructure Security Agency (CISA). *Exploring Memory Safety in Critical Open Source Projects*. 2024. URL: https://www.cisa.gov/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf.

[31] Alex Rebert and Christoph Kern. *Secure by Design: Google's Perspective on Memory Safety*. Google Security Engineering Technical Report. 2024. URL: https://storage.googleapis.com/gweb-research2023-media/pubtools/7665.pdf.

[32] Robert N.M. Watson, John Baldwin, David Chisnall, Tony Chen, Jessica Clarke, Brooks Davis, Nathaniel Filardo, Brett Gutstein, Graeme Jenkinson, Ben Laurie, Alfredo Mazzinghi, Simon Moore, Peter G. Neumann, Hamed Okhravi, Alex Richardson, Alex Rebert, Peter Sewell, Laurence Tratt, Murali Vijayaraghavan, Hugo Vincent, and Konrad Witaszczyk. "It Is Time to Standardize Principles and Practices for Software Memory Safety". In: *Commun. ACM* 68.2 (Jan. 2025), pp. 40–45. ISSN: 0001-0782. DOI: 10.1145/3708553.

[33] Jeff Vander Stoep and Alex Rebert. *Eliminating Memory Safety Vulnerabilities at the Source*. Google Online Security Blog. 2024. URL: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html.

[34] French National Agency for the Security of Information Systems (ANSSI). *Programming Rules to Develop Secure Applications with Rust – v1.0*. 2020. URL: https://cyber.gouv.fr/sites/default/files/2020/06/anssi-guide-programming_rules_to_develop_secure_applications_with_rust-v1.0.pdf.

[35] *Memory Safety in Ada, SPARK and Rust*. URL: https://www.adacore.com/uploads/techPapers/Memory-Safety-in-Ada-SPARK-and-Rust-V3.pdf.

[36] US National Institute of Standards and Technology (NIST). *Safer Langages | NIST*. URL: https://www.nist.gov/itl/ssd/software-quality-group/safer-languages.

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2013.

[38] US Defense Advanced Research Projects Agency (DARPA). *Eliminating Memory Safety Vulnerabilities Once and For All*. 2024. URL: https://www.darpa.mil/news/2024/memory-safety-vulnerabilities.

[39] Aymeric Fromherz and Jonathan Protzenko. *Compiling C to Safe Rust, Formalized*. 2024. arXiv: 2412.15042 [cs.PL]. URL: https://arxiv.org/abs/2412.15042.

[40] Emery D. Berger. "Software Needs Seatbelts and Airbags: Finding and Fixing Bugs in Deployed Software is Difficult and Time-Consuming. Here Are Some Alternatives." In: *Communications of the ACM* (2012).

[41] *Towards the next generation of XNU memory safety: kalloc_type*. Apple Security Research. 2022. URL: https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/.

[42] Apple Security Engineering and Architecture (SEAR). *Memory Integrity Enforcement: A complete vision for memory safety in Apple devices*. Apple Security Research. 2025. URL: https://security.apple.com/blog/memory-integrity-enforcement/.

[43] Robert N. M. Watson, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Ben Laurie, Simon W. Moore, Peter G. Neumann, Alexander Richardson, Peter Sewell, Konrad Witaszczyk, and Jonathan Woodruff. "CHERI: Hardware-Enabled C/C++ Memory Protection at Scale". In: *IEEE Security & Privacy* 22.4 (2024), pp. 50–61. DOI: 10.1109/MSEC.2024.3396701.

[44] Edmund M. Clarke and Jeannette M. Wing. "Formal methods: state of the art and future directions". In: *ACM Comput. Surv.* 28.4 (Dec. 1996), pp. 626–643. ISSN: 0360-0300. DOI: 10.1145/242223.242257.

[45] European Union Aviation Safety Agency. *Harmonised Software EASA AMC and FAA AC 20-115D have been published! | EASA*. 2017. URL: https://www.easa.europa.eu/en/newsroom-and-events/news/harmonised-software-easa-amc-and-faa-ac-20-115d-have-been-published.

[46] US Federal Aviation Administration. *AC 20-115D - Airborne Software Development Assurance Using EUROCAE ED-12( ) and RTCA DO-178( ).* 2017. URL: https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/1032046.

[47] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[48] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* 3rd edition. Pearson, 2015. ISBN: 013409266X.

[49] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. "Dynamic storage allocation: A survey and critical review". In: *Memory Management.* Ed. by Henry G. Baler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116. ISBN: 978-3-540-45511-0. DOI: 10.1007/3-540-60368-9_19.

[50] Igor Zhirkov. *Low-Level Programming: C, Assembly, and Program Execution on Intel 64 Architecture.* 1st edition. USA: Apress, 2017. ISBN: 1484224027.

[51] Jason Evans. "A scalable concurrent malloc (3) implementation for FreeBSD". In: *BSDCan — The Technical BSD Conference.* 2006.

[52] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. "Hoard: a scalable memory allocator for multithreaded applications". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS IX. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2000, pp. 117–128. ISBN: 1581133170. DOI: 10.1145/378993.379232. URL: https://doi.org/10.1145/378993.379232.

[53] Adrian Taylor, Bartek Nowierski, and Kentaro Hara. *Use-after-freedom: MiraclePtr.* Google Online Security Blog. 2022. URL: https://security.googleblog.com/2022/09/use-after-freedom-miracleptr.html.

[54] A. Turing. "Checking a large routine". In: *The Early British Computer Conferences (1989).* Cambridge, MA, USA: MIT Press, 1949, pp. 70–72. ISBN: 0262231360.

[55] Herman H. Goldstein and John von Neumann. "Planning and coding problems for an electronic computine instrument. part 2". In: *Collected Works* 5 (1947), pp. 80–151.

[56] Haskell B. Curry. *On the composition of programs for automatic computing.* 1949.

[57] Donald E. Knuth. *Robert W Floyd, in Memoriam.* Ed. by Thomas Haigh. DOI: 10.1109/MAHC.2004.1299661.

[58] Robert W Floyd. "Assigning meanings to programs". In: *Program Verification: Fundamental Issues in Computer Science.* Springer, 1967, pp. 65–81.

[59] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[60] P. J. Landin. "Correspondence between ALGOL 60 and Church's Lambda-notation: part I". In: *Commun. ACM* 8.2 (Feb. 1965), pp. 89–101. ISSN: 0001-0782. DOI: 10.1145/363744.363749.

[61] John C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002. DOI: 10.1109/LICS.2002.1029817.

[62] Arthur Charguéraud. "Separation Logic for sequential programs (Functional Pearl)". In: *Proc. ACM Program. Lang.* 4.ICFP (2020). DOI: 10.1145/3408998.

[63] Peter W. O'Hearn. "Resources, Concurrency, and Local Reasoning". In: *Theoretical Computer Science (TCS)*. 2007.

[64] Stephen Brookes. "A Semantics for Concurrent Separation Logic". In: *Theoretical Computer Science (TCS)*. 2007.

[65] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[66] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A static analyzer for large safety-critical software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI '03. San Diego, California, USA: Association for Computing Machinery, 2003, pp. 196–207. ISBN: 1581136625. DOI: 10.1145/781131.781153. URL: https://doi.org/10.1145/781131.781153.

[67] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008). ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: https://doi.org/10.1145/1347375.1347389.

[68] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7*. Oct. 2017. URL: http://coq.inria.fr.

[69] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. "A Metaprogramming Framework for Formal Verification". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2017. DOI: 10.1145/3110278.

[70] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[71] 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 Symposium on Operating Systems Principles (SOSP)*. 2009.

[72] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels". In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016.

[73] Xavier Leroy. "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2006.

[74] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2014.

[75] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A Formally-Verified C Static Analyzer". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2015.

[76] Diomidis Spinellis. *Continuous Unix commit history from 1970 until today*. URL: `https://github.com/dspinellis/unix-history-repo/blob/Research-V4-Snapshot-Development/sys/dmr/malloc.c`.

[77] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd edition. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[78] International Organization for Standardization. *C89 – ISO/IEC 9899:1990 - Information Technology–Programming languages–C*. 1990.

[79] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. "Towards optimization-safe systems: analyzing the impact of undefined behavior". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 260–275. ISBN: 9781450323888. DOI: 10.1145/2517349.2522728. URL: https://doi.org/10.1145/2517349.2522728.

[80] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834.

[81] International Organization for Standardization. *C99 – ISO/IEC 9899:1999 - Information Technology–Programming languages–C*. 1999.

[82] International Organization for Standardization. *C11 – ISO/IEC 9899:2011 - Information Technology–Programming languages–C*. 2011.

[83] International Organization for Standardization. *C17 – ISO/IEC 9899:2018 - Information Technology–Programming languages–C*. 2018.

[84] International Organization for Standardization. *C23 – ISO/IEC 9899:2024 - Information Technology–Programming languages–C*. 2024.

[85] Nick Stoughton. *DR 400 – `realloc` with size zero problems*. 2011. URL: https://open-std.org/JTC1/SC22/WG14/www/docs/n2396.htm#dr_400.

[86] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010. ISBN: 1593272200.

[87] David Goldblatt, Chris Kennelly, and Mark Santaniello. *N2699 – Sized Memory Deallocation*. 2021. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2699.htm.

[88] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. "POSIX abstractions in modern operating systems: the old, the new, and the missing". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 9781450342407. DOI: 10.1145/2901318.2901350.

[89] "IEEE/Open Group Standard for Information Technology–Portable Operating System Interface (POSIX™) Base Specifications, Issue 8". In: *IEEE/Open Group Std 1003.1-2024 (Revision of IEEE Std 1003.1-2017)* (2024), pp. 1–4107. DOI: 10.1109/IEEESTD.2024.10555529.

[90] *Building Linux with Clang/LLVM – The Linux Kernel documentation*. URL: https://docs.kernel.org/kbuild/llvm.html.

[91] *musl libc*. URL: https://musl.libc.org/.

[92] *`malloc_usable_size` (3) - Linux manual page*. 2025. URL: https://www.man7.org/linux/man-pages/man3/malloc_usable_size.3.html.

[93] *Replacing malloc (The GNU C Library)*. URL: https://sourceware.org/glibc/manual/2.42/html_node/Replacing-malloc.html.

[94] Per-Åke Larson and Murali Krishnan. "Memory allocation for long-running server applications". In: *Proceedings of the 1st International Symposium on Memory Management*. ISMM '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 176–185. ISBN: 1581131143. DOI: 10.1145/286860.286880. URL: https://doi.org/10.1145/286860.286880.

[95] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. "Snmalloc: A Message Passing Allocator". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 122–135. ISBN: 9781450367226. DOI: 10.1145/3315573.3329980. URL: https://doi.org/10.1145/3315573.3329980.

[96] Dave Dice and Alex Garthwaite. "Mostly lock-free malloc". In: *Proceedings of the 3rd International Symposium on Memory Management*. ISMM '02. Berlin, Germany: Association for Computing Machinery, 2002, pp. 163–174. ISBN: 1581135394. DOI: 10.1145/512429.512451. URL: https://doi.org/10.1145/512429.512451.

[97] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. "Protego: Overload Control for Applications with Unpredictable Lock Contention". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 725–738. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/cho-inho.

[98] Insu Yun, Dhaval Kapil, and Taesoo Kim. "Automatic Techniques to Systematically Discover New Heap Exploitation Primitives". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1111–1128. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/yun.

[99] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. "Mimalloc: Free list sharding in action". In: *Proceedings of the Asian Conference on Programming Languages and Systems (ASPLAS)*. 2019. DOI: 10.1007/978-3-030-34175-6_13.

[100] GrapheneOS Development Team. *hardened_malloc – GrapheneOS's hardened memory allocator*. https://github.com/GrapheneOS/hardened_malloc/blob/main/README.md#security-properties. 2021.

[101] Haroon Meer. *Memory Corruption Attacks: The (almost) Complete History*. 2010. URL: https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf.

[102] *fastbin_dup_into_stack.c – Educational Heap Exploitation*. URL: https://github.com/shellphish/how2heap/blob/master/glibc_2.35/fastbin_dup_into_stack.c.

[103] David Edward Gay. "Memory Management with Explicit Regions". Available at https://theory.stanford.edu/~aiken/publications/theses/gay.pdf. PhD thesis. University of California at Berkeley, 2001.

[104] *CWE-122: Heap-based Buffer Overflow (4.18)*. URL: https://cwe.mitre.org/data/definitions/122.html.

[105] *CWE-126: Buffer Over-read (4.18)*. URL: https://cwe.mitre.org/data/definitions/126.html.

[106] *CWE-416: Use After Free (4.18)*. URL: https://cwe.mitre.org/data/definitions/416.html.

[107] *CWE-415: Double Free (4.18)*. URL: https://cwe.mitre.org/data/definitions/415.html.

[108] *CWE-590: Free of Memory not on the Heap (4.18)*. URL: https://cwe.mitre.org/data/definitions/590.html.

[109] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. "PTAuth: Temporal Memory Safety via Robust Points-to Authentication". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1037–1054. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[110] *CWE-457: Use of Uninitialized Variable (4.18)*. URL: https://cwe.mitre.org/data/definitions/457.html.

[111] *CWE-908: Use of Uninitialized Resource (4.18)*. URL: https://cwe.mitre.org/data/definitions/908.html.

[112] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. "Type-After-Type: Practical and Complete Type-Safe Memory Reuse". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 17–27. ISBN: 9781450365697. DOI: 10.1145/3274694.3274705. URL: https://doi.org/10.1145/3274694.3274705.

[113] The Shellphish Team. *Educational Heap Exploitation*. https://github.com/shellphish/how2heap. 2023.

[114] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks". In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, p. 5.

[115] Marius Momeu, Simon Schnückel, Kai Angnis, Michalis Polychronakis, and Vasileios P. Kemerlis. "Safeslab: Mitigating Use-After-Free Vulnerabilities via Memory Protection Keys". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, pp. 1345–1359. ISBN: 9798400706363. DOI: 10.1145/3658644.3670279. URL: https://doi.org/10.1145/3658644.3670279.

[116] Emery D. Berger and Benjamin G. Zorn. "DieHard: Probabilistic Memory Safety for Unsafe Languages". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2006. DOI: 10.1145/1133255.1134000.

[117] Gene Novark and Emery D Berger. "DieHarder: securing the heap". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2010.

[118] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic*. 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1.

[119] Samin S. Ishtiaq and Peter W. O'Hearn. "BI as an assertion language for mutable data structures". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'01. 2001, pp. 14–26. DOI: 10.1145/360204.375719.

[120] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[121] Krzysztof R. Apt. "Ten Years of Hoare's Logic: A Survey—Part I". In: *ACM Trans. Program. Lang. Syst.* 3.4 (Oct. 1981), pp. 431–483. ISSN: 0164-0925. DOI: 10.1145/357146.357150. URL: https://doi.org/10.1145/357146.357150.

[122] Susan Owicki and David Gries. "Verifying properties of parallel programs: an axiomatic approach". In: *Commun. ACM* 19.5 (May 1976), pp. 279–285. ISSN: 0001-0782. DOI: 10.1145/360051.360224. URL: https://doi.org/10.1145/360051.360224.

[123] Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[124] David Park. "On the Semantics of Fair Parallelism". In: *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*. Berlin, Heidelberg: Springer-Verlag, 1979, pp. 504–526. ISBN: 3540100075.

[125] James H. Anderson and Mohamed G. Gouda. *A Criterion for Atomicity*. Tech. rep. USA, 1990.

[126] Stephen Brookes and Peter W. O'Hearn. "Concurrent separation logic". In: *ACM SIGLOG News* 3.3 (Aug. 2016), pp. 47–65. DOI: 10.1145/2984450.2984457. URL: https://doi.org/10.1145/2984450.2984457.

[127] K. Rustan M. Leino and Michał Moskal. "Usable auto-active verification". In: *Usable Verification Workshop*. 2010. URL: https://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf.

[128] *F⋆: A Proof-Oriented Programming Language*. URL: https://fstar-lang.org.

[129] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. "Dependent types and multi-monadic effects in F⋆ ". In: POPL'16 (2016), pp. 256–270. DOI: 10.1145/2914770.2837655.

[130] Nikhil Swamy, Guido Martínez, and Aseem Rastogi. *Proof-Oriented Programming in F⋆*. 2025. URL: https://fstar-lang.org/tutorial/proof-oriented-programming-in-fstar.pdf.

[131] *FStarLang/fstar-mode.el: Emacs support for F⋆*. URL: https://github.com/FStarLang/fstar-mode.el.

[132] *FStarLang/fstar-vscode-assistant: An interactive mode for F⋆ in VS Code.* URL: https://github.com/FStarLang/fstar-vscode-assistant.

[133] Théophile Wallez. *TWal/fstar.nvim: Neovim mode for F⋆.* URL: https://github.com/TWal/fstar.nvim.

[134] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. "Mariposa: Measuring SMT Instability in Automated Program Verification". In: *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 2023.

[135] *Sliding admit verification style.* 2020. URL: https://github.com/FStarLang/FStar/wiki/Sliding-admit-verification-style.

[136] *F⋆ standard library's FStar.Ghost module.* 2023. URL: https://github.com/FStarLang/FStar/blob/master/ulib/FStar.Ghost.fsti.

[137] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. "Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2021. DOI: 10.1145/3473590.

[138] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. "SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2020. DOI: 10.1145/3409003.

[139] *Steel.Reference module.* URL: https://github.com/FStarLang/steel/blob/main/lib/steel/Steel.Reference.fsti.

[140] *Steel.SpinLock module.* URL: https://github.com/FStarLang/steel/blob/main/lib/steel/Steel.SpinLock.fsti.

[141] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. "Verified Low-Level Programming Embedded in F*". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2017.

[142] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. "HACLxN: Verified Generic SIMD Crypto (for All Your Favourite Platforms)". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2020.

[143] *FStarLang/KaRaMeL: KaRaMeL is a tool for extracting low-level F⋆ programs to readable C code.* URL: https://github.com/FStarLang/karamel.

[144] *Steel.Array module.* URL: https://github.com/FStarLang/steel/blob/main/lib/steel/Steel.Array.fsti.

[145] Otto Moerbeek. *A new malloc(3) for OpenBSD.* 2009. URL: https://www.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf.

[146] *GrapheneOS: the private and secure mobile OS*. URL: https://grapheneos.org/.

[147] Jonathan Corbet. *The ongoing search for mmap_lock scalability*. URL: https://lwn.net/Articles/893906/.

[148] Jeff Bonwick. "The slab allocator: an object-caching kernel memory allocator". In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. USTC'94. Boston, Massachusetts: USENIX Association, 1994, p. 6.

[149] Vlastimil Babka. *The slab allocators of past, present, and future*. 2022.

[150] Otto Moerbeek. *How OpenBSD's malloc helps the developer*. 2023. URL: https://www.openbsd.org/papers/eurobsdcon2023-otto-malloc.pdf.

[151] Insu Yun, Woosun Song, Seunggi Min, and Taesoo Kim. "HardsHeap: a universal and extensible framework for evaluating secure allocators". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2021.

[152] Apple Security Research. *Towards the next generation of XNU memory safety: kalloc_type*. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/. 2022.

[153] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution". In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2016. DOI: 10.1007/978-3-319-41528-4_22.

[154] Christopher Pulte, Dhruv C Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. "CN: Verifying Systems C Code with Separation-Logic Refinement Types". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2023.

[155] Kiran Gopinathan, Mayank Keoliya, and Ilya Sergey. "Mostly Automated Proof Repair for Verified Libraries". In: (2023). DOI: 10.1145/3591221.

[156] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. "Adapting proof automation to adapt proofs". In: *Proceedings of the International Conference on Certified Programs and Proofs (CPP)*. 2018. DOI: 10.1145/3167094.

[157] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. "QED at large: A survey of engineering of formally verified software". In: *Foundations and Trends® in Programming Languages* 5.2-3 (2019), pp. 102–281.

[158] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. "Noise*: A library of verified high-performance secure channel protocol implementations". In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2022.

[159] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. "Hardening attack surfaces with formally proven binary format parsers". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2022.

[160] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. "Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification". In: *Proceedings of the International Conference on Functional Programming (ICFP)*. 2023. DOI: 10.1145/3607844.

[161] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. "Provably-Safe Multilingual Software Sandboxing using WebAssembly". In: *Proceedings of the USENIX Security Symposium*. 2022.

[162] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. "Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3655–3672. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao.

[163] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. "Dependent Types and Multi-Monadic Effects in F*". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2016.

[164] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008.

[165] Rich Felker. *Comparison between hardened_malloc and musl mallocs*. https://www.openwall.com/lists/musl/2020/05/13/1. 2020.

[166] LLVM Project. *Scudo Hardened Allocator*. https://llvm.org/docs/ScudoHardenedAllocato
html. 2023.

[167] Daan Leijen, Julien Voisin, and Matthew Parkinson. *mimalloc-bench – Suite for benchmarking malloc implementation*. https://github.com/daanx/mimalloc-bench. 2018.

[168] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. "FreeGuard: A Faster Secure Heap Allocator". In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2017.

[169] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. "Guarder: A Tunable Secure Allocator." In: *Proceedings of the USENIX Security Symposium*. 2018.

[170] Jacob Bramley, Dejice Jacob, Andrei Lascu, Jeremy Singer, and Laurence Tratt. "Picking a CHERI Allocator: Security and Performance Considerations". In: *Proceedings of the International Symposium on Memory Management (ISMM)*. 2023.

[171] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. "CAMP: Compiler and Allocator-based Heap Memory Protection". In: *Proceedings of the USENIX Security Symposium*. 2024.

[172] Márton Erdős, Sam Ainsworth, and Timothy M Jones. "MineSweeper: a "clean sweep" for drop-in use-after-free prevention". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2022.

[173] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. "NextGen-Malloc: Giving Memory Allocator Its Own Room in the House". In: *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 2023.

[174] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. "Preventing Use-After-Free Attacks with Fast Forward Allocation". In: *Proceedings of the USENIX Security Symposium*. 2021.

[175] Chris Rohlf. *Isolation Alloc*. `https://struct.github.io/iso_alloc.html`. 2020.

[176] Beichen Liu, Pierre Olivier, and Binoy Ravindran. "Slimguard: A secure and memory-efficient heap allocator". In: *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*. 2019.

[177] Mike Hommey. *Hooking the memory allocator in Firefox*. `https://glandium.org/blog/?p=2848`. 2012.

[178] The Mozilla development team. *mozjemalloc*. URL: `https://searchfox.org/firefox-main/source/memory/build/README`.

[179] The Chromium development team. *PartitionAlloc design*. URL: `https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md`.

[180] Periklis Akritidis. "Cling: A Memory Allocator to Mitigate Dangling Pointers." In: *Proceedings of the USENIX Security Symposium*. 2010.

[181] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. "Mesh: compacting memory management for C/C++ applications". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 333–346. ISBN: 9781450367127. DOI: `10.1145/3314221.3314582`. URL: `https://doi.org/10.1145/3314221.3314582`.

[182] JetStream2. *JetStream 2.1 In-Depth Analysis*. `https://browserbench.org/JetStream/in-depth.html`. 2022.

[183] Daiping Liu, Mingwei Zhang, and Haining Wang. "A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1635–1648. ISBN: 9781450356930. DOI: 10.1145/3243734.3243826. URL: https://doi.org/10.1145/3243734.3243826.

[184] *Bug 1805644 – Speedometer 2 is 5% faster with –disable-jemalloc*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1805644.

[185] Mike Hommey. *Logging Firefox memory allocations*. 2014. URL: https://glandium.org/blog/?p=3337.

[186] Amitabh Srivastava and Alan Eustace. "ATOM: a system for building customized program analysis tools". In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI '94. Orlando, Florida, USA, 1994, pp. 196–205. DOI: 10.1145/178243.178260.

[187] Robert Bernecky. "Profiling, performance, and perfection (tutorial session)". In: *Proceedings of the ACM/SIGAPL Conference on APL as a Tool of Thought (Session Tutorials)*. APL '89. New York, New York, USA: Association for Computing Machinery, 1989, pp. 31–52. ISBN: 0897913310. DOI: 10.1145/328877.328879.

[188] S. Gill. "The Diagnosis of Mistakes in Programmes on the EDSAC". In: *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* 206.1087 (1951), pp. 538–554. DOI: 10.1098/rspa.1951.0087.

[189] *Skylake bug: a detective story*. 2017. URL: https://tech.ahrefs.com/skylake-bug-a-detective-story-ab1ad2beddcd.

[190] *Pentium FDIV flaw FAQ*. 2011. URL: https://web.archive.org/web/20191126144803/http://www.trnicely.net/pentbug/pentbug.html.

[191] Terje Mathisen. *Pentium Secrets*. 1999. URL: https://archive.gamedev.net/archive/reference/articles/article213.html.

[192] Nishad Herath and Anders Fogh. *These are Not Your Grand Daddy's CPU Performance Counters – CPU Hardware Performance Counters for Security*. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf.

[193] Vincent M. Weaver. *Advanced Hardware Profiling and Sampling (PEBS, IBS, etc.): Creating a New PAPI Sampling Interface*. 2016. URL: https://web.eece.maine.edu/~vweaver/projects/perf_events/sampling/pebs_ibs_sampling.pdf.

[194] Muhammad Aditya Sasongko, Milind Chabbi, Paul H J Kelly, and Didem Unat. "Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison". In: *IEEE Transactions on Parallel and Distributed Systems* 34.5 (2023), pp. 1594–1608. DOI: 10.1109/TPDS.2023.3257105.

[195] Denis Bakhvalov. *Performance analysis vocabulary*. 2018. URL: https://easyperf.net/blog/2018/09/04/Performance-Analysis-Vocabulary.

[196] Denis Bakhvalov. *PMU counters and profiling basics*. 2018. URL: https://easyperf.net/blog/2018/06/01/PMU-counters-and-profiling-basics.

[197] Denis Bakhvalov. *Advanced profiling topics. PEBS and LBR*. 2018. URL: https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR.

[198] Denis Bakhvalov. *Understanding performance events skid*. 2018. URL: https://easyperf.net/blog/2018/08/29/Understanding-performance-events-skid.

[199] Andi Kleen. *An introduction to last branch records*. 2016. URL: https://lwn.net/Articles/680985/.

[200] Andi Kleen. *Advanced usage of last branch records*. 2016. URL: https://lwn.net/Articles/680996/.

[201] *Which Intel® Processor Models Support Intel® Processor Trace (Intel® PT)?* 2021. URL: https://www.intel.com/content/www/us/en/support/articles/000056730/processors.html.

[202] *perf-intel-pt(1) — Linux manual page*. 2024. URL: https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html.

[203] Denis Bakhvalov. *Enhance performance analysis with Intel Processor Trace*. 2019. URL: https://easyperf.net/blog/2019/08/23/Intel-Processor-Trace.

[204] Tristan Hume. *Magic-trace: Diagnosing tricky performance issues easily with Intel Processor Trace*. 2022. URL: https://blog.janestreet.com/magic-trace/.

[205] *perf: Linux profiling with performance counters*. URL: https://perfwiki.github.io/main/.

[206] Brendan Gregg. *Linux perf examples*. 2020. URL: https://www.brendangregg.com/perf.html.

[207] Daming D. Chen, Wen Shih Lim, Mohammad Bakhshalipour, Phillip B. Gibbons, James C. Hoe, and Bryan Parno. "HerQules: securing programs via hardware-enforced message queues". In: ASPLOS '21. Virtual, USA, 2021. DOI: 10.1145/3445814.3446736.

[208] Denis Bakhvalov. *Intel Processor Trace Part2. Better debugging experience*. 2019. URL: https://easyperf.net/blog/2019/08/30/Intel-PT-part2.

[209] *Intel® Architecture Instruction Set Extensions Programming Reference - v53, June 2024, 319433-053*. 2024. URL: https://www.intel.com/content/www/us/en/content-details/826290/intel-architecture-instruction-set-extensions-programming-reference.html.

[210] Alexander Popov. *Kernel-hack-drill and a new approach to exploiting CVE-2024-50264 in the Linux kernel*. 2025. URL: https://a13xp0p0v.github.io/2025/09/02/kernel-hack-drill-and-CVE-2024-50264.html.

[211] John Wickerson, Mike Dodds, and Matthew Parkinson. "Explicit stabilisation for modular rely-guarantee reasoning". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2010.

[212] Cliff B. Jones. "Tentative steps toward a development method for interfering programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 1983.

[213] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[214] Arash Sahebolamri, Scott D Constable, and Steve J Chapin. "A Formally Verified Heap Allocator". In: *Electrical Engineering and Computer Science - Technical Reports* (2018). https://surface.syr.edu/eecs_techreports/182.

[215] David Greenaway, June Andronick, and Gerwin Klein. "Bridging the gap: Automatic verified abstraction of C". In: *Proceedings of the International Conference on Interactive Theorem Proving (ITP)*. Springer. 2012, pp. 99–115.

[216] Andrew W Appel and David A Naumann. "Verified sequential malloc/free". In: *Proceedings of the International Symposium on Memory Management (ISMM)*. 2020. DOI: 10.1145/3381898.3397211.

[217] Andrew W. Appel. "Verified Software Toolchain". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2011. DOI: 10.1007/978-3-642-19718-5_1.

[218] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types". In: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. 2021.

[219] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2015. DOI: 10.1145/2775051.2676980.

[220] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2017. DOI: 10.1007/978-3-662-54434-1_26.

[221] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018). DOI: 10.1017/S0956796818000151.

[222] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. "Verus: A Practical Foundation for Systems Verification". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pp. 438–454. ISBN: 9798400712517. DOI: 10.1145/3694715.3695952. URL: https://doi.org/10.1145/3694715.3695952.

[223] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Verus: Verifying Rust Programs using Linear Ghost Types". In: *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 2023. DOI: 10.1145/3586037.

[224] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 2010.

[225] Jialin Li, Andrea Lattuada, Yi Zhou, Jonathan Cameron, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Linear types for large-scale systems verification". In: *Proc. ACM Program. Lang.* OOPSLA1 (Apr. 2022). DOI: 10.1145/3527313. URL: https://doi.org/10.1145/3527313.

[226] Bin Fang, Mihaela Sighireanu, Geguang Pu, Wen Su, Jean-Raymond Abrial, Mengfei Yang, and Lei Qiao. "Formal modelling of list based dynamic memory allocators". In: *Science China Information Sciences* 61 (2018), pp. 1–16.

[227] Dachuan Yu, Nadeem A Hamid, and Zhong Shao. "Building certified libraries for PCC: Dynamic storage allocation". In: *Proceedings of the European Conference on Programming Languages and Systems (ESOP)*. 2003.

[228] Yu Zhang, Yongwang Zhao, David Sanan, Lei Qiao, and Jinkun Zhang. "A verified specification of TLSF memory management allocator using state monads". In: *Proceedings of the International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA)*. 2019.

[229] Nicolas Marti, Reynald Affeldt, Akinori Yonezawa, and Department Of Computer Science. "Formal Verification of the Heap Manager of an Operating System using Separation Logic". In: *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*. 2006.

[230] Harvey Tuch, Gerwin Klein, and Michael Norrish. "Types, Bytes, and Separation Logic". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 2007.

[231] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. "Formal verification of a memory allocation module of Contiki with Frama-C: a case study". In: *Proceedings of the International Conference on Risks and Security of Internet and Systems (CRiSIS)*. 2016.

[232] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. "For a microkernel, a big lock is fine". In: *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*. 2015.

[233] Damien Doligez and Xavier Leroy. "A concurrent, generational garbage collector for a multithreaded implementation of ML". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 1993.

[234] Damien Doligez and Georges Gonthier. "Portable, unobtrusive garbage collection for multiprocessor systems". In: *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 1994.

[235] Adam Sandberg Ericsson, Magnus O Myreen, and Johannes Åman Pohjola. "A verified generational garbage collector for CakeML". In: *Journal of Automated Reasoning* 63 (2019), pp. 463–488.

[236] Yannick Zakowski, David Cachera, Delphine Demange, Gustavo Petri, David Pichardie, Suresh Jagannathan, and Jan Vitek. "Verifying a concurrent garbage collector with a rely-guarantee methodology". In: *Journal of Automated Reasoning* 63 (2019), pp. 489–515.

[237] Sheera Shamsu, Dipesh Kafle, Dhruv Maroo, Kartik Nagar, Karthikeyan Bhargavan, and KC Sivaramakrishnan. "A Mechanically Verified Garbage Collector for OCaml: A Mechanically Verified Garbage Collector for OCaml". In: *J. Autom. Reason.* 69.2 (May 2025). ISSN: 0168-7433. DOI: 10.1007/s10817-025-09721-0. URL: https://doi.org/10.1007/s10817-025-09721-0.

[238] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. "PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs". In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025). DOI: 10.1145/3729311. URL: https://doi.org/10.1145/3729311.

[239] Karl Cronburg and Samuel Z Guyer. "Floorplan: spatial layout in memory management systems". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2019, pp. 81–93.

[240] Son Ho and Jonathan Protzenko. "Aeneas: Rust verification by functional translation". In: *Proceedings of the ACM on Programming Languages* 6.ICFP (2022), pp. 711–741. DOI: 10.1145/3547647.

[241] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. "Efficiently and precisely locating memory leaks and bloat". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: Association for Computing Machinery, 2009, pp. 397–407. ISBN: 9781605583921. DOI: 10.1145/1542476.1542521. URL: https://doi.org/10.1145/1542476.1542521.

[242] Thomas Ball and James R. Larus. "Optimally profiling and tracing programs". In: *ACM Trans. Program. Lang. Syst.* 16.4 (July 1994), pp. 1319–1360. ISSN: 0164-0925. DOI: 10.1145/183432.183527. URL: https://doi.org/10.1145/183432.183527.

[243] Valgrind Developers. *Massif: a heap profiler*. URL: https://valgrind.org/docs/manual/ms-manual.html.

[244] Ziyang Xu, Yebin Chon, Yian Su, Zujun Tan, Sotiris Apostolakis, Simone Campanoni, and David I. August. "PROMPT: A Fast and Extensible Memory Profiling Framework". In: *Proc. ACM Program. Lang.* OOPSLA1 (Apr. 2024). DOI: 10.1145/3649827. URL: https://doi.org/10.1145/3649827.

[245] Bobby Powers. *mstat: a fine-grained, cgroup-based tool for profiling memory usage over time of a process tree*. URL: https://github.com/bpowers/mstat.

[246] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. "Characterizing a Memory Allocator at Warehouse Scale". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 192–206. ISBN: 9798400703867. DOI: 10.1145/3620666.3651350. URL: https://doi.org/10.1145/3620666.3651350.

[247] Jin Zhou, Sam Silvestro, Steven (Jiaxun) Tang, Hanmei Yang, Hongyu Liu, Guangming Zeng, Bo Wu, Cong Liu, and Tongping Liu. "MemPerf: Profiling Allocator-Induced Performance Slowdowns". In: *Proc. ACM Program. Lang.* OOPSLA2 (Oct. 2023). DOI: 10.1145/3622848. URL: https://doi.org/10.1145/3622848.

[248] Benjamin Zorn and Dirk Grunwald. "Evaluating models of memory allocation". In: *ACM Trans. Model. Comput. Simul.* 4.1 (Jan. 1994), pp. 107–131. ISSN: 1049-3301. DOI: 10.1145/174619.174624. URL: https://doi.org/10.1145/174619.174624.

[249] Trishul Chilimbi, Richard Jones, and Benjamin Zorn. "Designing a trace format for heap allocation events". In: *Proceedings of the 2nd International Symposium on Memory Management*. ISMM '00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 35–49. ISBN: 1581132638. DOI: 10.1145/362422.362435. URL: https://doi.org/10.1145/362422.362435.

[250] Oliver Perks, Simon D. Hammond, Simon J. Pennycook, and Stephen A. Jarvis. "WMTools - Assessing Parallel Application Memory Utilisation at Scale". In: *Computer Performance Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 148–162. DOI: 10.1007/978-3-642-24749-1_12.

[251] DJ Delorie, Carlos O'Donell, and Florian Weimer. 2016. URL: https://blog.linuxplumbersconf.org/2016/ocw/system/presentations/3921/original/LPC%202016%20-%20linux%20and%20glibc_%20The%204.5TiB%20malloc%20API%20trace.pdf.

[252] *Tracing the Malloc API*. URL: https://www.sourceware.org/glibc/wiki/MallocTracing.

[253] Xu Ji, Chao Wang, Nosayba El-Sayed, Xiaosong Ma, Youngjae Kim, Sudharshan S. Vazhkudai, Wei Xue, and Daniel Sanchez. "Understanding object-level memory access patterns across the spectrum". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126917. URL: https://doi.org/10.1145/3126908.3126917.

[254] Matthew Hertz and Emery D. Berger. "Quantifying the performance of garbage collection vs. explicit memory management". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 313–326. ISBN: 1595930310. DOI: 10.1145/1094811.1094836. URL: https://doi.org/10.1145/1094811.1094836.

[255] Mathieu Poirier. *Coresight - HW Assisted Tracing on ARM*. 2014. URL: https://docs.kernel.org/trace/coresight/coresight.html.

[256] *Analyzing CPU usage with the Processor Trace instrument | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/xcode/analyzing-cpu-usage-with-processor-trace.

[257] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. "REPT: Reverse Debugging of Failures in Deployed Software". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 17–32. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/weidong.

[258] Farzam Dorostkar. *ThreadMonitor: Low-overhead Data Race Detection using Intel PT*. 2023. URL: https://tracingsummit.org/ts/2023/files/Farzam_Dorostkar-Tracing_Summit.pdf.

[259] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. "Efficient Protection of Path-Sensitive Control Security". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 131–148. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding.

[260] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. "PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace". In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY '17. Scottsdale, Arizona, USA: Association for Computing Machinery, 2017, pp. 173–184. ISBN: 9781450345231. DOI: 10.1145/3029806.3029830. URL: https://doi.org/10.1145/3029806.3029830.

[261] Xinyang Ge, Weidong Cui, and Trent Jaeger. "GRIFFIN: Guarding Control Flows Using Intel Processor Trace". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 585–598. ISBN: 9781450344654. DOI: 10.1145/3037697.3037716. URL: https://doi.org/10.1145/3037697.3037716.

[262] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. "PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 633–645. ISBN: 9781450367523. DOI: 10.1145/3321705.3329828. URL: https://doi.org/10.1145/3321705.3329828.

[263] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[264] *The grind tactic – The Lean Language Reference*. URL: https://lean-lang.org/doc/reference/4.22.0-rc2/The--grind--tactic/.

Appendix A

Implementation of linked lists using Steel combinators

A.1 A basic linked lists library using combinators

In this section, we include a basic linked lists library that is defined using Steel combinators.

```
module LList.Selectors

open Steel.Effect.Atomic
open Steel.Effect
open Steel.Reference
module Mem = Steel.Memory
module L = FStar.List.Tot
module G = FStar.Ghost

noeq
type cell (a: Type0) = {
  next: ref (cell a);
  data: a
}

let t (a: Type0) = ref (cell a)
let is_null_t (#a: Type0) (ptr: t a) = is_null ptr
let next (#a: Type0) (c: cell a) : t a
  = c.next
let mk_cell (#a: Type0) (next: t a) (data: a)
  = {next; data}

let null_rewrite (#a: Type0) (ptr: t a) (_:unit)
  : GTot (list a)
  = []
```

```
26
27 let rec llist0 (#a: Type0) (ptr: t a) (n: G.erased nat)
28   : Pure vprop
29   (requires True)
30   (ensures fun r -> t_of r == list a)
31   (decreases G.reveal n)
32   =
33   if (G.reveal n = 0)
34   then (
35     pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr)
36   )
37   else (
38     (vdep
39       (vptr ptr)
40       (fun v -> llist0 v.next (G.hide (n-1))))
41     `vrewrite`
42     (fun x -> (dfst x).data :: (dsnd x))
43   )
44
45 let llist_sl (#a: Type0) (ptr: t a) (n: G.erased nat) : Mem.slprop u#1
46   = hp_of (llist0 ptr n)
47
48 let llist_sel (#a: Type0) (ptr: t a) (n: G.erased nat)
49   : selector (list a) (llist_sl ptr n)
50   = fun m -> sel_of (llist0 ptr n) m
51
52 [@@__steel_reduce__]
53 let llist' #a r n : vprop' =
54   {hp = llist_sl r n;
55    t = list a;
56    sel = llist_sel r n}
57 unfold
58 let llist (#a: Type0) (r:t a) (n: G.erased nat) = VUnit (llist' r n)
59
60 [@@ __steel_reduce__]
61 let v_llist (#a: Type0) (#p: vprop) (r:t a) (n: nat)
62   // some boilerplate
63   (h:rmem p{FStar.Tactics.with_tactic selector_tactic (
64     can_be_split p (llist r n) /\ True)})
65   : GTot (list a)
66   = h (llist r n)
67
68 let llist_to_llist0 (#opened:_) (#a: Type0) (ptr: t a) (n: G.erased nat)
69   : SteelGhost unit opened
70   (llist ptr n)
71   (fun _ -> llist0 ptr n)
72   (requires fun _ -> True)
```

```
  (ensures fun h0 _ h1 ->
    h0 (llist ptr n) == h1 (llist0 ptr n)
  )
  =
  change_slprop_rel (llist ptr n) (llist0 ptr n)
    (fun x y -> x == y)
    (fun _ -> ())

let llist0_to_llist (#opened:_) (#a: Type0) (ptr: t a) (n: G.erased nat)
  : SteelGhost unit opened
  (llist0 ptr n)
  (fun _ -> llist ptr n)
  (requires fun _ -> True)
  (ensures fun h0 _ h1 ->
    h0 (llist0 ptr n) == h1 (llist ptr n)
  )
  =
  change_slprop_rel (llist0 ptr n) (llist ptr n)
    (fun x y -> x == y)
    (fun _ -> ())

let llist0_norm_nil (#a: Type0) (ptr: t a) (n: G.erased nat)
  : Lemma
  (requires G.reveal n = 0)
  (ensures
    pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr)
    ==
    (llist0 ptr n)
  )
  =
  assert_norm(begin
  if (G.reveal n = 0)
  then pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr)
  else (
    vdep (vptr ptr) (fun v -> llist0 v.next (n-1))
    `vrewrite`
    (fun x -> (dfst x).data :: (dsnd x))
  )
  end
  ==
  llist0 ptr n)

let llist0_sl_nil (#opened:_) (#a: Type0) (ptr: t a) (n: G.erased nat)
  : SteelGhost unit opened
  (llist ptr n) (fun _ -> llist ptr n)
  (requires fun _ -> G.reveal n = 0)
  (ensures fun h0 _ h1 ->
```

```
120     h0 (llist ptr n) == h1 (llist ptr n) /\
121     v_llist ptr n h0 == []
122   )
123   =
124   llist_to_llist0 ptr n;
125   llist0_norm_nil ptr n;
126   let s0 : G.erased (t_of (llist0 ptr n))
127     = gget (llist0 ptr n) in
128   let s1 : G.erased (list a)
129     = s0 in
130   change_equal_slprop
131     (llist0 ptr n)
132     (pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr));
133   elim_vrewrite (pure (is_null_t ptr)) (null_rewrite #a ptr);
134   intro_vrewrite (pure (is_null_t ptr)) (null_rewrite #a ptr);
135   change_equal_slprop
136     (pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr))
137     (llist0 ptr n);
138   llist0_to_llist ptr n;
139   noop ()
140
141 let llist0_norm (#a: Type0) (ptr: t a) (n: G.erased nat)
142   : Lemma
143   (requires n > 0)
144   (ensures
145     (vdep (vptr ptr) (fun v -> llist0 v.next (n-1))
146       `vrewrite`
147       (fun x -> (dfst x).data :: (dsnd x)))
148     ==
149     (llist0 ptr n)
150   )
151   =
152   assert_norm(begin
153   if (G.reveal n = 0)
154   then pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr)
155   else (
156     vdep (vptr ptr) (fun v -> llist0 v.next (n-1))
157     `vrewrite`
158     (fun x -> (dfst x).data :: (dsnd x))
159   )
160   end
161   ==
162   llist0 ptr n)
163
164 let llist0_sl_subst (#opened:_) (#a: Type) (ptr: t a) (n1 n2: G.erased nat)
165   : SteelGhost unit opened
166   (vdep (vptr ptr) (fun v -> llist0 v.next n1)
```

```
167     `vrewrite`
168     (fun x -> (dfst x).data :: (dsnd x)))
169   (fun _ -> vdep (vptr ptr) (fun v -> llist0 v.next n2)
170     `vrewrite`
171     (fun x -> (dfst x).data :: (dsnd x)))
172   (requires fun _ -> n1 == n2)
173   (ensures fun h0 _ h1 ->
174     h0 (vdep (vptr ptr) (fun v -> llist0 v.next n1)
175       `vrewrite`
176       (fun x -> (dfst x).data :: (dsnd x)))
177     ==
178     h1 (vdep (vptr ptr) (fun v -> llist0 v.next n2)
179       `vrewrite`
180       (fun x -> (dfst x).data :: (dsnd x)))
181   )
182   =
183   change_equal_slprop
184     (vdep (vptr ptr) (fun v -> llist0 v.next n1)
185       `vrewrite`
186       (fun x -> (dfst x).data :: (dsnd x)))
187     (vdep (vptr ptr) (fun v -> llist0 v.next n2)
188       `vrewrite`
189       (fun x -> (dfst x).data :: (dsnd x)))
190
191 val llist0_sl_pack (#opened:_) (#a: Type0) (ptr: t a) (n: G.erased nat)
192   : SteelGhost unit opened
193   (vdep (vptr ptr) (fun v -> llist0 v.next n)
194     `vrewrite`
195     (fun x -> (dfst x).data :: (dsnd x)))
196   (fun _ -> llist0 ptr (n+1))
197   (requires fun h0 -> not (is_null ptr))
198   (ensures fun h0 _ h1 ->
199     h0 (vdep (vptr ptr) (fun v -> llist0 v.next n)
200       `vrewrite`
201       (fun x -> (dfst x).data :: (dsnd x)))
202     ==
203     h1 (llist0 ptr (n+1))
204   )
205
206 let llist0_sl_pack ptr n
207   =
208   let x = n+1 in
209   llist0_sl_subst ptr n (x-1);
210   llist0_norm ptr x;
211   change_equal_slprop
212     (vdep (vptr ptr) (fun v -> llist0 v.next (x-1))
213       `vrewrite`
```

```
214       (fun x -> (dfst x).data :: (dsnd x)))
215     (llist0 ptr x);
216   change_equal_slprop
217     (llist0 ptr x)
218     (llist0 ptr (n+1))
219
220 val cons (#opened:_) (#a: Type0) (hd: t a) (ptr: t a) (n: G.erased nat)
221   : SteelGhost unit opened
222   (vptr hd `star` llist ptr n)
223   (fun _ -> llist hd (n+1))
224   (requires fun h0 ->
225     let v : cell a = sel hd h0 in
226     v.next == ptr
227   )
228   (ensures fun h0 r h1 ->
229     let v : cell a = sel hd h0 in
230     let l0 : list a = v_llist ptr n h0 in
231     let l1 : list a = v_llist hd (n+1) h1 in
232     l1 == v.data :: l0
233   )
234
235 let cons hd ptr n
236   =
237   let h0 = get () in
238   let c = G.hide (sel hd h0) in
239   (**) vptr_not_null hd;
240   (**) assert (G.reveal (next c) == ptr);
241   (**) llist_to_llist0 ptr n;
242   (**) intro_vdep
243     (vptr hd)
244     (llist0 ptr n)
245     (fun v -> llist0 v.next n);
246   (**) intro_vrewrite
247     (vdep (vptr hd) (fun v -> llist0 v.next n))
248     (fun x -> (dfst x).data :: (dsnd x));
249   (**) assert (not (is_null hd));
250   (**) llist0_sl_pack hd n;
251   (**) llist0_to_llist hd (n+1)
252
253 val cons_malloc (#a: Type0) (ptr: t a) (n: G.erased nat) (v: a)
254   : Steel (t a)
255   (llist ptr n)
256   (fun r -> llist r (n+1))
257   (requires fun _ -> True)
258   (ensures fun h0 r h1 ->
259     let l0 : list a = v_llist ptr n h0 in
260     let l1 : list a = v_llist r (n+1) h1 in
```

```
261     l1 == v :: l0
262   )
263
264 let cons_malloc ptr n v
265   =
266   let c = mk_cell ptr v in
267   let ptr' = malloc c in
268   cons ptr' ptr n;
269   return ptr'
270
271 val intro_nil (#a: Type0) (_: unit)
272   : Steel (t a)
273   emp (fun ptr -> llist ptr 0)
274   (requires fun _ -> True)
275   (ensures fun  _ ptr h1 ->
276     let l1 : list a = v_llist ptr 0 h1 in
277     l1 == [] /\
278     is_null_t ptr
279   )
280
281 let intro_nil #a _
282   =
283   (**) llist0_norm_nil #a null 0;
284   intro_pure (is_null_t #a null);
285   (**) intro_vrewrite (pure (is_null_t null)) (null_rewrite #a null);
286   (**) change_equal_slprop
287     (pure (is_null_t null) `vrewrite` (null_rewrite #a null))
288     (llist0 #a null 0);
289   (**) llist0_to_llist null 0;
290   return null
291
292 val tl (#a: Type0) (ptr: t a) (n: G.erased nat{n > 0})
293   : Steel (t a)
294   (llist ptr n)
295   (fun ptr' -> vptr ptr `star` llist ptr' (n-1))
296   (requires fun _ -> True)
297   (ensures fun h0 ptr' h1 ->
298     let l0 : list a = v_llist ptr n h0 in
299     let l1 : list a = v_llist ptr' (n-1) h1 in
300     let c : cell a = sel ptr h1 in
301     Cons? l0 /\
302     L.tl l0 == l1 /\
303     c.data == L.hd l0 /\
304     c.next == ptr'
305   )
306
307 let tl ptr n
```

```
308   =
309   (**) llist_to_llist0 ptr n;
310   (**) llist0_norm ptr n;
311   (**) change_equal_slprop
312     (llist0 ptr n)
313     (vdep (vptr ptr) (fun v -> llist0 v.next (n-1))
314       `vrewrite`
315       (fun x -> (dfst x).data :: (dsnd x)));
316   (**) elim_vrewrite
317     (vdep (vptr ptr) (fun v -> llist0 v.next (n-1)))
318     (fun x -> (dfst x).data :: (dsnd x));
319   (**) let c_ghost = elim_vdep (vptr ptr) (fun v -> llist0 v.next (n-1)) in
320   let c = read ptr in
321   (**) assert (G.reveal c_ghost == c);
322   (**) change_equal_slprop
323     (llist0 (next c_ghost) (n-1))
324     (llist0 c.next (n-1));
325   (**) llist0_to_llist (next c) (n-1);
326   return c.next
327
328 val length (#a: Type0) (ptr: t a) (n: G.erased nat)
329   : Steel nat
330   (llist ptr n)
331   (fun _ -> llist ptr n)
332   (requires fun _ -> True)
333   (ensures fun h0 l h1 ->
334     let l0 : list a = v_llist ptr n h0 in
335     h0 (llist ptr n) == h1 (llist ptr n) /\
336     L.length l0 == l /\
337     l == G.reveal n
338   )
339
340 module P = Steel.FractionalPermission
341
342 let lemma_null_implies_zero (#a: Type0) (ptr: t a) (n: G.erased nat)
343   (l: G.erased (list a))
344   (m: Mem.mem)
345   : Lemma
346   (requires
347     Mem.interp (hp_of (llist ptr n)) m /\
348     llist_sel ptr n m == G.reveal l /\
349     is_null_t ptr
350   )
351   (ensures
352     G.reveal n = 0
353   )
354   =
```

```
355   if (G.reveal n = 0)
356   then ()
357   else (
358     llist0_norm ptr n;
359     interp_vdep_hp
360       (vptr ptr) (fun v -> llist0 v.next (n - 1)) m;
361     ptr_sel_interp ptr m;
362     pts_to_not_null ptr P.full_perm (ptr_sel ptr m) m
363   )
364
365 let lemma_nonnull_implies_positive (#a: Type0) (ptr: t a) (n: G.erased nat)
366   (l: G.erased (list a))
367   (m: Mem.mem)
368   : Lemma
369   (requires
370     Mem.interp (hp_of (llist ptr n)) m /\
371     llist_sel ptr n m == G.reveal l /\
372     not (is_null_t ptr)
373   )
374   (ensures
375     G.reveal n > 0
376   )
377   =
378   if (G.reveal n = 0)
379   then (
380     llist0_norm_nil ptr n;
381     assert (hp_of (llist ptr n) == hp_of (pure (is_null_t ptr)));
382     Mem.pure_interp (is_null_t ptr) m
383   ) else ()
384
385 let rec length ptr n
386   =
387   (**) let h0 = get () in
388   (**) let l = G.hide (v_llist ptr n h0) in
389   if (is_null_t ptr)
390   then (
391     (**) extract_info (llist ptr n) l (G.reveal n = 0)
392       (lemma_null_implies_zero ptr n l);
393     (**) assert (G.reveal n = 0);
394     (**) llist0_sl_nil ptr n;
395     return 0
396   ) else (
397     (**) extract_info (llist ptr n) l (G.reveal n > 0)
398       (lemma_nonnull_implies_positive ptr n l);
399     (**) assert (G.reveal n > 0);
400     let ptr' = tl ptr n in
401     let l' = length ptr' (n-1) in
```

```
402     let l = 1 + l' in
403     (**) cons ptr ptr' (n-1);
404     (**) change_equal_slprop
405       (llist ptr (n-1+1))
406       (llist ptr n);
407     return l
408   )
```

A.2 Equivalence to a textbook linked lists predicate

In this section, we include F* proofs that a variation over the previously defined linked lists predicate using combinators is equivalent to textbook linked lists. The only slight change is that the selector of the defined predicate is now of type `list (cell a)` in order to make some proofs easier. This is due to the way the considered textbook linked lists library is implemented, see the `Selectors.LList` example included in the Steel repository. Recovering previous selector type `list a` from the defined separation logic predicate `llist_cells` would be straightforward, e.g., by adding another `vrewrite` combinator on top of it. Finally, we note that while the textbook predicate defined in the `Selectors.LList` module is not exactly the textbook one mentioned in Section 2.2.3.2, it is equivalent to it [62].

```
module LList.Selectors.Equiv

module G = FStar.Ghost

open Steel.Effect.Atomic
open Steel.Effect
open Steel.Reference
module Mem = Steel.Memory
module L = FStar.List.Tot
module P = Steel.FractionalPermission

// https://github.com/FStarLang/steel/
// file: share/steel/examples/steel/Selectors.LList.fst
module SL = Selectors.LList
friend Selectors.LList

let cell (a: Type0) = SL.cell a
let t (a: Type0) = SL.t a

let is_null_t (#a: Type0) (ptr: t a) = is_null ptr
let null_rewrite (#a: Type0) (ptr: t a) (_:unit)
  : GTot (list (cell a))
  = []

let f_rewrite (#a: Type0) (v: dtuple2 (cell a) (fun _ -> list (cell a)))
  : Tot (list (cell a))
  =
  dfst v :: dsnd v

// selectors is of type (cell a)
let rec llist_cells (#a: Type0) (ptr: t a) (n: nat)
  : Pure vprop
  (requires True)
```

```
34   (ensures fun r -> t_of r == list (cell a))
35   (decreases n)
36   =
37   if (n = 0)
38   then (
39     pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr)
40   )
41   else (
42     (vdep
43       (vptr ptr)
44       (fun v -> llist_cells v.next (n-1)))
45     `vrewrite`
46     (fun x -> f_rewrite x)
47   )
48
49 let llist_cells_norm (#a: Type0) (ptr: t a) (n: nat)
50   : Lemma
51   (requires n > 0)
52   (ensures
53     (vdep (vptr ptr) (fun v -> llist_cells v.next (n-1))
54       `vrewrite`
55       (fun x -> f_rewrite x))
56     ==
57     (llist_cells ptr n)
58   )
59   =
60   assert_norm(begin
61   if (n = 0)
62   then pure (is_null_t ptr) `vrewrite` (null_rewrite #a ptr)
63   else (
64     vdep (vptr ptr) (fun v -> llist_cells v.next (n-1))
65     `vrewrite`
66     (fun x -> f_rewrite x)
67   )
68   end
69   ==
70   llist_cells ptr n)
71
72 let llist_pack (#a:Type0)
73   (ptr: t a) (n: nat) (m:Mem.mem)
74   : Lemma
75   (requires
76     n > 0 /\
77     Mem.interp (hp_of (vptr ptr)) m /\
78     (let hd = sel_of (vptr ptr) m in
79     Mem.interp (hp_of (vptr ptr) `Mem.star`
80                 hp_of (llist_cells hd.next (n-1))) m
```

```
  ))
  (ensures
    Mem.interp (hp_of (llist_cells ptr n)) m /\
    (let hd = sel_of (vptr ptr) m in
    let tl = sel_of (llist_cells hd.next (n-1)) m in
    let l = sel_of (llist_cells ptr n) m in
    l == hd :: tl
  ))
  =
  let hd = sel_of (vptr ptr) m in
  let tl = sel_of (llist_cells hd.next (n-1)) m in
  llist_cells_norm ptr n;
  interp_vdep_hp
    (vptr ptr) (fun v -> llist_cells v.next (n - 1)) m;
  vdep_sel_eq
    (vptr ptr) (fun v -> llist_cells v.next (n - 1)) m;
  // vrewrite selector: see vrewrite_sel, casts seem required
  let x = sel_of (vdep (vptr ptr) (fun v -> llist_cells v.next (n-1))) m in
  assert (x == (|hd,tl|));
  let p = vdep (vptr ptr) (fun v -> llist_cells v.next (n-1)) in
  let open FStar.Tactics in
  assert (hp_of p == normal (hp_of p)) by trefl();
  vrewrite_sel_eq
    (vdep (vptr ptr) (fun v -> llist_cells v.next (n -1)))
    (fun x -> f_rewrite x) m;
  let y : _ = (vrewrite_sel
    (vdep (vptr ptr) (fun v -> llist_cells v.next (n -1)))
    (fun x -> f_rewrite x)
    <: selector' _ _ ) m in
  let z : _ = f_rewrite ((normal (sel_of
    (vdep (vptr ptr) (fun v -> llist_cells v.next (n - 1))))
    <: selector' _ _) m) in
  assert (y == z);
  assert_norm (z == f_rewrite x)

let rec equiv_from_oldstyle_to_newstyle (#a:Type0)
  (ptr: t a) (l: list (cell a)) (m:Mem.mem)
  : Lemma
  (requires Mem.interp (SL.llist_sl' ptr l) m)
  (ensures
    Mem.interp (hp_of (llist_cells ptr (L.length l))) m /\
    sel_of (llist_cells ptr (L.length l)) m == l
  )
  (decreases (L.length l))
  =
  match l with
  | [] ->
```

```
128     Mem.pure_interp (ptr == null #(cell a)) m;
129     Mem.pure_interp (is_null_t ptr) m;
130     vrewrite_sel_eq (pure (is_null_t ptr)) (null_rewrite #a ptr) m
131
132   | hd :: tl ->
133     // SL.llist_sl' = p1 * p2 * p3
134     let p1 = pts_to_sl ptr P.full_perm hd in
135     let p2 = SL.llist_sl' hd.next tl in
136     let p3 = Mem.pure (ptr =!= SL.null_llist) in
137     let m12, m3 = Mem.id_elim_star (p1 `Mem.star` p2) p3 m in
138     let m1, m2 = Mem.id_elim_star p1 p2 m12 in
139     // llist_cells = p1' * p2'
140     let p1' = vptr ptr in
141     let p2' = llist_cells hd.next (L.length l - 1) in
142
143     intro_ptrp_interp ptr P.full_perm hd m1;
144     assert (Mem.interp (hp_of p1') m1);
145     ptr_sel_interp ptr m1;
146     pts_to_witinv ptr P.full_perm;
147     assert (sel_of p1' m1 == hd);
148
149     equiv_from_oldstyle_to_newstyle hd.next tl m2;
150     assert (Mem.interp (hp_of p2') m2);
151     assert (sel_of p2' m2 == tl);
152
153     Mem.intro_star (hp_of p1') (hp_of p2') m1 m2;
154     Mem.join_commutative m1 m2;
155     assert (G.reveal m12 == Mem.join m1 m2);
156     llist_pack ptr (L.length l) m12
157
158 let llist_unpack (#a:Type0)
159   (ptr: t a) (n: nat) (m:Mem.mem)
160   : Lemma
161   (requires
162     Mem.interp (hp_of (llist_cells ptr n)) m /\
163     n > 0)
164   (ensures (
165     Mem.interp (hp_of (vptr ptr)) m) /\
166     (let hd = sel_of (vptr ptr) m in
167     Mem.interp (hp_of (vptr ptr) `Mem.star` hp_of (llist_cells hd.next (n-1))) m /\
168     (let tl = sel_of (llist_cells hd.next (n-1)) m in
169     let l = sel_of (llist_cells ptr n) m in
170     l == hd :: tl
171   )))
172   =
173   let l = sel_of (llist_cells ptr n) m in
174   llist_cells_norm ptr n;
```

```
  let p = vdep (vptr ptr) (fun v -> llist_cells v.next (n-1)) in
  let open FStar.Tactics in
  assert (hp_of p == normal (hp_of p)) by trefl();
  assert (Mem.interp (hp_of
    (vdep (vptr ptr) (fun v -> llist_cells v.next (n - 1)))
  ) m);
  interp_vdep_hp
    (vptr ptr) (fun v -> llist_cells v.next (n - 1)) m;
  vdep_sel_eq
    (vptr ptr) (fun v -> llist_cells v.next (n - 1)) m;
  let x = sel_of (vdep (vptr ptr) (fun v -> llist_cells v.next (n-1))) m in
  let hd = sel_of (vptr ptr) m in
  let tl = sel_of (llist_cells hd.next (n-1)) m in
  // vrewrite selector: see vrewrite_sel, casts seem required
  vrewrite_sel_eq
    (vdep (vptr ptr) (fun v -> llist_cells v.next (n -1)))
    (fun x -> f_rewrite x) m;
  let y : _ = (vrewrite_sel
    (vdep (vptr ptr) (fun v -> llist_cells v.next (n -1)))
    (fun x -> f_rewrite x)
    <: selector' _ _ ) m in
  let z : _ = f_rewrite ((normal (sel_of
    (vdep (vptr ptr) (fun v -> llist_cells v.next (n - 1))))
    <: selector' _ _) m) in
  assert (y == z);
  assert_norm (z == f_rewrite x)

let rec equiv_from_newstyle_to_oldstyle (#a:Type0)
  (ptr: t a) (n: nat) (m:Mem.mem)
  : Lemma
  (requires Mem.interp (hp_of (llist_cells ptr n)) m)
  (ensures (
    let l = sel_of (llist_cells ptr n) m in
    Mem.interp (SL.llist_sl' ptr l) m
  ))
  (decreases n)
  = match n with
  | 0 ->
    Mem.pure_interp (ptr == null #(cell a)) m;
    Mem.pure_interp (is_null_t ptr) m;
    vrewrite_sel_eq (pure (is_null_t ptr)) (null_rewrite #a ptr) m

  | _ ->
    llist_unpack ptr n m;
    // llist_cells = p1 * p2
    let p1 = vptr ptr in
    assert (Mem.interp (hp_of p1) m);
```

```
222     let hd = sel_of (vptr ptr) m in
223     let p2 = llist_cells hd.next (n-1) in
224     assert (Mem.interp (hp_of p2) m);
225     let tl = sel_of p2 m in
226     let l = sel_of (llist_cells ptr n) m in
227     assert (l == hd :: tl);
228     let m1, m2 = Mem.id_elim_star (hp_of p1) (hp_of p2) m in
229     // SL.llist_sl' = p1' * p2' * p3'
230     let p1' = pts_to_sl ptr P.full_perm hd in
231     let tl = sel_of p2 m2 in
232     let p2' = SL.llist_sl' hd.next tl in
233     let p3' = Mem.pure (ptr =!= SL.null_llist #a) in
234
235     // 1. pts_to
236     ptr_sel_interp ptr m1;
237     assert (Mem.interp p1' m1);
238     // 2. recursive call to SL.llist_sl'
239     equiv_from_newstyle_to_oldstyle hd.next (n - 1) m2;
240     assert (Mem.interp p2' m2);
241     // 3. ptr =!= null
242     pts_to_not_null ptr P.full_perm hd m1;
243     assert (ptr =!= SL.null_llist #a);
244
245     // SL validity
246     Mem.intro_star p1' p2' m1 m2;
247     Mem.join_commutative m1 m2;
248     assert (G.reveal m == Mem.join m1 m2);
249     Mem.emp_unit (p1' `Mem.star` p2');
250     Mem.pure_star_interp (p1' `Mem.star` p2')
251       (ptr =!= SL.null_llist #a)
252       m;
253     assert (Mem.interp (SL.llist_sl' ptr (hd::tl)) m)
```