SRI International

August 16, 2017

Where Art Thou Heap Metadata?

DARPA Contract Number N66001-15-C-4061

Final Report

Ian A. Mason (PI), Drew Dean, Bruno Dutertre,
Jorge A. Navas, and Dan Wallach

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document summarizes the research performed under DARPA Contract Number N66001-15-C-4061 by SRI International, and presents the project's results. The project started in August 2015 and was completed in August 2017. The Principal Investigator for this project was Drew Dean, until his departure in July 2016. Ian A. Mason took over as PI after Drew Dean left. The co-investigators were Bruno Dutertre (SRI), Jorge Navas (SRI), and Dan Wallach (Rice University).

1.1 Project Overview

Almost all modern programs make extensive use of dynamically allocated data structures such as linked lists, trees, and other pointer- or reference-filled data structures. If the lifetime of the data exceeds the lifetime of the function or method allocating the data, as is often the case, then the data can not be allocated on the stack, and is instead allocated from the heap.[1] Dynamically allocated data structures are popular because they allow programs not to have hard-coded limits in them, and adapt their memory use to the actual usage of the program, rather than a developer's best guess as to how the application will be used.

There are two primary downsides to dynamically allocated data structures:

1. Naïve use of dynamic memory allocation can incur significant performance cost.

2. Pointer usage errors can introduce additional security vulnerabilities into software, particularly a class of vulnerability known as a heap overflow.

[1] There are conflicting uses of the word "heap" as a heap is also the name of a specific data structure. We use the computer science systems terminology in this white paper, where a process' address space is divided into sections commonly referred to as code (aka text), data, heap, and stack space.

Figure 1.1: Schematic Layout of a Standard Heap. Malloc metadata is in red, dynamically-allocated data structures are in black, and free memory is in white. Both red and black areas have pointers to same-colored addresses. A real heap would have thousands/millions regions of red and black.

This project investigates backward-compatible improvements to dynamic memory allocation that mitigate the security vulnerabilities of dynamically allocated data structures.

1.2 Background

We focus on applications written in C that use the system provided `malloc()` and `free()` routines.[2] As expected, the use of dynamically allocated data structures incurs some overhead; in particular, `malloc()` and `free()` need to manage heap space, keeping track of what memory is currently in use, and what memory is currently free and available. We call this overhead *malloc metadata*, as there is no standard terminology. The traditional place to store malloc metadata is in the heap itself, and the most troublesome heap overflow exploits take advantage of this placement.[3] Figure 1.1 shows a simple visualization of the heap. The important point is that the dynamically allocated data and malloc metadata are contiguous and interspersed. The color of any given heap location may change over time, as memory is allocated and returned to the heap. The `free()` routine, however, implicitly assumes that addresses it expects to be red, are actually red. Given certain pointer errors (such as the double freeing of a pointer), an attacker can violate this assumption, which can lead to arbitrary code execution. An illustrative example can be found in CERT Advisory CA-2002-07, "Double Free Bug in zlib Compression Library." Regrettably little progress has been made in the intervening 13 years, as this vulnerability has been repeated over and over.

[2]For simplicity of discussion, we omit `calloc()`, `realloc()`, `valloc()`, and `posix_memalign()`, and the default `C++ new` and `delete` operators. These pose the exact same issues and are covered by our approach.

[3]Pointer misuses can introduce arbitrary bugs and vulnerabilities into software, but overwriting of malloc metadata is the predominant exploitation vector of heap overflow vulnerabilities.

Figure 1.2: Schematic Heap with Separate Metadata. Color codes are as in Fig. 1.1.

1.3 Approach

We propose to refactor the implementation of `malloc()` and `free()` to produce a heap layout as shown in Figure 1.2. The important difference is that the malloc metadata is held in a separate memory region, one that is not contiguous with the heap. This metadata region is allocated by the operating system at a randomized address, and so its location cannot be deduced from the address of the client memory.

This separation improves security, but its implementation requires a mechanism for locating the metadata associated with a particular client memory address. Implementing this *lookup mechanism* efficiently—especially in a multi-threaded situation—is the core problem we face in this project.

In the single-threaded case, a hash table is an efficient data structure that adds minimal overhead to finding the metadata associated with a dynamically allocated pointer. This prevents simple errors from corrupting malloc metadata. Instead of reallocated heap data getting confused with malloc metadata (as in CERT-2002-07), an attacker would need primitives to both read the pointer to this region, and overwrite arbitrary memory locations. While such attack primitives are certainly possible, the attacker's workload has been increased, and the current stockpile of heap overflow vulnerabilities become obsolete. A side benefit of our approach is that a double free bug can be readily detected and ignored or logged for debugging purposes. In an ideal world, the malloc metadata would be stored in a separate protection domain. In the CHERI architecture developed by SRI and Cambridge University in DARPA's CRASH program, this would be easy to implement efficiently with capabilities. Conventional machines would need to use another process to guarantee separation, which may impose too much overhead, depending on the performance of interprocess communication mechanisms.

1.4 Accomplishments

In year one of this project, we produced several prototypes, culminating with an alpha version of a variant of glibc's malloc for x86-64 Linux called `sri-glibc`. This allocator does not include any metadata in client memory. It is *thread safe*, and introduces no

new contention compared to the glibc original. Ongoing benchmarking indicates that the overhead is often very much less than 10%. One of the key ingredients to `sri-glibc` is an efficient lock-free mechanism that maps client addresses to the arena the memory belongs to.

In year two, we focused on verification and re-implementation of our hash table. The version we developed in year one had several concurrency bugs that we fixed in year two. In addition, we have addressed a significant issue related to memory management. Our hash tables rely on linear probing and expand when the current capacity is insufficient. This creates larger and larger copies of the table. Old copies eventually become inaccessible and not touched by any thread. Our year-one implementation did not keep track of this and never released any copy. The new implementation detects when old copies are no longer necessary and releases the corresponding memory to the operating system. We have developed our new hash-table implementation as a stand-alone package, independent of malloc.

We have also formalized key aspects of the hash-table implementation, namely the growing and data migration mechanism. We have developed a state-machine model of these mechanisms and verified critical properties using the SAL model checker.

1.5 Publications

The following paper describes improvements to the SeaHorn static analyzer that were developed as part of this project, in an attempt to prove the absence of null pointer dereferences in openBSD's malloc:

> Arie Gurfinkel and Jorge A. Navas, A Context-Sensitive Memory Model for Verification of C/C++ Programs, *Static Analysis Symposium (SAS 2017)*.

This paper will be presented at the 24th Static Analysis Symposium to be held in New York City from August 30th to September 1st, 2017.

1.6 Software

We have developed several prototypes of malloc and lock-free hash tables during the project. Our re-implementation of GNU Libc's malloc that separates metadata from the head is publicly available under an open-source license at `https://github.com/SRI-CSL/sri-glibc-malloc`. We are planning to publish our latest lock-free hash table using a similar model at `https://github.com/SRI-CSL/LFHT`, pending open-source approval.

Chapter 2

Summary of Research and Results

The main results of this project are prototype implementations of malloc libraries, and implementations of lock-free hash tables. We also applied verification tools including static analyzers and model checkers to these software implementation and algorithms.

2.1 Family Tree of Glibc Allocators

Most of the malloc prototypes we developed belong to the family of memory allocators commonly used with Linux, in particular the Glibc-based distributions. These include most Linux distributions that do not target embedded systems. Figure 2.1 gives an abridged family tree of these memory allocators.

The root of the tree is Doug Lea's malloc (`dlmalloc`), which was born in 1987. It was originally intended for single threaded applications with scarce resources. Gloger's allocator (`ptmalloc2`) extends `dlmalloc` with *arenas*. Arenas are independent self-contained heaps that can be utilized simultaneously by separate threads, reducing contention for global locks. Glibc's built-in memory allocator is a direct descendent of `ptmalloc2`. `Ptmalloc2`'s main attraction over Glibc malloc is its ease of compilation. Glibc malloc is deeply entwined in the Glibc infrastructure and is very difficult to extract into a stand-alone library. Glibc has also evolved somewhat since its fork from `ptmalloc2`, the arena locking is somewhat more complex, and the metadata has grown to include skip lists for the bin free lists.

Piessens's `Dnmalloc` [33] is a package from 2005 that modified `dlmalloc` to move its metadata out of the heap, much like our proposal. This implementation used a lookup table based on a 32 bit address space that could not be extended to modern x86-64 hardware.

The green nodes in Figure 2.1 show three prototypes that we developed in this project.

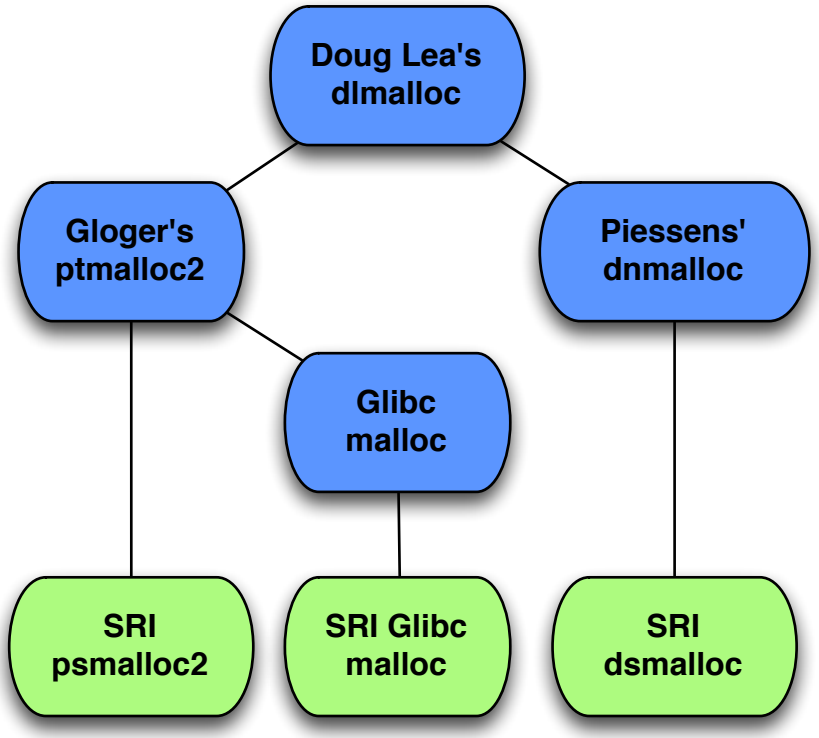

Figure 2.1: The family tree of C memory allocators starting from Doug Lea's work in the 1990s. Blue allocators are ancestors of our work. Green allocators have been developed within this project.

- Our first prototype (`dsmalloc`) generalized Doug Lea's malloc along the lines of `dnmalloc` by implementing linear hashing, a version of dynamic hashing that uses a variation of Larson's algorithm as published in [13].

- Our second prototype (`psmalloc2`) builds upon Gloger's malloc (`ptmalloc2`). `Psmalloc2` succeeds in removing almost all metadata from client memory. The only remaining piece of metadata is the index of the arena the memory belongs to. From a security point of view, what is critical is that the metadata in the heap is an integer, not a pointer. If the integer is corrupted (either by accident or by malice), `free()` or `realloc()` won't be able to find the pointer, and can signal an error, but these routines won't go chasing attacker supplied pointers and end up overwriting security-critical data.

- In our final prototype `sri-glibc`, we eliminated the need for an arena number in the client chunk by implementing a lock-free API to determine arena index from client pointer. We also tailored the metadata to improve performance by reducing the number of hash table lookups. This `sri-glibc` prototype is an alpha version. We tried to keep changes to a minimum, in the hope an upstream adoption, however as the changes added up, a cleaner approach becomes more apparent, and eventually essential.

In addition to the allocators mentioned previously—which all descend from Doug Lea's malloc—we also examined malloc implementations in Windows and Apple's OS X and iOS. A summary of implementation and security features is given in the Appendix.

2.2 Lock-Free Data Structures

Because we separate metadata from client memory, all our prototypes require a separate allocation mechanism for the metadata and the related data structures. These are not full-fledged allocators since they are dedicated to allocating objects of small range of (small) sizes. We call these *pool allocators*. To avoid excessive overhead, we wanted our pool allocators to be lock-free. This led us to Maged Michael's paper on scalable, lock-free dynamic memory allocation [18]. This paper received PLDI's "2004 Most Influential Paper Award" in 2014. The following quote explains this award.

> Maged Michaels PLDI04 paper is considered a landmark in memory allocation for multi-threaded programs, presenting the first completely lock-free general-purpose user-space allocator. It provides good performance with respect to scalability, speed and space efficiency, while at the same time only relying on common hardware and OS support. The work is highly regarded and frequently referenced, and is also the basis of multiple memory allocator implementations, both in IBM products and in follow-on research.

Working from a 32-bit x86 implementation of Michael's algorithm [25], we have developed a 64-bit x86_64 version (`elfpa`) that has no pointer metadata in the client memory. This is a full malloc implementation that relies on an almost lock-free hash table as a lookup mechanism.

Implementing a lock-free lookup was a major accomplishment of our project that enabled us to remove all metadata from Glibc's malloc library. This solved a key difficulty, namely, efficiently mapping an arbitrary client pointer to the relevant arena without incurring the cost of a locking contention.

In year one of the project, we implemented two versions of lock-free dynamic hash table. Both were influenced by Maier's paper [15]. Both implementations are based on

linear probing hash tables that expands when the current capacity is insufficient. The most complex aspect of such tables is the expansion phase. In one version, we relied on pthread locking and signaling primitives to allow a single thread to grow the table. This prevents concurrent access to the table while it is growing, which we considered a tolerable tradeoff. This implementation could not be ported to Glibc as the pthread primitives are not available. Instead, we developed another growing mechanism that requires no locking. In the growing phase, each thread that accesses the table must pay its due by migrating key-value pairs from the old table to the new one.

In year two, we continued developing and improving the lock-free dynamic hash table. We started a clean general-purpose version of the table, independent of malloc. The overall design did not change. The table maps 64 bit keys (i.e., integer or pointers) to 64 bit values. We use linear probing and store the entries in a fixed-size array. When this array is full, we create a new copy of double its size and migrate data from the previous array to the new array. As mentioned previously, this migration process is performed in a decentralized fashions. Any thread that accesses the hash table must first copy a fixed number of entries from the old to the new array. We also improved the algorithm so that old array copies that are no longer useful can be freed and the memory they use can be reclaimed by the operating system. For this purpose, we rely on reference counting to keep track of the number of threads that have a pointer to each array copy. Properly managing and updating these reference counters turned out to be quite difficult, and forced us to use locks in some critical parts of the code to avoid race conditions. The new implementation is then not entirely lock free. However, the locks are used only in limited contexts during the migration process and initialization of some data structures. In normal usage, operations such as adding or removing entries, and searching for a key do not require any locks. The appendix (Section C) presents the hash-table design and algorithms in detail.

To get confidence in this hash-table implementation, we performed extensive stress testing. We also applied formal verification to key parts of the migration process.

2.3 Formal Analysis

In year two of the project, we investigated several formal-method tools and applied them to an existing malloc implementation and to our own code, including our new dynamic hash table.

2.3.1 Static Analysis of Malloc

We have applied static analysis tools to a simplified version of OpenBSD's memory allocation library. This library implements a so-called BIBOP (Big Bag of Pages) allo-

cation algorithm. This allocator was implemented by Otto Moerbeek and was described at the EuroBSD Conference in 2009 [22]. This allocator is an interesting target for analysis since it is relatively simple and the code is small, but the allocator achieves good performance.

In a first attempt, we applied the SeaHorn static analyzer [9] to this code, without any modification. SeaHorn is an open-source verification tool for safety properties of programs. It works at the level of LLVM bitcode so it can be easily applied to software written in C or C++. SeaHorn can check safety properties such as the absence of null-pointer dereferences and buffer overflows. It can also verify user-provided properties written as C-like assertions. Our initial goal was to show that the OpenBSD malloc did not have null-pointer dereferences.

It turned out that some of the algorithmic tricks used by OpenBSD were beyond what can be modeled in SeaHorn, because the backend solver used by SeaHorn works on problem formulated in linear arithmetic. The OpenBDS code uses various bit-level encoding that do not translate to linear arithmetic. As a result, analysis with SeaHorn does not work well on the original code. The modeling to too imprecise and SeaHorn produces to too many false positives.

We addressed these issues by replacing problematic parts of the code with simpler variants (sometimes using non-deterministic constructs) that are easier for SeaHorn to handle. We were able to prove the absence of null-pointer dereferences for simple test drivers that allocate constant-size chunks of memory. We then extended these drivers to simulate an arbitrary number of calls to malloc immediately followed by calls to free. Each malloc uses a non-deterministic size argument. This test corresponds to the following loop:

```
while (*) { sz = random(); p = malloc(sz); free(p); }
```

For this example, we did not obtain a complete proof with SeaHorn. The tool did not find safe inductive invariants. But we could explore the code using SeaHorn's bounded model checking engine after unrolling all the loops eight times. We did not find any null pointer dereference.

A detailed analysis showed that the invariants necessary to prove the absence of null-pointer dereference are beyond what the backend solver can produce. For example, some of them require learning facts about integer divisibility and other require quantifiers.

Overall, we found that automated verification of malloc algorithms (even relatively simple ones) is beyond the reach of current static analysis techniques. Semi-automated techniques where the code is annotated by hand with invariants, pre-conditions, and post-conditions might be applicable (e.g., using FramaC [12]). We did not have the resource for such an exercise within the project.

2.3.2 Modeling and Verification of Lock-Free Algorithms

As an alternative to static analysis of code, we looked at verification of higher-level, more abstract models of algorithms and data structures. We focused on building state-machine models of some of the lock-free algorithms we use and verifying invariant properties using model checkers. We relied on the SAL and Sally model checkers.

SAL is a set of tools for specification and analysis of state machines [6], developed and distributed by SRI International. SAL includes traditional model-checking tools, including a BDD-based symbolic model checker and bounded model checkers for both finite and infinite systems. The state-machine models we developed were all infinite and we analyzed them using pure bounded model checking (to search for counterexamples to properties of interest) and so-called k-induction. K-induction is a generalization of the standard induction principle that can be used to prove invariant properties of state-transition systems. In most cases, k-induction is not fully automated and requires the user to discover useful auxiliary lemmas that can be proved by k-induction and imply the property one cares about.

Our main results are based on a state-machine model of the data migration algorithm used in our lock-free hash table. We proved a key invariant of this algorithm using SAL. The proof involves a set of auxiliary lemmas the we discovered and added by hand. Details of the model and verification results are presented in the appendix.

We also experimented with SRI's new model checker called Sally [7]. Sally is a more advanced tool than SAL. It extends traditional k-induction with algorithms that attempt to automatically discover an inductive strengthening of the property of interest. Sally builds on the IC3 method due to Bradley [1] and relies on SMT solvers to incrementally discover relevant auxiliary lemmas.

We attempted to verify the hash-table migration process using Sally, starting with the same state-machine model we built for SAL. Unfortunately, Sally does not manage to prove the key invariant on its own. It learns larger and larger sets of auxiliary invariants but it does not converge. This is a limitation of the underlying SMT technology used by Sally. We believe that one of the auxiliary lemma that we used in our manual proof with SAL is beyond what can be learned automatically by Sally. In summary, the proof is too hard for Sally to discover automatically. In future work, we will consider alternative techniques for learning auxiliary invariants based on abstract interpretation or other methods, which we believe might help in this case.

Although Sally did not work on the most complex model we developed in this project, we used it successfully to verify a simple implementation of the CAS instruction, a non-blocking queue, and Simpson's four-slot algorithm, a classical example of lock-free algorithm for asynchronous communication [28].

Chapter 3

Conclusion

Separating metadata from the heap improves software reliability by making heap-based attacks more difficult. One way to achieve this separation is to modify an existing memory-allocation library such as glibc. We have delivered such a modified glibc in the first year of this project. One problem with this approach is dealing with the inherent complexity of the glibc code, which our new allocator only adds to. An alternative is to start from a blank slate or form a clean code base such as the allocators of OpenBSD and FreeBSD, which may be a better solution and makes the code less difficult to formally analyze.

We have produced several prototypes allocators, most of which were derived from the glibc family tree, depicted in Figure 2.1. We also implemented Maged Michael's lock-free allocator, and a variety of lock-free data structures. This work culminates with an alpha version of a variant of glibc's malloc for x86-64 Linux called `sri-glibc`. This allocator does not include any metadata in client memory. It is *thread safe*, and introduces no new contention compared to the glibc original. Ongoing benchmarking indicates that the overhead is often very much less than 10%.

One of the key ingredients in all such allocators is an efficient mechanism for locating the metadata for a memory chunk given a pointer to this chunk. We have developed and implemented several versions of hash tables to satisfy this requirement. A key goal is to ensure minimum contention when accessing the hash table in multi-threaded applications. We started with completely lock-free implementations but these had disadvantages in terms of memory usage. In year two of the project, we designed and implemented a more-efficient version that is not completely lock-free but can release unreachable memory to the operating system. Although locks are used in some critical sections, most access to the table are lock free and introduce little contention between threads. Fully lock-free hash table that can grow and release unused memory is a hard research problem and we did not find any paper or implementation that convincingly

solve these issues.

We have investigated formal verification of memory-allocation algorithms and lock-free hash tables. We found that the algorithms and data structures used in malloc (even relatively simple versions of malloc) are too complex for out-of-the-box verification with state-of-the-art static analysis tools. Partial results based on bounded exploration are more practical, but we speculate that full verification at the source-code level requires hand-written annotations. The main issue is that malloc algorithms maintain global, complex data structures. Verification requires discovering non-trivial invariants about these data structures that are implicit in the code. The current generation of static-analysis tools does not have enough reasoning power to automatically learn such complex invariants.

We explored formal analysis using at a more abstract level than source code, by building state-machine models of algorithms. Formal verification is more tractable at this level, but still requires significant effort. We have successfully verified an important property of our hash-table implementation with model-checking tools. This provides some evidence of correctness even though the models are highly simplified. For example, the number of threads is fixed and small. We did not have time to explore manual verification with tools such as theorem provers which would be a natural extension of this work and potentially lead to more general correctness results.

Bibliography

[1] Aaron R Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87, 2011.

[2] Apple Corporation. bmalloc source code., 2016.

[3] Apple Corporation. libmalloc source code., 2016.

[4] Apple Corporation. Webkit source code., 2016.

[5] Standard Performance Evaluation Corporation. SPEC CPU 2006, 2006.

[6] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Tiwari Ashish. SAL 2. In Rajeev Alur and Doron A. Peled, editors, *Computer-Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004.

[7] Jovanović Dejan and Dutertre Bruno. Property-directed k-induction. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design (FMCAD 2016)*, pages 85–92, 2016.

[8] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[9] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Nava. The SeaHorn verification framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer-Aided Verification (CAV 2015)*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[10] Ben Hawkes. Attacking the Vista heap. In *RuxCon 2008*, Sydney, Australia, November 2008. `https://www.lateralsecurity.com/downloads/hawkes_ruxcon-nov-2008.pdf`, also presented at BlackHat 2008.

[11] Ken Johnson and Matt Miller. Exploit mitigation improvements in Windows 8. In *BlackHat 2012*, Las Vegas, NV, July 2012. `http://media.blackhat.`

com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf.

[12] Florent Kirchner, Nokolai Kosmatov, Virgile Prevosto, Julian Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[13] Per-Ake Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, April 1988.

[14] Zhenhua (Eric) Liu. Advanced heap manipulation in Windows 8. In *Black-HatEU 2013*, Amsterdam, Netherlands, March 2013. Video: https://www.youtube.com/watch?v=mDfS8yBLzZU, Paper: https://media.blackhat.com/eu-13/briefings/Liu/bh-eu-13-liu-advanced-heap-WP.pdf.

[15] Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: fast *and* general?(!). In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 34:1–34:2, 2016.

[16] Ian A. Mason, Jorge A. Navas, and Bruno Dutertre. *An (Almost) Lock Free Hash Table that grows dynamically.* SRI International, August 2017. https://github.com/SRI-CSL/LFHT.

[17] Sean Metcalf. *Microsoft Enhanced Mitigation Experience Toolkit (EMET) 5 Protection Methods*. Microsoft, August 2014. https://adsecurity.org/?p=157.

[18] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN conference on Programming langauge design and implementation*, PLDI'04, pages 35–46, New York, NK, USA, 2004. ACM.

[19] Microsoft. *Software Defense: mitigating heap corruption vulnerabilities*, October 2013. https://blogs.technet.microsoft.com/srd/2013/10/29/software-defense-mitigating-heap-corruption-vulnerabilities/.

[20] Charlie Miller, Dion Blazakis, Dino Dai Zovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. *The iOS Hacker's Handbook*. Wiley, May 2012. Heap attacks are discussed in chapter 7.

[21] Charlie Miller and Dino Dai Zovi. *The Mac Hacker's Handbook*. Wiley, 2009. Heap attacks are discussed in chapter 8, see also, https://www.trailofbits.com/resources/macos_xploitation_slides.pdf.

[22] Otto Moerbeek. A new malloc(3) for OpenBSD. In *EuroBSDCon*, 2009. http://openbsd.md5.com.ar/papers/eurobsdcon2009/otto-malloc.pdf.

[23] Nemo. OS X heap exploitation techniques. *Phrack*, 63(5), January 2005. http://phrack.org/issues/63/5.html.

[24] Reverser. Reverse engineering Mac OS X, 2016.

[25] Scott Schneider. Maged michaels lock-free memory allocator., 2006.

[26] Steven Seeley. Ghost in the allocator: Abusing the Windows 7/8 Low Fragmentation Heap. In *Hack in the Box 2012*, Amsterdam, Netherlands, May 2012.

[27] Simon_Z. *Efficacy of MemoryProtection against use-after-free vulnerabilities*. HP Enterprise, July 2014. http://community.hpe.com/t5/Security-Research/Efficacy-of-MemoryProtection-against-use-after-free/ba-p/6556134.

[28] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings E (Computers and Digital Techniques)*, 137(1):17–30, January 1990.

[29] Mate Soos. Cryptominisat, an advanced SAT solver, 2016.

[30] Chris Valasek and Tarjei Mandt. Windows 8 heap internals. In *BlackHat 2012*, Las Vegas, NV, July 2012. Slides: https://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf, Paper: http://illmatics.com/Windows\%208\%20Heap\%20Internals.pdf.

[31] Mark Yason. *Understanding IE's New Exploit Mitigations: The Memory Protector and the Isolated Heap*. IBM, August 2014.

[32] Mark Yason. *MemGC: Use-After-Free Exploit Mitigation in Edge and IE on Windows 10*. IBM, August 2015.

[33] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the 8th International Conference on Information and Communications Security*, ICICS'06, pages 379–398, Berlin, Heidelberg, 2006. Springer-Verlag.

Appendix A

Survey of Malloc Implementations

A.1 Apple (OS X and iOS)

Security researchers have paid attention to Apple operating systems, [24] but its memory allocator has received less attention. Some discussion appears in Miller and Zovi [20,21] in 2009 and 2012. The earliest public discussion seems to be from Nemo [23] in 2005.

As of 2016, Apple uses two different `malloc` implementations: `libmalloc` and `bmalloc`. `libmalloc` appears to date from Apple's acquisition of NeXT, while `bmalloc` is a brand new system, used in Apple's Safari browser. Both are distributed as open source on Apple's web site [2, 3]. `bmalloc` is also distributed as part of Apple's open source Webkit [4] (i.e., the open source project from which Apple's Safari browser is derived).

A.1.1 LibMalloc

`libmalloc` is implemented in C and exports a superset of the usual `malloc` routines. Notably, it has a concept of *zones*, allowing a program to allocate fresh memory from any zone. Under the hood, these zones are essentially the same as the *arenas* used in the glibc memory allocators. With Apple explicitly exposing these zones, a multi-threaded program can explicitly manage multiple zones to avoid locking contention when memory is allocated.

Zones also support *batch allocation*, facilitating efficient allocation of large numbers of objects of the same size. Apple's developer pages promote these and other `libmalloc`-specific features.[1] Apple also supports zone-specific allocation in its Co-

[1] `https://developer.apple.com/library/mac/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html`

coa Objective-C libraries[2] and apparently zone allocation is also used inside the newer Swift programming environment.[3]

The downside of `libmalloc`'s approach, however, is that alternative `malloc` implementations do not implement the various zone-related methods and are not drop-in compatible with programs or libraries that expect to have the extended `libmalloc` features.

`libmalloc` gives explicit credit to the Hoard allocator as an influence on its design. This analysis focuses on management of the "tiny" regions, used for small object sizes.

Tiny objects are allocated from *regions*. Each region is 1MB in size and holds 64520 16-byte *blocks*. A region also has a *region trailer* which contains the metadata for the region. The trailer includes a bitmask to track the free/busy state of each tiny block. When a block is allocated, it has no per-block header or trailer. Free blocks are maintained using a doubly-linked list.

Apple's design, in the presence of a use-after-free situation, potentially allows attackers to have access to the free-list pointers. Furthermore, if an attacker can overflow the last block in a region, the attacker will then have access to that region's metadata.

`libmalloc` maintains a checksum in its heap metadata, added sometime between 2009 and 2012, that defeats some heap metadata overwrite attacks. A checksum is stored in the high four bits of each free-list pointer, based on a randomly-generated cookie [20].

`libmalloc` also has a variety of flags that radically change how it operates, along with a related *Guard Malloc* library[4]. These help developers in finding memory errors in their programs. They have a serious impact upon performance. They could potentially be repurposed for security if their performance could be improved. Of note, `MallocGuardEdges` adds guard pages around large blocks, which will cause memory overruns to abort. Also, several environment variables instruct `libmalloc` to perform consistency checks on its heap. If corruption is detected, the process will abort.

A.1.2 BMalloc

Before 2015, the Webkit browser used Google's `tcmalloc`, then Apple abruptly replaced it with `bmalloc`.

The `bmalloc` implementation does not presently have the standard `malloc` API. Instead, a family of similarly-named calls ("mbmalloc", "mbcalloc", "mbfree", etc.) are provided, along with an internal switch that can route these calls to the "real" `malloc`

[2] `https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/ObjectAllocation/ObjectAllocation.html`

[3] `http://www.russbishop.net/swift-how-did-i-do-horrible-things`

[4] `https://developer.apple.com/library/ios/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html`

implementation or handle them internally. A program linked against `bmalloc` can then be switched over to a third-party `malloc` implementation. `bmalloc` does not expose anything analogous to the "zone" features of `libmalloc`.

`bmalloc` has a unique design. It could be best described as "lazy", in that when a block is freed, a pointer to it is placed in a fixed-size buffer. In the common case, allocation and free operations are constant time, with a very low cost.

`bmalloc` supports multiple *heaps*. A heap has *small pages* which in turn have *small lines*. `bmalloc` tracks the free space within these small lines. Then, for each of a variety of *size classes*, `bmalloc` maintains "bump allocators" which are backed by those small lines. Objects are allocated starting at the beginning of a line and continuing sequentially to the end.

Each heap has an asynchronous "scavenger" which processes its free lists, looking for memory it can reuse. If the scavenger gets behind, the normal allocation and memory freeing routines can also do the same work.

Unlike `libmalloc`, memory objects in `bmalloc` never hold metadata when they're freed. Instead of an inline free list, there is a counter in the header for each line of memory to track how much is in use versus how much is free. If the counter indicates a line of memory is entirely unused, it can be recycled for future allocations.

The metadata resides in headers at the front of each heap. The asynchronous nature of the free-list scavenger makes it harder for an attacker to predict the state of the heap at any given time, which is otherwise essential to Heap Feng Shui attacks.

`bmalloc` has no explicit features to detect use-after-free or double-free operations which might be exploited as part of an attack, but it has clearly been engineered to make some attacks more difficult.

A.2 Windows

Microsoft's Windows memory allocator has been the subject of regular attention from the hacking community. In response, Microsoft has developed particular security solutions for Internet Explorer and legacy systems going back to Windows XP. As part of their ongoing security efforts, Microsoft has also been improving the general resistance of the default `malloc` implementation that ships with Windows. Here, we focus particularly on the design in Windows 8.

A.2.1 Browser-specific memory allocators

Circa 2014, Internet Explorer introduced two new defenses [27, 31].

Memory protector Newly freed objects are zeroed out but are not immediately made available for reuse. Instead, a mark-and-sweep style system detects whether there are live references from the current call stack to any of these heap objects after they've been freed. This is effective at defeating "use after free" attacks.

Isolated heap This exposes multiple memory arenas to the user level, similar to Apple's `libmalloc` (see Section A.1.1). Operations that could potentially cross arenas, like a `realloc`, will fail. This allows Internet Explorer to isolate potentially attackable objects (e.g., aspects of the HTML DOM) from system objects. This could help defeat some classes of heap overflow attacks.

In Windows 10, Microsoft rolled out "MemGC", which provides additional memory protections for its Edge and Internet Explorer 10 browsers [32]. MemGC provides a concurrent mark-and-sweep garbage collector, which originates with Microsoft's JavaScript implementation, but which also serves the rest of the browser runtime system.

A.2.2 Enhanced Mitigation Experience Toolkit

Also in 2014, Microsoft released the fifth version of its Enhanced Mitigation Experience Toolkit (EMET5) [17]. EMET5 can be applied to versions of Windows as old as Windows XP, and has a variety of features, such as being able to prevent an application like Microsoft Word from loading an Adobe Acrobat plugin. To hinder heap attacks, EMET5 adds:

Heap spraying Preallocates blocks in regions that known heap-spray attacks like to use. (Trivial for new attacks to work around.)

Null page The page at the bottom of memory is pre-allocated to defeat attacks meant to trick the memory allocator into giving a reference to the zero / null address.

ASLR + DEP In addition to defending against a variety of other attacks, EMET adds some randomness to where `malloc`'s pages will be located.

A.2.3 Windows 8

The Windows 8 heap memory allocator is sufficiently complicated that any discussion here will necessarily be superficial.

The basic design of Microsoft's heap allocator is very similar to other arena-based allocators, with a front-end "low fragmentation heap" (LFH) that manages large numbers of small allocations, and a back-end that allocates memory from the operating system and feeds it to the LFHs, as necessary.

Where Windows Vista and earlier versions arranged free blocks as a doubly-linked list, Windows 8 now uses a bitmap, placing critical metadata out of the reach of attackers. Similarly, where metadata is still present, it is generally encoded (i.e., XORed with a constant), to make it hard for attackers to do anything useful.

The most comprehensive explanation and analysis of the Windows 8 heap allocator can be found in Valasek and Mandt [30]. They describe the structures and algorithms in great detail. Johnson and Miller [11] discuss ongoing mitigations within Microsoft's heap allocator. With specific reference to heap allocation attacks, a Microsoft technical report [19] summarizes the changes in Windows 8:

Heap integrity checks `malloc` and `free` perform checks on the metadata and terminate the computation if corruption is detected.

Fatal exceptions In older versions, when `malloc` or `free` had some sort of error, they would swallow it, possibly returning `null`, and thus giving attackers multiple attempts at any given attack. All errors are now immediately fatal.

Freeing metadata Hawkes [10] shows how a use-after-free attack, where the attacker can overwrite a pointer to the heap metadata, could then allow an attacker to attack the heap metadata structure itself. Windows 8 has specific logic in it to detect this attack.

Encoded metadata A function pointer in each heap header is XORed with a constant to make it unusable by an attacker. That constant is now relocated to where it's (hopefully) not easily read or written by an attacker. Encoding is also used to protect metadata for the "low fragmentation heap".

Guard pages All large allocations ($>$ 1MB on a 64-bit machine) go directly to the VM system, which creates a trailing guard page. Smaller allocations come from "heap segments" which also now have trailing guard pages. In some cases, these heap segments are broken up, nondeterministically, with additional guard pages.

Allocation ordering Allocations are also done in a a nondeterministic order, making it harder for an attacker to do Heap Feng Shui.

Seeley [26] observes that, while allocation order is randomized, the higher-level block allocations made from the operating system won't be. He estimates a lower-bound 50% chance of successfully performing a heap attack on Windows 8, while pointing out how much easier it would be on Windows 7 and earlier. Liu [14] makes similar observations.

Appendix B

Malloc With Separate Metadata

We now give more details about `sri-glibc` malloc. As all malloc-based memory management libraries, it manages system memory on behalf of client software. It gets memory from the operating system using system calls such as `sbrk` and `mmap`. System memory comes in large page-aligned blocks that the library splits into smaller sub-blocks called *chunks*. Client software obtains new chunks via calls to `malloc` and `realloc`, and releases chunks via calls to `free`. Glibc maintains various data structures, such as tables and lists, to manage these chunks.[1]

As we discussed previously, Glibc malloc attempts to minimize contention for multi-threaded applications by maintaining several independent heaps called *arenas*. In single-threaded applications, Glibc relies on a single main arena. In the multi-threaded case, each thread is transparently assigned a separate arena for its allocation.

B.1 Metadata

Most traditional malloc implementations (including all those listed in Figure 2.1) manage memory by keeping track of the start, size, status (e.g., in use, free, and mmapped), and other information about client memory. In current Glibc, the metadata is stored as a header in each chunk. It consists of the following fields:

```
struct malloc_chunk {
  INTERNAL_SIZE_T      prev_size;
  INTERNAL_SIZE_T      size;
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;
```

[1] An interesting feature of the Glibc family of allocators is that they maintain the invariant that there are no adjacent free chunks. This is achieved by coalescing adjacent free chunks.

```
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
};
```

The most important fields are `size` and `prev_size`, which store the size of a chunk and the size of the previous chunk, respectively. Because all sizes are rounded up to multiples of 16, the low-order bits of the `size` field store status information: whether or not the previous chunk is in use, whether it is mmapped, and whether it belongs to the main arena.[2] The remaining fields are used only for free chunks. They are pointers in various lists of free blocks, called *bins*. The metadata for a memory chunk at address `a` is stored in a header (at address $\texttt{a} - 16$).

Given an arbitrary client pointer, assumed to be the start of a valid chunk, glibc reads the header to determine whether the chunk is mmapped. If so, the chunk is handled directly by the operating system. Otherwise, the first step in all operations is to determine the chunk's arena and lock it. Arena indices are not stored explicitly in the metadata. Instead they are computed using offset calculations and pointer-chasing tricks.

Our prototype reuses most of the existing Glibc concepts, data structures, and algorithms. We only remove the metadata headers from the client chunks and move them into a separate region of memory (as shown in Figure 1.2). The main challenge is to replace the simple header read by an alternative lookup mechanism. Simple ideas such as a global hash-table are too inefficient to be practical.

Our solution is a two-step procedure. First, from an arbitrary pointer, we determine in a *lock-free fashion* the arena to which the pointer belongs. As an additional security feature, this first step detects whether a client pointer is the start of a valid chunk. Once the arena is determined, we lock it as in normal glibc, and access the chunk's metadata using a linear dynamic hash table (as in [13]). Each arena has its own local hash table that stores metadata for every chunk in that arena. The important property is that this scheme does not incur more locking overhead than the original glibc.

B.2 Testing and Benchmarking

We have validated all our prototypes on a set of applications that make heavy use of dynamic-memory allocation. Our primary tests include the Yices regression tests, benchmarks for Cryptominisat, a multi-threaded Boolean SAT solver, and the SPEC CPU 2006 integer benchmark suite.

Yices is an SMT solver developed by SRI [8]. It is written in C and includes more than 900 regression tests. We chose Yices as a testing target because we are in control

[2] On 32-bit machines, the sizes are typically multiples of 8, not 16, but the status fields are the same.

of the code and because it extensively uses `malloc`, `realloc`, and `free`. The Yices regression tests revealed bugs in existing malloc such as `dnmalloc` and in Schneider's implementation of Michael's lock-free malloc. All our prototypes successfully pass all Yices regression tests.

We selected Cryptominisat to exercise our prototypes in a multi-threaded application. Cryptominisat is an open-source SAT solver developed by Mate Soos [29]. It is a relatively large application written in C++. It is memory-intensive and can be run in multi-threaded mode. It is easy to test Cryptominisat using large problems available from the SAT solvers competitions. Many of them require significant runtime for Cryptominisat and require large numbers of calls to the malloc functions.

We used Yices and Cryptominisat primarily for validation and debugging. For more extensive performance measurements (and comparison with baseline), we ran the SPEC CPU 2006 integer benchmark suite [5]. This benchmarking amounted to several days of CPU time. We have completed benchmarking of the original GLibc malloc, of `psmalloc2`, of our 64-bit implementation of Maged Michael's malloc, and of preliminary versions of `sri-glibc`. Benchmarking of the final `sri-glibc` is ongoing. Results are shown in Tables B.1 to B.4. Each table follows the standard SPEC CPU format, together with a overhead percentage, and represents the average of 10 runs. The two important columns are the third and fifth. The third one displays the average run time per benchmark family in seconds, while the fifth displays the percentage difference between the third column of this malloc version compared to the base glibc version. These benchmark results are just preliminary ones, and should really just be used to get a general sense of the performance of each prototype.

B.3 Possible Improvements

From our profiling efforts on glibc, it was clear that significant amounts of overhead were going into metadata lookups. In traditional glibc's `malloc`, every block has its own metadata inline, just below the address of the pointer being given to `malloc`'s caller. Consequently, when `free` is called, all that `free` needs to do is subtract a constant from the pointer and it has all the metadata it needs. Because this metadata represents a security risk, wherein an attacker who can perform a heap overflow can overwrite this metadata, the primary goal of SRI's project was to relocate this metadata elsewhere. Consequently, the HeapMetadata project added a, per arena, hashtable to map from a heap pointer to its corresponding metadata structure.

Using this hashtable, relative to the original process of just doing some pointer arithmetic, can be a non-trivial cost when the number of live pointers grows into the millions. Benchmark applications which rapidly allocate and free very small objects are particu-

Benchmark	Base (s)	Run time (s)	Base Ratio	% Overhead (%)
400.perlbench	9770	271	36.1	0
401.bzip2	9650	378	25.5	0
403.gcc	8050	232	34.7	0
429.mcf	9120	200	45.6	0
445.gobmk	10490	374	28.0	0
456.hmmer	9330	354	26.3	0
458.sjeng	12100	429	28.2	0
462.libquantum	20720	318	65.3	0
464.h264ref	22130	443	49.9	0
471.omnetpp	6250	299	20.9	0
473.astar	7020	305	23.0	0
483.xalancbmk	6900	180	38.4	0

Table B.1: SPEC CPU Benchmarks: original GLIBC

Benchmark	Base (s)	Run time (s)	Base Ratio	% Overhead (%)
400.perlbench	9770	331	29.5	22
401.bzip2	9650	376	25.6	-1
403.gcc	8050	260	31.0	12
429.mcf	9120	208	43.8	4
445.gobmk	10490	373	28.2	-1
456.hmmer	9330	356	26.2	0
458.sjeng	12100	421	28.8	-2
462.libquantum	20720	315	65.8	-1
464.h264ref	22130	443	50.0	0
471.omnetpp	6250	423	14.8	41
473.astar	7020	307	22.9	0
483.xalancbmk	6900	513	13.4	185

Table B.2: SPEC CPU Benchmarks: psmalloc2

Benchmark	Base (s)	Run time (s)	Base Ratio	% Overhead (%)
400.perlbench	9770	517	18.9	90
401.bzip2	9650	403	23.9	6
403.gcc	8050	388	20.7	67
429.mcf	9120	227	40.1	13
445.gobmk	10490	403	26.0	7
456.hmmer	9330	386	24.2	9
458.sjeng	12100	472	25.6	10
462.libquantum	20720	327	63.4	2
464.h264ref	22130	489	45.3	10
471.omnetpp	6250	286	21.9	-5
473.astar	7020	313	22.4	2
483.xalancbmk	6900	156	44.2	-14

Table B.3: SPEC CPU Benchmarks: elfpa

Benchmark	Base (s)	Run time (s)	Base Ratio	% Overhead (%)
400.perlbench	9770	336	29.0	23
401.bzip2	9650	409	23.6	8
403.gcc	8050	259	31.1	11
429.mcf	9120	241	37.9	20
445.gobmk	10490	404	26.0	8
456.hmmer	9330	384	24.3	8
458.sjeng	12100	463	26.2	7
462.libquantum	20720	331	62.6	4
464.h264ref	22130	481	46.0	8
471.omnetpp	6250	437	14.3	46
473.astar	7020	334	21.0	9
483.xalancbmk	6900	232	29.7	28

Table B.4: SPEC CPU Benchmarks: sri-glibc

larly impacted by these costs.

Our first effort was to simulate a read-cache that sits in front of this hashtable. With as few as 16 entries in the cache, our simulations indicated a hit rate as high as 50% for these highly impacted benchmark applications. This suggested that a cache might be beneficial. Implementing this cache did indeed significantly reduce the CPU cache-misses encountered when running the impacted benchmarks, but *benchmark performance did not improve*. The cost of looping through the cache to find a hit turned out to add enough overhead to the cache misses to negate the benefits of cache hits.

We also constructed a read cache with only two elements. Our data showed that this much smaller cache would have a hitrate of approximately 28%, with code that would be much more efficient (e.g., avoiding branches by using conditional-move instructions). Nonetheless, benchmark performance again did not improve.

Lastly, the hashtable is implemented in the standard fashion where entries that map to the same bucket are stored as a linked list (also called "separate chaining"[3]). We hypothesized that when an entry is found in one of these linked lists, we could relocate it to the head of the list. This takes constant time, and would ensure that a "hot" entry would be found sooner. Performance measurements showed performance improvements of approximately 4% on the relevant benchmarks.

[3]Wikipedia has a nice discussion of how hashtable collisions are commonly handled. `https://en.wikipedia.org/wiki/Hash_table\#Collision_resolution`

Appendix C

Mostly Lock-Free Hashtable

Lock-free hash tables are a major component of our memory allocators. They enable the allocator to efficiently locate the metadata for a chunk of memory based on a pointer to this chunk (and to detect that a pointer is invalid if it has no associated metadata). In our glibc allocator implementation, the hash table efficiently maps pointers to independent memory heaps called *arenas*. It is clearly necessary to avoid locks when locating the arena for a memory chunk as locking would incur a major overhead in multi-threaded applications.

Implementing a fixed size lock-free linear-probing hash table is a relatively simple task. Many examples can be found online. An important aspect that is often not addressed is to allow the table to grow as more entries are added. This is key to ensure efficiency and low hash-collision rates. Growing the table leads to maintaining several verisons of the table: when the table grows, one must create a new larger, initially empty table and transfer the content from the current table to the new table. This creates a new difficulty: when and how can we reclaim the memory used by the old copy, in a lock-free fashion?

The algorithm we present here solves both problems, it is inspired by a paper by Tobias Maier [15], but is different in several important aspects.

C.1 Overview

Our hash table maps integral keys to integral values. The implementation supports insertion of new $(key, value)$ pairs, updates of the value assigned to an existing key, removal of a key, and search. Updates and insertions are implemented by a single *add* operation. These operations can be performed concurrently by multiple threads. In the typical case, none of these operations require obtaining a lock.

We use linear probing. The hash-table content is stored in a shared array. Search for a

key start sat some index i derived from the key's hash value then proceeds incrementally from i. Each entry in the array consists of two 64-bit integers or pointers. Updates to this array are performed atomically using compare-and-swap (CAS) operations. Our algorithm attempts to keep the array less than 70% full (this can be configured at compile time). When this threshold is exceeded, we allocate a fresh array of double the current capacity and migrate entries from the current array into the fresh array. Removal of an entry is done by storing a special value called a *tombstone* in the entry's value field. Entries with tombstoned values are not copied during the migration process.

Our hash table uses 64-bit keys and values. Table entries must be aligned on addresses that are multiples of four. These constraints are imposed by the CAS operations on Intel hardware. We also use the low-order bit of the keys as a mark.

The hash table consists of a descriptor and a linked list of dynamically-allocated arrays, called table headers. The first element in this list is the most recent copy of the table. The next element is the previous copy and so forth. In most cases, threads operate on the most current table (i.e., the first one in the list). When this table becomes full, we allocate a fresh array of double its size and add it at the front of the list. At this point, all threads that operate on the table are required to migrate entries from the old table to the new current table, the head of the list. A key invariant that our algorithm maintains is that migration is always from the second table to the first table in the list. In other words, the current table cannot become full before the previous table has been completely migrated. We say then that this older table is *assimilated*.

During the migration process, a thread that operates on the table must first copy a fixed number M of key/values pairs from the old table to the new table. The number M is chosen so that a new table cannot become full before the old table has been assimilated.

To avoid running out of memory, it is desirable to free old copies of the table that are no longer needed. But, because of thread scheduling, some threads may keep pointers to old versions of the tables (i.e., elements deep in the linked list). We can only free old copies when we are sure that no threads have pointers to them. We use reference counting for this purpose. Maintaining correct reference counts introduces race conditions that we could not eliminate with CAS. We then introduced a locking mechanism, which is used only in critical operations that are part of the migration process. Typical accesses to the table do not require locking.

C.2 Data Structures

The hash table consists of a *descriptor* of type `lfht_t` that stores a state, a pointer to the list of table headers, and a lock. The pointer to the list and the state are stored as a single 64 bit integer to allow us to atomically update them.

```
typedef struct lfht_s {
  uintptr_t hdr:62, state:2;
  pthread_mutex_t lock;
} lfht_t;
```

Each element of the list is a structure of type `lfht_hdr_t`. It stores an array of entries, each of type `lfht_entry_t`, counters and sizes, and a pointer to the next list element.

```
typedef struct lfht_entry_s {
  volatile uint64_t  key;
  volatile uint64_t  val;
} lfht_entry_t;

typedef struct lfht_hdr_s lfht_hdr_t;

struct lfht_hdr_s {
  atomic_uint_least32_t reference_count;
  atomic_bool assimilated;
  uint64_t sz;
  uint32_t max;
  uint32_t threshold;
  atomic_uint_least32_t count;
  lfht_hdr_t *next;
  lfht_entry_t *table;
};
```

The following three routines deconstruct the bit fields of an `lfht_t` object.

```
static inline lfht_hdr_t *lfht_table_hdr(const lfht_t* ht){
  return (lfht_hdr_t *)(uintptr_t)(ht->hdr << 2);
}

static inline lfht_state_t lfht_state(const lfht_t* ht){
  return ht->state;
}

static inline void lfht_set(lfht_t* ht, lfht_hdr_t * hdr, lfht_state_t state){
  ht->state = state;
  ht->hdr = (uintptr_t)hdr >> 2;
}
```

Even though we can atomically update the `state` and `hdr` fields of a table descriptor, we protect them by a lock. Critical operations use this lock to prevent concurrent modifications of the `state` and `hdr` fields. The lock is used when a handle is initialized, when the table grows, and when a thread scans the list to check whether old assimilated tables can be reclaimed. It is probably possible to implement the latter operation in a lock-free manner, but we have not found a way to avoid race conditions in the handle initialization and growing processes that relies only on CAS.

A table can be in one of four states:

```
typedef enum lfht_state {
  INITIAL,
  EXPANDING,
  EXPANDED,
  FINAL
} lfht_state_t;
```

These correspond to the following situations:

- INITIAL: The list contains a single, fresh table.
- EXPANDING: The migration process is in progress: there is more than one table, the current table is new and its predecessor contains key-value pairs that have not yet been copied into the new table.
- EXPANDED: The migration process has completed: there is more than one table, the current table is new and its predecessor has been assimilated. All key-value pairs in the predcessor table have been copied into the new table.
- FINAL: The table has been deleted.

As a first approximation, our hash table consists of a state and a pointer to the list header, the front of the linked list of tables. The question then, is "when can we release old tables?" Because of scheduling delays, we cannot ensure that all threads have a consistent, up-to-date view of the table. Slow threads may wake up and have pointers to old table headers and mistakenly treat them as the current table.

We solve this problem by using reference counting. All hash table operations must be made using a per-thread *handle*. The handle is a structure of type `lfht_handle_t`, and stores the identifier of the thread that owns it, a pointer to the table, a pointer to the handle's notion of the current table header, and the oldest table header that the thread has an interest in:

```
typedef struct lfht_handle_s {
  pthread_t owner;
  lfht_t *table;
  lfht_hdr_t *table_hdr;
  lfht_hdr_t *last;
} lfht_handle_t;
```

When a thread creates a handle to a table, it checks whether the table is in the expanding state. If so, it stores a pointer to the current table in `table_hdr` and to the previous table in `last`, and the thread increments the reference counters of both table headers. If the table is not expanding, the thread stores a pointer to the current table in both the `table_hdr` and the `last` fields, and it increments the current table's reference count.

As other threads access the table, the handle may become out of date: the `table_hdr` may not be the same as the first element in the hash table's list of headers. However, on creation, we must ensure that the handle points to the current table and that the reference counters are incremented atomically. Otherwise, a race condition could cause the handle to store invalid pointers to table headers that have been deleted concurrently by another thread. This is one of the critical sections that protect with a lock. A crucial invariant that we maintain is that a thread has incremented the reference count of every table header from the its current `table_hdr` to its `last` field.

Before any operation on the table, threads attempt to ensure that their handle is up to date, by updating the `table_hdr` and `last` pointers and adjusting reference counts. A thread that sets the reference counter of a table header to zero can scan the list to determine whether list headers can be deleted. The reference counters that we maintain are then the number of threads that point to a particular header through their handle (as opposed to the number of pointers to each table). Updates to these reference counters are then relatively infrequent, which is important for performance as they must be performed atomically.

C.3 Algorithms

We describe the process of initializing a handle, updating a handle, and using that handle to add an entry to the table. We also describe the growing of the table and the migration of entries from the previous table to the current. Creation of the table is routine and can be gleaned from the source code [16].

C.3.1 Handle Initialization and Update

Initializing a `lfht_handle_t` object is relatively simple. We set the `owner` and `table` fields to the appropriate objects. We then block until we have obtained the table's `lock`. Once we have the lock we can retrieve the current table header, increment its reference count, and assign it to the handle's `table_hdr` field. If the previous table header is not assimilated, we presume that the thread maybe required to migrate entries, and so we increment its reference count as well. Recording the fact that the `last` field points to the *last* `table_hdr` in the list whose reference count we have incremented. We must protect this operation by obtaining the mutex to ensure that the `actual` is in fact pointing to the current table header.

```
void init_handle(lfht_handle_t *h, lfht_t* table){
  lfht_hdr_t *actual, *next;
  h->owner = pthread_self();
  h->table = table;
```

```
  pthread_mutex_lock(&table->lock);
  actual = lfht_table_hdr(table);
  next = actual->next;
  atomic_fetch_add(&actual->reference_count, 1);
  if(next != NULL && !table_is_assimilated(next)){
    atomic_fetch_add(&next->reference_count, 1);
    h->last = next;
  } else {
    h->last = actual;
  }
  h->table_hdr = actual;
  pthread_mutex_unlock(&table->lock);
}
```

While we take care to ensure that upon initialization a thread's handle is up to date, we cannot maintain such an invariant. However the handle is designed so that it is particularly easy to determine if it is up to date or not, using the `upto_date` routine. A thread's handle is no longer up to date in the situation when the table has been grown by another thread. Once a thread determines that its handle is no longer up to date, it must rectify the problem. Because we have no control over thread scheduling, we cannot assume that the table has only been grown once. It may have grown several times. As a result updating a handle requires not only updating the `table_hdr` and `last` fields, but also making sure that the reference counts are accurate. This involves incrementing the reference counts of all headers that precede `table_hdr` in the list, and decrementing the reference counts of all stale table header. This is the reason why we include a `last` pointer in the handle. Since a handle is local to a thread, there are no risks of no race conditions when it is updated.

```
static bool upto_date(lfht_handle_t *h){
  return lfht_table_hdr(h->table) == h->table_hdr;
}

static void update_handle(lfht_handle_t *h){
  lfht_hdr_t *p;
  bool assimilated;
  lfht_t *table = h->table;
  lfht_hdr_t *actual = lfht_table_hdr(table);
  lfht_hdr_t *current  = h->table_hdr;
  lfht_hdr_t *last = h->last;
  int prior_count = 2;
  if(actual == current){
    return;
  }
  h->table_hdr = actual;
  p = actual;
  while(true){
    atomic_fetch_add(&p->reference_count, 1);
    p = p->next;
    if(p == current){ break; }
  }
  if(current == last){  return; }
```

```
  p = current->next;
  while(true){
    assimilated = table_is_assimilated(p);
    assert(assimilated);
    prior_count = atomic_fetch_sub(&p->reference_count, 1);
    if(p == last){ break; }
    p = p->next;
  }
  h->last = current;
  if(prior_count == 1){
    free_table_check(h->table, "update_handle", true);
  }
}
```

C.3.2 Addition

Adding a key-value pair to a table illustrates how we use the handle. We first compute the hash of the key. We then attempt to participate in any migrating that might be required. We will describe the migration process in detail shortly but two properties are important. First, the migration scans the old table starting from the hash of the key we are interesting in. If this key is present in the old table, the migration will then migrate it to the new opy. This ensures that the addition will happen after the right key value pair has been moved to the current table. Second, the migration process endeavors to ensure that the thread handle is up to date. Once we have performed our migration duties, we then proceed to add the key-value pair to the table by calling function `_lfht_add`. If this succeeds, and the handle is still up to date, we increment the table header's count, and check whether the table needs to grow. If `_lfht_add` succeeds, but our handle is no longer up to date, we have added to an old table. Function `_lfht_add` may also fail if the current table is full. In either case we need to update our handle and try again. If the table was full, we rely on another thread to grow the table. This is sound since all threads that successfully added to the table beyond its threshold will try to grow the table. One will succeed.

After a succesful addition to the most current table, we increment the table header's entry `count` and attempt to grow the table if this count is above the threshold.

```
bool lfht_add(lfht_handle_t *h, uint64_t key, uint64_t val){
  bool retval, current;
  lfht_hdr_t *table_hdr;
  uint_least32_t count;
  uint32_t hash = jenkins_hash_ptr((void *)key);
  while(true){
    _migrate_table(h, key, hash);
    retval = _lfht_add(h, key, hash, val);
    current = upto_date(h);
    if( retval && current ){ break; }
    if( ! current ){ update_handle(h); }
  }
  table_hdr = h->table_hdr;
  count = atomic_load(&table_hdr->count);
```

```
  if (count >= table_hdr->threshold){
    _grow_table(h->table, table_hdr);
  }
  return retval;
}
```

C.3.3 Growing the Table

A thread calls function `_grow_table` when it detects that the current table (as seen from the thread's hanlde) is full. Several threads may attempt to grow the same table concurrently, so we check whether the thread's handle is up to date. If not, this may indicate that another thread has successfully grown the table. The actual task of growing the table is relatively simple: we allocate a new table header that is twice the size of the current header [1] and we set it as the current header, and we change the table state to `EXPANDING`. We do this under the protection of the lock, not because of race conditions involved in the process (we could easily update pointer and state atomically with a CAS), but because other threads may be in delicate operations, such a initializing a new handle to the table, or scanning the table header list looking for tables to reclaim.

```
bool _grow_table(lfht_t *ht,  lfht_hdr_t *hdr){
  lfht_hdr_t *ohdr, *nhdr;
  bool retval = false;
  uint32_t omax, nmax;
  pthread_mutex_lock(&ht->lock);
  ohdr = lfht_table_hdr(ht);
  if(hdr == ohdr){
    omax = ohdr->max;
    if (omax < MAX_TABLE_SIZE) {
      nmax = 2 * ohdr->max;
      nhdr  = alloc_lfht_hdr(nmax);
      if (nhdr != NULL){
        nhdr->next = ohdr;
        lfht_set(ht, nhdr, EXPANDING);
        retval = true;
      }
    }
  }
  pthread_mutex_unlock(&ht->lock);
  return retval;
}
```

C.3.4 Migration

The routine `_migrate_table` attempts to move `MIGRATIONS_PER_ACCESS` key-value pairs from the old table header to the new table header. The routine is sensitive

[1] Since our main application is a memory allocator, we use `mmap` to allocate table header memory.

to the possibility that the table state may change. The workhorse of this process is subroutine `assimilate` which returns the number of key-value pairs actually moved. If this number is less than the requested amount, we can safely assume that the table has been fully migrated. In this case we atomically update the `assimilated` field of the old table header, and, while holding the lock, we update the state of the stable from `EXPANDING` to `EXPANDED`.[2] As mentioned earlier, when moving key-value pairs from the old table header to the new table header, we start the migration at the hash of the key of the API operation that triggered the call to `_migrate_table`.

```
static int32_t _migrate_table(lfht_handle_t *h, uint64_t key, uint32_t hash){
  int table_state;
  lfht_t *ht;
  lfht_hdr_t *hdr, *ohdr;
  uint32_t moved;
  bool finished, assimilated;
  int32_t migrated = -1;

 restart:

  update_handle(h);
  ht = h->table;
  table_state = lfht_state(ht);
  if (table_state == EXPANDING){
    hdr = lfht_table_hdr(ht);
    ohdr = hdr->next;
    if (!upto_date(h) ){ goto restart; }
    if(hdr == h->last){ return migrated; }
    moved = assimilate(h, ohdr, key, hash,  MIGRATIONS_PER_ACCESS);
    finished = moved < MIGRATIONS_PER_ACCESS;
    assimilated = false;
    migrated = moved;
    if (finished  &&  atomic_compare_exchange_strong(&ohdr->assimilated, &assimilated,
        true)){
      pthread_mutex_lock(&ht->lock);
      if (lfht_state(ht) == EXPANDING  && lfht_table_hdr(ht) == hdr){
        lfht_set(ht, hdr, EXPANDED);
        free_table_check(h->table, "_migrate_table", false);
      }
      pthread_mutex_unlock(&ht->lock);
    }
  }
  return migrated;
}
```

The `assimilate` routine starts out at the slot corresponding to the `hash` and attempts to move the `key` and its associated value to the new table if it is present and is not tombstoned. Along the way it also tries to move `count - 1` other non-assimilated

[2]Race condition: not requiring that the thread updating the table state holds the lock, introduces a nasty race condition, where the thread being a bit slow actually toggles the state from `EXPANDING` to `EXPANDED` for a migration that has just commenced, not the previous one that just finished.

non-tombstoned key-value pairs. The actual move is done by the routine `_lfht_move`, which we will describe shortly. After a successful move, a key is marked as assimilated in the old table (by setting the key's low order bit). Because the mark happens after the move, several threads may try to move the same key-value pair but only of them will succeed. Marking keys before the move would avoid this issue (as then a single thread would attempt the move) but this causes a more subtle problem. Moving an entry from the old to the new table is not atomic and a thread can be interrupted during the process. Such a thread could mark the last key to be migrated then go to sleep before completing the copy. In this situation, all keys are marked in the old table and no other thread will find anything to migrate. More and more entries may then be added to the new table before the old table header is marked as fully assimilated. Marking keys only after the move completes alleviates this problem.

```
static uint32_t assimilate(lfht_handle_t *h, lfht_hdr_t *from_hdr, uint64_t key,
    uint32_t hash,  uint32_t count){
  uint32_t retval, mask, j, i;
  lfht_entry_t current, *table;
  uint64_t akey;
  bool success = false
  retval = 0;
  mask = from_hdr->max - 1;
  table = from_hdr->table;
  if (table_is_assimilated(from_hdr)) {
    return retval;
  }
  j = hash & mask;
  i = j;
  while (true) {
    current = table[i];
    if (current.key == 0 || key_equal(current.key, key)) {
      success = true;
    }
    if (current.key == key ||
        (retval < count && current.key != 0 && !key_is_assimilated(current.key))) {
      if (current.val != TOMBSTONE){
        _lfht_move(h, current.key, jenkins_hash_ptr((void *)current.key), current.val);
      }
      akey = set_assimilated(current.key);
      if(cas_64((volatile uint64_t *)&(table[i].key), current.key, akey)){
        retval ++;
      }
    }
    if (success && retval >= count){
      break;
    }
    i++;
    i &= mask;
    if ( i == j ){
      break;
    }
  }
  return retval;
}
```

C.3.5 Adding Entries

We now describe the two operations that perform linear probing in the underlying table: `_lfht_move` and `_lfht_add`. The move operation is the workhorse of `assimilate`. It is simpler than `_lfht_add` because it never fails: it either successfully moves a new key-value pair or does nothing if the key it is trying to move is already present in the new table. The presence of this key in the new table indicates that another thread did the move previously. Function `_lfht_add`, on the other hand, must handle both the caase of a replacement (i.e., chaging the value associated with a key that is already present), and the situation where the key has been marked as assimilated. This can happen if the table has grown while this thread was interrupted and asleep, and other threads have been busily migrating keys in the meantime.

Both the add and move operations take a handle, a key, the hash of the key, and the desired value. They commence traversing the array at the position associated with the hash, looking for an empty slot, or a slot that is already populated by the same key (or an assimilated variant in the case of `_lfht_add`). If an empty slot is found, then both operations perform a 128 bit CAS to replace the key and value in one atomic action. If the key is already present, then the `_lfht_move` operation simply returns. The `_lfht_add` operation performs a 128 bit CAS if the key is not marked as assimilated, otherwise it fails. Updating the key-value pair atomically (using a 128-bit CAS) is important to ensure consistency when the table is growing.[3]

```
static void _lfht_move(lfht_handle_t *h, uint64_t key, uint32_t hash, uint64_t val){
  uint32_t mask, j, i;
  lfht_entry_t*  table, current, desired;
  lfht_t *ht = h->table;
  lfht_hdr_t *hdr = lfht_table_hdr(ht);
  table = hdr->table;
  mask = hdr->max - 1;
  j = hash & mask;
  i = j;
  desired.key = key;
  desired.val = val;
  while (true) {
    current = table[i];
    if (current.key == 0){
      if (cas_128((volatile u128_t *)&(table[i]), *((u128_t *)&current),  *((u128_t *)&
          desired))){
        atomic_fetch_add(&hdr->count, 1);
        goto exit;
      } else {
        continue;
```

[3]Race Condition: if we did a 64 bit CAS of the key followed by 64 bit CAS of the value, then key-value pairs can get lost. This happens when one thread is adding a new key value pair to the table, but is operating on the old table. Once it adds the key, another thread that is migrating values could look at the key value pair, deduce that it has been tombstoned, and mark it as assimilated. In this way the new value can be lost.

```
      }
    }
    if ( current.key ==  key ){
      goto exit;
    }
    i++;
    i &= mask;
    if (i == j) break;
  }
  assert(false);
 exit:
}


static bool _lfht_add(lfht_handle_t *h, uint64_t key, uint32_t hash, uint64_t val){
  uint32_t mask, j, i;
  lfht_entry_t*  table, current, desired;
  lfht_t *ht = h->table;
  lfht_hdr_t *hdr = lfht_table_hdr(ht);
  bool retval = false;

  table = hdr->table;
  mask = hdr->max - 1;
  j = hash & mask;
  i = j;
  desired.key = key;
  desired.val = val;
  while (true) {
    current = table[i];
    if (current.key == 0){
      if (cas_128((volatile u128_t *)&(table[i]), *((u128_t *)&current),  *((u128_t *)&
          desired))){
        atomic_fetch_add(&hdr->count, 1);
        retval = true;
        goto exit;
      } else {
        continue;
      }
    }
    if ( key_equal(current.key, key) ){
      if( key_is_assimilated(current.key) ){
        retval = false;
        goto exit;
      } else {
        if (cas_128((volatile u128_t *)&(table[i]), *((u128_t *)&current),  *((u128_t
            *)&desired))){
          retval = true;
          goto exit;
        } else {
          continue;
        }
      }
    }
    i++;
    i &= mask;
    if (i == j) break;
  }
 exit:
  return retval;
}
```

Appendix D

Modeling and Verification of Concurrent Migration