SLIMalloc II: Making C Safer than the "*Memory-Safe*" Languages

January 2023

"He who controls the allocator, owns the system!" - Anonymous

Pierre Gauthier, CEO
pierre@trustleap.com

TWD Industries AG (1998)
TWD Holding AG (2009)

Abstract

In **June 2020**, SLIMalloc [1] delivered the most advanced features available (1) enabled on-the-fly at runtime on a per-heap basis, (2) *real-time* invalid-pointer detection blocking memory access violations, (3) locating and reporting memory allocation and system errors in source code and binaries: applications, libraries or syscalls, (4) detection, location and remediation of all memory leaks, (5) trace of all allocations, even in third-party code, (6) a smaller source-code, (7) higher performance despite more features, (8) a far more constant execution time, (9) a low memory usage (10) three times less kinds of padding, all located at unpredictable addresses, (11) resource usage (RAM, CPU, network, disk, etc.) monitoring with SVG charts of all tracked processes, (12) a signal handler that documents and *recovers* fatal system errors to let *unharmed* applications decide of their faith – instead of continuing with corruption eventually leading to a crash or hack.

In **November 2020**, the industry then stated [30] that we must replace the *free* 50-year old C language by much slower, less-capable, "*memory safe*" languages (written in C, like the OS they use) because: "*Using C and C++ is bad for society, bad for your reputation, and bad for your customers*."

In **November 2022**, the "*NSA recommends that organizations use memory safe languages*". [28]

As fast as the (unsafe) allocators, **SLIMalloc now makes C** "***memory safe***" to preserve and leverage (1) 50 years C software investments, (2) strategic engineering skills, and (3) unmatched performance.

Keywords: Energy savings, IoT, System on Chip, Datacenter, Software Engineering, Security, Privacy, Troubleshooting, Operating System, Memory Allocation, Malloc.

Introduction

We all use a lot of third-party code (OS, libraries, runtimes, compilers, CPUs, etc.) but one is special [17]: "*Google estimated that 90% of Android vulnerabilities are memory safety issues.*" [30]

Security fails by ignoring *Turing award winner* Ken Thompson's reminder of a 1973 DoD report:

- *"No amount of source-level verification will protect you from using untrusted code."* [2]
- "*If an error in an operating system program allows a penetration program to work, that program will work every time it is executed – without detection.*" [6]

1. All programs involve a memory allocator even before they start running their own code,
2. all applications, libraries, and even operating systems have to rely on a memory allocator,
3. OS and application remote updates can (and do) replace code before and after it is executed.

With flawed *by-design* OS memory allocators, what could possibly go wrong? [18] [21] [22] [32]

Denying it costs $183Bn yearly ($538Bn in 2030) [5] in pointless expenses: with flawed allocators cyber-security vendors can only sell ineffective cures, and OS vendors ever-growing vulnerabilities.

Stealth Remote Code Injection (non-technical readers can jump to the next page)

The ***Out-Of-Bounds*** (OOB) and ***Use-after-free*** (UAF) vulnerabilities, both part of ***2019-2022 CWE Top 25 Most Dangerous Software Weaknesses*** [4] illustrate how unexpected chains of consequences take place – when some predetermined circumstances are met.

It can easily escape the most rigorous source-code audits [3] [10] since NONE, PART or ALL of the three steps below can take place OUTSIDE of the application source code. The faulty third-party code just makes "benign" errors (integer cast/type/overflow/truncation/sign):

```
// (1) allocate memory (done by our code, a third-party library, the system)
c->request = malloc(REQUEST_SIZE); ...

// (2) conditionally free memory (done by our code, a library, or the system)
int error = validate_user_request(c); // likely to be using libc functions

// (3) a freed block may be re-used (by our code, a library, or the system)
//     to modify memory (overflow, use-after/arbitrary/double/-free...)
// ===> hijack program execution flow
// modify memory at/near pointer address to redirect program execution by
// overwriting a function pointer, running commands, triggering a DoS,
// crash, exit, restart, bogus signal handler, etc.
void log_and_send_error(connection_t *c)
{... free(c->request);
 ... c->reply = malloc(REPLY_SIZE);
 ...
}
```

"Benign" Undetected Errors leading to Memory-Related Program Execution Flow Violations:

Violation: Integrity **Consequence**: Modify Memory
An OOB `write` (string format/integer/API-call/etc. error) or using uninitialized or previously freed memory may corrupt valid data (return address, application objects, buffer size, GOT (Global Offset Table), function pointer, memory allocator free-list, signal/exception handler, etc.).

Violation: Availability **Consequence**: DoS (Crash, Exit, or Restart)
After an OOB `write` or if the memory allocator performs block coalescence (and/or reallocation) after previously freed data has been modified when it was reused, ***then*** the program may crash due to invalid data being used by the memory allocator for this block's accounting data.

Violation: Integrity, Confidentiality, Availability **Consequence**: Arbitrary Code Execution
After an OOB `write` or after previously freed data has been maliciously modified, ***then*** the program execution flow may be hijacked to execute arbitrary code (input-injected shell code, call to application or library functions, or a string of commands to execute via `system()` or `exec()`).

Summary

An OS, third-party library, programming language runtime or design can compromise perfectly written applications – just by accidentally doing "benign" bugs and careless memory allocations.

These undetected vulnerabilities are exploitable only because all memory allocators neglect security:

- *"At the advantage of program execution speed"* [31] – StackOverflow.com
- *"70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues."* [11] – Microsoft Security Response Center
- "*Apple's T2 Security Chip has an unfixable flaw allowing hackers to disable macOS security features and install malware."* [12] – Wired.com

In 2020 SLIMalloc has proved the three above statements wrong.
In 2023 SLIMalloc makes C applications, libraries, and the system "*memory-safe*".

This document demonstrates how the ***memory allocator defines an operating system's security and performance***, discussing what can be done beyond SLIMalloc to leave today's sorry current state of inefficiency and insecurity caused by this poorly designed and so overlooked central OS component.

Java and .Net are "*memory-safe*" and yet much more unsafe than C [34]. Being "m*emory-safe*" is not enough to resolve today's cyber-security chaos. Since an OS is written in C, only C can detect and block system-wide malicious activity (like SLIMalloc does since 2020).

Today's ubiquitous vulnerability has very tangible economic and strategic consequences for:

- **Critical Infrastructure** (energy grid, drinkable water, telecoms, transportation),
- **Automotive** (remotely spied, hijacked and stolen cars, charging stations, etc.),
- **MedTech** (health sensors, insulin pumps, pace-makers, medical appliances),
- **FinTech** (payment/trading/exchange/compensation devices and platforms),
- **Defense** (all connected equipment is vulnerable, can be used against owners).

SLIMalloc will be licensed on favorable terms to any organization willing to widely deploy it.
As time goes, SLIMalloc will add new game-changer features (like ubiquitous network security).

The only way to ensure stable long-term fruitful exchanges is to guaranty everyone's outcomes.

I. SECURITY

"*protection of a person, building, organization or country against threats such as crime, criminals and attacks by foreign countries.*"
(Cambridge International Dictionary of English, 1995)

I. SECURITY / 1. What *"memory safe"* means for SLIMalloc, for C, and for all others?

For SLIMalloc, it means that the deadly C code below becomes 100% safe – without source code modification or recompilation – just by using SLIMalloc (which accelerates programs as a bonus):

Faulty Source code	Result

`// --- CASE 1 ----------------------`
`// Note: getchar() would avoid the`
`//  malloc() and the gets()`
`//  security issue. A failed`
`//  malloc() will crash, else`
`//  gets() might corrupt memory:`
`//`
`bool ask_yes_no(char *quest)`
`{`
`  char *res = malloc(8); // no check!`
`  printf("%s? [y] n: ", quest);`
`  gets(res); // res[] OOB`
`  bool r = res[0] != 'n'; // crash`
`  free(res); // crash/corruption`
`  return r;`
`}`
 	
`// --- CASE 2 ----------------------`
`// Note: strncpy() could avoid the`
`//  potential memory corruption`
`//  - if properly used (misused`
`//  it creates the same issue):`
`//`
`int main(int argc, char *argv[])`
`{`
`  char *p = calloc(8); // no check!`
`  strcpy(p, argv[1]);  // crash/OOB`
 	
`// --- CASE 3 ----------------------`
`// Note: here a server uses a wrong`
`//  (uninitialized, confused,`
`//  overflowed) buffer size`
`//  leading to a possible crash`
`//  and/or memory corruption:`
`//`
`int  fd  = accept(fd, NULL, NULL);`
`int  len = atoi(argv[1]); // variable`
`char *p  = malloc(100);  // fixed`
`read(fd, p, len); // crash, p[] OOB`
	--- Without SLIMalloc ---------------
	If the keyboard, command-line-args, network-input length given by LibC or the kernel is NULL or larger than the allocated buffer, a read/write memory access violation results that can cause memory corruption leading to a crash or, if it alters control-flow, to (remote) arbitrary code execution.
	
	--- With SLIMalloc -----------------
	Before an OOB can corrupt memory, it is detected and blocked. A warning is emitted (using a user-defined file descriptor which, for example, can be stderr or a log file).
	Depending on the heap->opt.abort flag the process continues unharmed (with or without running a user-defined callback) or it aborts.
	
	WHEN pointers belong to the reachable process address-space, detecting and blocking memory access violations DOES NOT slow-down programs. ELSE, a SIGSEGV is raised, due to invalid pointers or when a guard-page is hit (in which cases the signal handler will slow down the program further).
	
	SLIMalloc automated errors reporting and remediation slow-down programs a bit. But getting the ROOT CAUSE of errors is certainly better than a mysterious crash or undetected probes and remote code execution exploits.
	
	Continuing unharmed is key: servers can close the current connection OR stop and restart a faulty thread and keep the other threads running.

SLIMalloc automatically blocks 70 to 90% of the root causes [11] [12] [28] of past and future vulnerabilities exploited by criminals [17] [30] – immensely reducing the attack surface.

This security covers most of the C/C++ errors that happen inside **and outside** of your code: in **third-party library** calls, **OS usermode (LibC)** functions, and **OS usermode syscall interfaces**, a task that other "*memory-safe*" programming languages, incompatible with C, cannot accomplish.

For C, being fully "*memory safe*" implies an automated coverage of all the possible cases, like OOBs and arbitrary writes on global, stack or heap-allocated variables, on C structure fields (or C++ class members), and on SLIMalloc-unmanaged memory areas (`sbrk/mmap` areas, custom allocators, etc.).

Late 2022, the NSA has explained how and why memory safety may involve slow-downs:

> "*Memory safety can be costly in performance and flexibility. There is also* ***considerable performance overhead*** *associated with checking the bounds on every array access that could potentially be outside of the array.*" [28]

Avoiding this "*considerable overhead*", like SLIMalloc does as seen later, is vital for datacenters (energy costs) and for the IoT at the network Edge (costs of high-volume hardware deployments).

The so-called "*memory safe*" languages that are not even half "*memory safe*"

All applications and libraries have to call "black-box" CPU and OS functions to use a computer (to access disks, displays and networks adapters, CPU instructions) and all of this code is executed outside of the scrutiny of the (often incompatible with C) "*memory safe*" language runtimes.

> As *Turing award winner* Ken Thompson (who has later worked for Google) wrote:
> *"You can't trust code that you did not totally create yourself. No amount of source-level verification or scrutiny will protect you from using untrusted code."* [2] (1984)

The "*memory safety*" concept is sound, but ignoring most of the executed code leads to disastrous consequences: to prevent abuse "*memory safety*" must be enforced in applications and the system.

Operating systems are written in C so only SLIMalloc can enforce "*memory safety*" in applications and system-wide: other "*memory-safe*" languages only focus on applications using *their language*.

In this sorry state of things, SLIMalloc covers more ground than any other solution by:

1. reporting where memory violations took place – inside and outside of your code;
2. intercepting program execution before memory corruption can cause harm, but letting programs continue running unharmed instead of crashing or aborting;
3. documenting violations on-the-fly (with the decompilation of the offending code if needed) because OS/App updates may inadvertently erase the smoking-gun;
4. protecting applications, libraries, and the system while accelerating their execution;
5. offering an alternative: the main operating systems, Web browsers and memory allocators are created by a bunch of companies and shareholders based in one single country. This makes it difficult for foreigners to feel that their needs are always taken into account.

SLIMalloc is not making everything bullet-proof, today nothing can. But in less than 2 years it has delivered unique pro-active capacity – and, if we get the support we need, much more will come.

I. SECURITY / 2. Real-Life Undetected Damage

How fragile and easy to abuse are the "*most advanced*" memory allocators? Let's see that:

Buffer overflow

```
char sz = 8,  n = 5, *p[n]; // the code below is run twice to do free + malloc
for(int i = 0; i < n; i++) p[i] = malloc(sz); // allocate blocks
memset(p[0], 'A', n * sz);                    // overflow first block
for(int i = 0; i < n; i++) printf(" %d %p '%s'\n", i, p[i], p[i]); // print
for(int i = 0; i < n; i++) free(p[i]);        // free blocks
```

It's a texbook case: we read data (network, database, multimedia file, keyboard, command-line...) that overflow the storage buffer and overwrite what follows – and we call `free()` and `malloc()` again.

If OS/library code can corrupt your data then "*memory-safe*" languages are helpless. And if a block contains a `function pointer`, then this will lead to (probably remote) arbitrary code execution.

GLibC malloc (freelist/blocks corruption detected at free)	Block offset: 16 bytes in 132 KB Area

```
0 0x19b8010 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
1 0x19b8030 'AAAAAAAA'
2 0x19b8050 ''
3 0x19b8070 ''
4 0x19b8090 ''
*** Error in `./oob': free(): invalid next size (fast): 0x00000000019b8010 ***
Aborted
```

GLibC has never segregated its metadata, exposing it to OOB attacks. So, in-place block-accounting is corrupted by the undetected overflow until corrupted metadata is accessed by `free()`, an operation which fails due to corruption, hence the long-awaited overdue `abort()`. This protection can easily be bypassed... but, shockingly, this is much better than another allocator taking more detours to check if its integrity has been compromised – and two other allocators that do not even bother to check.

JEmalloc (undetected overflows, blocks corruption)	Block offset: 4 MB in 8 MB Area

```
0 0x7f425a61d000 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
1 0x7f425a61d008 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
2 0x7f425a61d010 'AAAAAAAAAAAAAAAAAAAAAAAA'
3 0x7f425a61d018 'AAAAAAAAAAAAAAAA'
4 0x7f425a61d020 'AAAAAAAA'

0 0x7f425a61d020 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
1 0x7f425a61d018 'AAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
2 0x7f425a61d010 'AAAAAAAAAAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
3 0x7f425a61d008 'AAAAAAAAAAAAAAAAAAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
4 0x7f425a61d000 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
```

JEmalloc "segregates" its block `freelist` so, after a successful and undetected buffer overflow, the second allocation pass has reused the freed blocks in reverse-order – without detecting anything wrong. As the last block of the first allocation-pass became the first re-used by the second pass, the overflow coverage has doubled. Freeing and reallocating overflowed blocks is undetected *by-design* (and by choice since no checks are ever done). Why bother, right?

TCmalloc (undetected overflows, blocks corruption)	Block offset: 8 MB in 9 MB Area

```
0 0x1dce008 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA8��#'
1 0x1dce010 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA8��#'
2 0x1dce018 'AAAAAAAAAAAAAAAAAAAAAAA8��#'
3 0x1dce020 'AAAAAAAAAAAAAAA8��#'
4 0x1dce028 'AAAAAAA8��#'

0 0x1dce008 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ8��#'
1 0x1dce010 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ8��#'
2 0x1dce018 'ZZZZZZZZZZZZZZZZZZZZZZZ8��#'
3 0x1dce020 'ZZZZZZZZZZZZZZZ8��#'
4 0x1dce028 'ZZZZZZZ8��#'
```

TCmalloc "segregates" its `freelist` metadata and overflows are undetected. It re-uses freed blocks in their natural order so the second buffer overflow replaces the first overflow. The second pass freeing and re-using overflowed blocks is also undetected. There are no integrity checks. Google is not afraid of memory errors, maybe because that's not its data that are compromised [17] [30].

MIMalloc (undetected OOB, blocks+freelist corruption)	Block offset: 128 KB in 64 MB Area

```
0 0x4b040020080 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
1 0x4b040020088 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
2 0x4b040020090 'AAAAAAAAAAAAAAAAAAAAAAA'
3 0x4b040020098 'AAAAAAAAAAAAAAAA'
4 0x4b0400200a0 'AAAAAAAA'

0 0x4b0400200a8 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
1 0x4b0400200b0 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ'
2 0x4b0400200b8 'ZZZZZZZZZZZZZZZZZZZZZZZ'
3 0x4b0400200c0 'ZZZZZZZZZZZZZZZZ'
4 0x4b0400200c8 'ZZZZZZZZ'
```

MIMalloc keeps using in-place `freelist` metadata (like GLibC). No freed block are re-used because it keeps allocating new space for (much) longer than our loop. Not reallocating blocks is why the two overflows are undetected even after two free/malloc loops – and despite the fact that MIMalloc uses compiler options designed to catch buffer overflows [35] as recommended by the NSA [28].

Enabling MIMalloc "secure" options (encoded `freelist`, random blocks) requires a recompilation:

MIM "secure" (undetected OOB, blocks/free corruption)	Block offset: 0-4 KB in 63 MB Area

```
0 0x2bbec020580 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAZU"
1 0x2bbec020500 ''
2 0x2bbec020940 ''
3 0x2bbec0207c0 ''
4 0x2bbec020400 ''

0 0x2bbec020e00 'ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZuU"
1 0x2bbec020dc0 ''
2 0x2bbec020a40 ''
3 0x2bbec020280 ''
4 0x2bbec020a00 ''
```

Overflows and `freelist` corruptions are also undetected by MIMalloc "secure". Like with GLibC, metadata corruption will cause an `abort()` only after a new allocation meets a corrupted `freelist`. Microsoft just delays the checks made by `free()` and `malloc()` with a pseudo-random numbers generator to pick a new random block when `malloc()` is invoked. Detecting corruption long after damage has been done lets attackers to their job and hide their tracks to stay undetected.

All these allocators could detect corruption with instant integrity checks. They just don't do it. As a result, application/system buffer overflows and metadata corruption are... undetected.

Memory corruption can generate immediate consequences (crash, control-flow hijacking) or long-delayed visible effects (crash, garbage state and output). Primary causes are increasingly more difficult to find as they get older – hidden behind smoke and mirrors ("*garbage in, garbage out*").

Then developers are blamed. In reality, they might be totally innocent – the OS allocator is guilty:

SLIMalloc (on-the-fly detection, blocking and reporting – no overflow, no corruption)

```
> OOB: memset() accessed:40 block-size:8 OOB:32
  caller ./oob.c:9 main()
0 0x41e254c6a700 'AAAAAAAA'
1 0x41e254c6a708 ''
2 0x41e254c6a710 ''
3 0x41e254c6a718 ''
4 0x41e254c6a720 ''

> OOB: memset() accessed:40 block-size:8 OOB:32
  caller ./oob.c:14 main()
0 0x41e254c6a720 'ZZZZZZZZ'
1 0x41e254c6a718 ''
2 0x41e254c6a710 ''
3 0x41e254c6a708 ''
4 0x41e254c6a700 ''
```

```
Here heap->opt.rnd_block
= false so block-address
randomization is not
active (all blocks are
allocated with their
natural order and
increment - instead of
at random addresses) but
the result is the same
in both cases.

Changing options is done
on-the-fly, on a per-
heap basis, without
recompilation.
```

SLIMalloc is the only allocator to (1) traverse the first allocation/free pass unharmed, and to (2) complete the second allocation/free pass without damage, by (3) detecting, blocking, locating and reporting the exact location and precise cause of the OOB memory access violation on-the-fly.

JE/TC/MIMalloc "segregate" metadata in their block-area... exposed to undetected OOBs, defeating the very purpose of segregated metadata, which was initially to serve security. [14]

50 years ago, C/*asm* programmers had the technical background to understand the implications:

> "*If an error in an operating system program allows a penetration program to work, that program will work every time it is executed – without detection.*" [6]
> – Electronics Systems Division, Air Force Systems Command, Computer Security Developments Summary, report MCI-74-1 (1973)

The flawed memory-allocator vendors (claiming to be the root cause of 70-90% of all vulnerabilities for decades) now want to secure the world with unsafe [34] "*memory-safe*" languages. Ahem, really?

I. SECURITY / 3. Secure Allocators vs Security Layers

"*LLVM ASan*" aims for correctness while "*GWP-Asan*" is merely for probabilistic verifications. SLIMalloc does systematic security checks – *while accelerating programs:*

	LLVM ASan (shadow-ram)	**Valgrind** (VM)	**Dr. Memory (Google, MIT)**	**GPerftools** (allocator)	**SLIMalloc** (allocator)
Technology(4)	CTI (compile-time instrumentation) "systematic" (yet many cases are bypassed)	DBI (dynamic binary instrumentation)	DBI (dynamic binary instrumentation)	TCmalloc "probabilistic" heap-checker library	SLIMalloc "systematic" detect and block (or recover) errors
Slowdown vs Acceleration(2)	2x slowdown	20x slowdown	10x slowdown	25%-50% slowdown	30% to 5x acceleration!
Detects and **Blocks**					
Heap OOB out-of-bounds	most / no	yes / no	yes / no	some / no	most / yes
WWW write-what-where	some / no	no / no	no / no	no / no	some / yes
UAF use-after-free (dangling ptr)	yes / no	yes / no	yes / no	some / no	yes(3) / yes
Leaks	yes / no	yes / no	some(1) / no	yes / no	yes / yes

(1) for Dr. Memory, non-freed memory is not a leak; only blocks missing a pointer are leaks.
(2) slowdown or acceleration, as compared to the imposed default system memory allocator.
(3) all UAFs are caught if application/library is recompiled with SLIMalloc (no APIs involved).
(4) "systematic" doesn't mean exhaustive, it means that checks are not merely "probabilistic".

Blocking memory errors *on-the-fly* lets programs avoid corruption, continue unharmed instead of aborting (when a violation is detected) or crashing, often delayed (with ignored violations). It lets SLIMalloc document bugs and attacks (not mere core-dumps) and even retaliate in real-time.

This is invaluable in production, when input-related errors are triggered and nobody is watching. It also helps if you don't want to crash/fix/recompile a program 10,000 times to find all its bugs.

All security layers have blind areas and almost all cause a "*considerable performance overhead*" (including the `Intel MPX` technology [15] that the `GCC` compiler documented and made available for a while as an alternative to the `ASan libraries`) – hence, maybe, Google's "*probabilistic*" approach to reduce its costs... at the expense of security.

Only a very few (continuously and generously funded) can afford to waste considerable resources. To stay alive, the rest of us, mere mortals, have no choice but to count every cent they spend. SLIMalloc lets everybody spend and risk less while doing more.

I. SECURITY / 4. Allocator Features and Options

All memory allocators allows end-users to tune their behavior by changing options:

- **Facebook JEmalloc** by editing a global */etc/malloc.conf* configuration file (which has lower priority than the per-binary setting, which gets lower priority than the */etc* setting, which gets lower priority than the environment settings) – on the top of an API that requires modifying and recompiling the application. There are no security options.

- **Google TCmalloc** by modifying and recompiling your application to use its API. Security options require to use external libraries that deliver memory leaks collection, guard-pages and canaries.

- **Microsoft MIMalloc** requires to use its API and recompile the application. Some API calls related to security seem confusingly redundant until you try them:
  mi_heap_contains_block(heap, p); mi_heap_check_owned(heap, p); mi_check_owned(p); mi_is_in_heap_region(p);
  Security options are enabled with the MIMalloc ***secure*** flag at compile time.

- **SLIMalloc**'s main security features can be changed ***on-the-fly*** at runtime and on a ***per-heap*** basis (by just changing a flag variable), for a third-party library call, any portion of your code, or even dynamically to match changing system conditions (high/low CPU/RAM workload, application or network attack, monitoring, reporting, retaliating, etc.). Some options, if undesired, can be disabled at compile time.

SLIMalloc's core secure allocator uses less than 2,000 lines of C code. The rest of its code (almost 8,000 lines of code) is for the many SLIMalloc features that other allocators don't offer:

Allocator	Language	blank-lines	comment-lines	code-lines
JEmalloc	C, C++	9,954	11,383 (18% of code)	62,708
TCmalloc	asm, C, C++	8,917	12,417 (23% of code)	54,068
MIMalloc	C	2,359	3,421 (27% of code)	12,461
SLIMalloc	C	1482	7,638 (76% of code)	9,995

This table only includes each company's source code used by its memory allocator (without security for JEmalloc and TCmalloc). It does not include system libraries or third-party libraries (like *libunwind*, used by TCmalloc and, as an option, by Jemalloc).

TCmalloc relies on the Google Abseil C++ library (159,540 lines of code – without blanks and comments) to which it may add the "*probabilistic*" GWP-Asan library (not used in our study).

Now you may wonder if there's a way to compare these code-bases in terms of functionalities. That's what the next table attempts to do ("*the documentation is the source code*" – and it's long).

Security Features and Troubleshooting Tools (GLibC "security" not listed: slow, obsolete, non-thread-safe, see [1])

Feature	JEmalloc	TCmalloc	MIMalloc	SLIMalloc
Spot invalid pointers (automatically, and/or with public function)	no (crash) limited to misaligned pointers	no (crash) limited to misaligned pointers	no (crash) limited to misaligned pointers in areas, because MIMalloc's dedicated functions are too slow, and therefore unusable to protect the allocator and applications	yes real-time, fault source code location, warn, abort or continue unharmed; (real-time function available for developers to identify valid and freeable pointers), see [1]
Block memory allocation errors	some (corruption, crash) limited to misaligned pointers in block areas	some (corruption, crash) "GWP-ASan" canaries (corruption detection after free for blocks < 8 KB), otherwise limited to misaligned pointers	some (corruption, crash) limited to misaligned pointers in block areas	yes real-time, preventive, fault source code location, abort or continue unharmed, see [1]
Out-Of-Bounds (OOB) read/write errors (buffer overflow or buffer underflow)	no (corruption, crash)	some (corruption, crash) "GWP-ASan" is limited to guard pages and canaries (corruption detection after free for blocks < 8 KB)	some (abort, crash) very partial support in slower "secure mode", poor reporting	yes warning, real-time, preventive, fault source code location, abort or continue unharmed with user-defined callback
Write-What-Where (WWW) errors	no (corruption, crash) no security for access in allocator padding (metadata seems to be properly segregated)	no (corruption, crash) no security for access in allocator metadata and padding	no (corruption, crash) no security for access in allocator padding and metadata "segregated" at a known offset(!)	yes allocator padding is very scarce and located at random addresses (like allocator metadata which is segregated)
Double-free() Invalid-free() Invalid-realloc()	yes (corruption, crash) abort/warn, limited to misaligned pointers	yes (corruption, crash) abort/warn, mostly limited to misaligned pointers	yes (corruption, crash) abort/warn, limited to misaligned pointers	yes real-time, preventive, fault source code location, warn, abort or continue unharmed with user-defined callback, see [1]

Use-after-free (UAF) protection	no (corruption)	some “GWP-ASan” does it with *“low probability”* detection via canaries (blocks < 8 KB) and guard pages	no (corruption)	yes real-time, preventive, full-coverage if code recompiled with SLIMalloc (no API calls involved)
Canaries (corruption detection at free() step)	yes called ”redzones”, constant or junk-filled, slow	yes “GWP-ASan” canaries (limited to blocks < 8 KB)	yes “10% overhead” with “secure” mode	yes very fast, for all blocks, encoded, fault source code location, see [1]
Guard pages (density is key: that's how many mines you have in a minefield; the more you have, the most effective)	no	yes “GWP-ASan” provides user-defined density (default: every 100M bytes / 24,415 OS pages)	yes “10% overhead” secure mode; fixed density: 1 guard-page after a 4 MB mimalloc-page / 1,024 OS pages	yes user-defined density (can also be random per class-size) default: every 360 KB / 90 OS pages
Segregated metadata	yes (at an offset in the allocated-blocks area, exposed to OOB attacks)	yes (at an offset in the allocated-blocks area, exposed to OOB attacks)	some (not freelists) (at an offset in the allocated-blocks area, exposed to OOB attacks)	yes random by-design (not stored in the allocated blocks area)
Address randomization	no	no	yes at free()	yes at malloc()
Zero-memory on malloc	yes but only once: not with realloc() if ptr != NULL	no	no	yes option can be enabled for any portion of the code, see [1]
Zero-memory on free	no	no	no	yes option can be enabled for any portion of the code (password verification, secret key management, etc.)
Delayed memory reuse	no	no	yes via delayed free()	yes picking random blocks
Memory leak detection and remediation	no	some “heap-checker” prints leak information with stack traces of leaked objects' allocation	no but a function can traverse all remaining memory allocations at a given time	yes very fast, reports block address and size in memory, locates calls in source code

		sites; “heap profiler” helps finding functions that allocate a lot of memory	(after-the-facts – so the related malloc API calls in the code are no longer reachable) see [1]	(address, visible/debug symbol names, line numbers), dedicated list and repair functions, see [1]
Signal handler (with system fault reporting, ***recovery***, bypass options, and user-defined callback)	no no signal handler; profiling has support for collecting backtraces	no a signal handler offers a callback to *store* backtraces	no no signal handler	yes real-time, with fault location in source code and automatic remediation, the VMA (Virtual Memory Area) type: code, stack or heap, along with data block range, access rights, section, block-size, hexdump; warn, abort or continue unharmed with user-defined callback, see [1]
Constant execution time	no consecutive runs of the same task lead to very variable execution times, even without any concurrent background tasks	no consecutive runs of the same task lead to very variable execution times, even without any concurrent background tasks	no consecutive runs of the same task lead to very variable execution times, even without any concurrent background tasks	yes consecutive runs of the same task lead to very constant execution times – even under various system loads; very low variability
Profiling, statistics, text charts (without dependencies)	yes profiling API	yes profiling API	yes statistics API	yes text charts with block size breakdown of used/free blocks; allocation tracing brings total freedom to implement user-defined schemes, see [1]
Benchmarking, monitoring, charting (without dependencies)	no	no	no	yes without instrumentation, for any binary even with different allocators, in one pass with several series, saves data files, generates statistics and SVG charts

I. SECURITY / 5. The Nature of the Persistent Problem

Today's ever-increasing cyber chaos mandates action:

> *“Is memory safety relevant? In 2017, 55% of remote-code execution (RCE) causing bugs in Microsoft due to memory errors.”*
> – Kanad Sinha and Simha Sethumadhavan, Columbia University, New York, USA

But replacing C by much less capable programming languages (written in C, like the OS they use) will not resolve the problem [34]. The OS needs a few critical rewrites [29] as a few players, having received far too much of a good thing [18] [29] [33], are no-longer trustworthy [36].

The *nature of the menace* has been documented by U.S. department of Defense – 50 years ago – far before the *“cyber-security”* expression was invented:

> “*If an error in an operating system program allows a penetration program to work, that program will work every time it is executed – without detection.*” [6]
> – Electronics Systems Division, Air Force Systems Command, Computer Security Developments Summary, report MCI-74-1 (1973)

And the cyber-security industry is equally affected (hence its recurring failures):

> *"Security mechanisms can be maintained only if both the operating system* [...] *and its operational files are protected from unauthorized modification. [Otherwise] it would be possible for an experienced user to modify the operating system and circumvent the security mechanisms without the likelihood of detection."* [6]
> – National Computer Security Center, Final Evaluation Report of Computer Security Corporation Sentinel, CSC-EPL-87/004 (1987)

Until a truly independent international competition is made possible again and trustworthy compilers and CPUs prevent OS/application memory-management from being by far the ***most important and persistent root cause*** [11] [12] [28] [30] of all vulnerabilities, not much will change because today's OS vendors and their customers at the government find an interest in the *status-quo* [7] [8] [9].

For decades, and now daily, we hear about “new” exploits taking advantage of “new” bugs. The state of vulnerability is ever-expanding – like if the largest best-funded world vendors could not learn from their mistakes, and like if software could not be corrected as time goes.

> In reality, very little is “new”, and this chaos exists only because a very few primary causes are not addressed by the ever-growing (security-irrelevant) software patches and updates.
>
> Small software publishers usually lack the experience and skills, but their application *bugs* aren't the *cause* of the chaos – they merely are the entry-doors to the OS vulnerabilities.

For the first time, an *effective* solution is made available and SLIMalloc's promise is to improve continuously at protecting the system and all applications – a rare value-proposition in such an overcrowded, over-funded and heavily government-subsided market [14] [16] [24] [25] [26] [27].

II. SPEED

"*rate at which something moves or happens.*"
(Cambridge International Dictionary of English, 1995)

II. SPEED / 1. Why Not Just Compare Apples to Oranges?

Most memory allocator benchmarks ignore the question of available and enabled features – and what each feature or option actually does for each product. Why bother, right?

We planned to use TCmalloc *"GWP-ASan"* and MIMalloc *"secure"* to have only two allocators without security options (GLibC and JEmalloc) and three delivering various levels of security.

TCmalloc + *"GWP-ASan"* (a *light* version of "*LLVM ASan*" involving a 2x slowdown), provides:

- ***guard-pages*** (protecting against *read/write* access violations... 1 byte every 100 million),
- ***canaries*** (detecting *write* access violations at the *free()* step... "*only for blocks < 8 KB*"):

  *"GWP-ASan is only capable of finding a **subset** of the memory issues detected by ASan. Furthermore, GWP-ASan's bug detection capabilities are only **probabilistic**."*

  "*GWP-ASan has limited diagnostic information for buffer overflows within alignment padding, since overflows of this type will not touch a **guard page**. For write-overflows, GWP-ASan will still be able to detect the overflow during deallocation by checking whether **magic bytes** have been overwritten*".

TCmalloc and MIMalloc were so slow with security features that we had to disable them. So, in our benchmarks, all memory allocators (all but SLIMalloc) lack security options:

- **GLibC** its security add-ons are too old, too slow, and not thread-safe [1],
- **JEmalloc** by choice, little is done for security (or other features) to favor speed,
- **TCmalloc** Google GperfTools lacks the (purposely limited) *"GWP-ASan"* features,
- **MIMalloc** the secure "*performance penalty of 10%*" was much higher in our tests,
- **SLIMalloc** delivers by-design "*memory safety*" features that cannot be disabled.

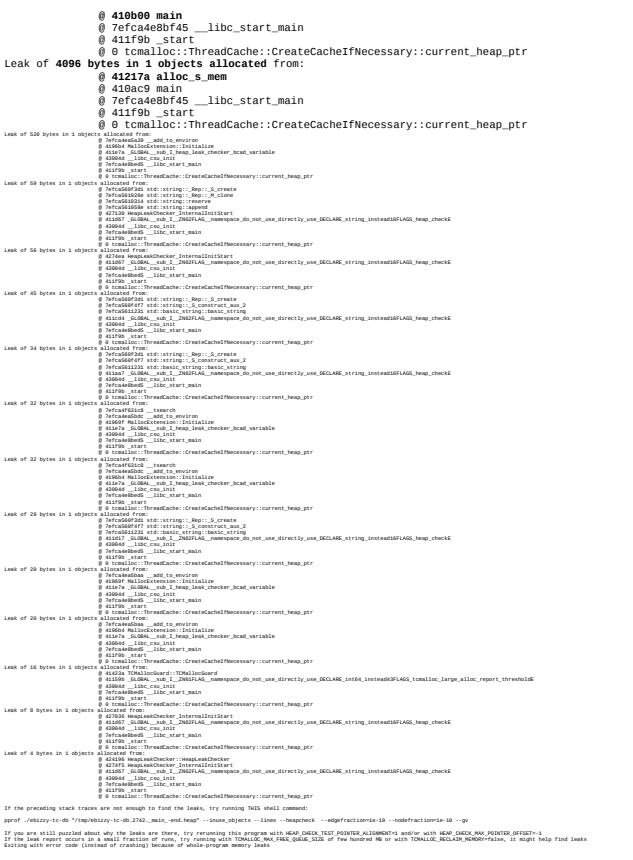
Even boring "common features" (below ***memory leaks***) may deliver quite different value:

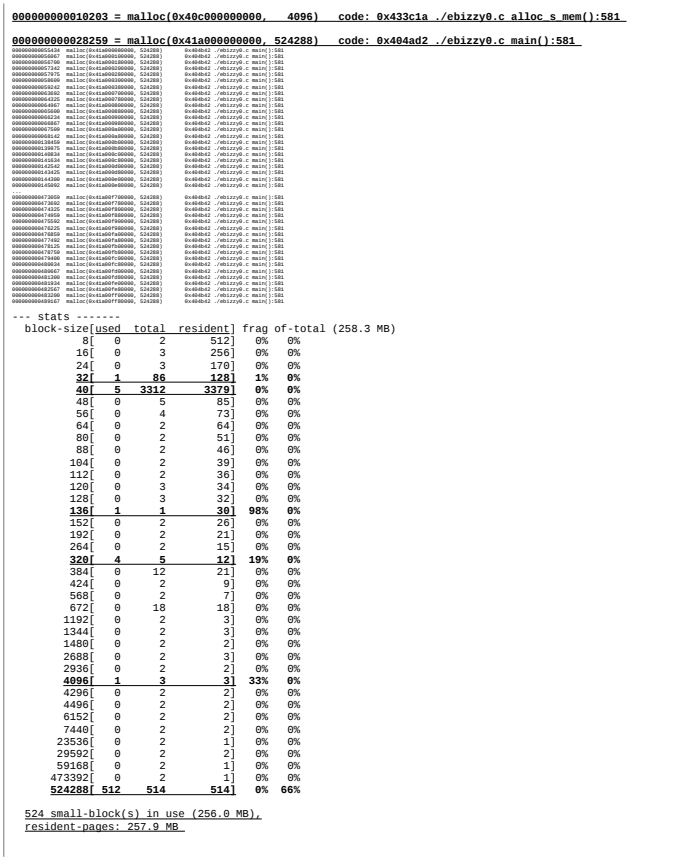
TCmalloc GPerftools (heap-checker):
- 20% slower than without tracing leaks,
- sums 512 leaks (without giving the block size!),
- sorts reported leaks by size (losing chronology!),
- lists (arguably pointless) C++ stacktraces,
- limits the report to "*The 15 largest leaks*"!?

```
Intel Ebizzy benchmark
TC_ALLOC
-----------------------------------------------------------------
- 17270 records/s
- total time: 13.897 seconds
- user CPU time ............... 16.276 sec (16.276 per thread)
- system CPU time ............. 43.926 sec (43.926 per thread)
- RSS, current real RAM use ........ 296861696 bytes (283.1 MB)
- RSS peak ........................ 298471424 bytes (284.6 MB)
- page reclaims ................ 125855424512 bytes (117.2 GB)
- voluntary context switches ..... 142034 (threads waiting, locked)
- involuntary context switches ..... 2683 (time slice expired)

Not looking for thread stacks; objects reachable only from there will be reported as leaks
Leak check _main_ detected leaks of 268440427 bytes in 526 objects
The 15 largest leaks:
Using local file ./ebizzy-tc-db.
Leak of 268435456 bytes in 512 objects allocated from:
```

SLIMalloc:
- as fast as not tracing leaks 2x faster than TC,
- lists source-code line numbers and file names,
- lists malloc API function/arguments/return-value,
- lists all 512 blocks, with their address and size,
- reports the leaks as they happened (not sorted),
- lists block count / size + in-use / freed / resident,
- lists the (properly identified) TLS GLibC leaks.

```
Intel Ebizzy benchmark
SLIMalloc heap[0].opt(265): abort guardpages:90 trace
-----------------------------------------------------------------
- 37169 records/s
- total time: 6.457 seconds
- user CPU time ............... 0.409 sec (0.409 per thread)
- system CPU time ............. 0.104 sec (0.104 per thread)
- RSS, current real RAM use ....     296042496 bytes (282.3 MB)
- RSS peak ....................     296042496 bytes (282.3 MB)
- page reclaims ...............     281739264 bytes (268.6 MB)
- voluntary context switches ... 14 (threads waiting, locked)
- involuntary context switches . 1831 (time slice expired)

--- trace -------
000000367282984 = calloc(0x404800000140,    320)
   code: 0x7ff4769adee5 /build/eglibc-xkFqqE/eglibc-2.19/elf/dl-tls.c _dl_allocate_tls():296
000000312699508  calloc(0x404800000280,    320) 0x7f44e177eee5 /build/eglibc-xkFqqE/eglibc-2.19/elf/dl-tls.c _dl_allocate_tls():296
000000312742744  calloc(0x4048000003c0,    320) 0x7f44e177eee5 /build/eglibc-xkFqqE/eglibc-2.19/elf/dl-tls.c _dl_allocate_tls():296
000000312784144  calloc(0x404800000500,    320) 0x7f44e177eee5 /build/eglibc-xkFqqE/eglibc-2.19/elf/dl-tls.c _dl_allocate_tls():296
```





TCmalloc reports require the `Gperftools heap-checker library`, the use of `command line arguments` and the setup of `environment variables`. You will probably get it wrong the first times you try to use this very verbose feature (hence its "*15 largest leaks*" truncated output).
It seems that it was designed to limit memory leaks rather than to eradicate them. A design choice that *may* make sense but which is not necessarily *your* choice as a memory allocator end-user.

Despite being generated with a lower CPU and RAM overhead, SLIMalloc (leak-tracing or allocation-tracing) reports have been designed to help you resolve problems:

- When needed, you enable leaks-tracing with `heap->opt.trace = true;` (at run time) and you can then call `print_trace(stdout, heap);` at any time to generate a report, and/or `free_leaks(heap);` or even `get_trace(heap);` to manually process the list.

- With the reported source-code line numbers, the malloc API function name (`malloc, calloc, realloc`, etc.), the requested and allocated block sizes and input and output addresses (ie: `outptr = realloc(inptr, size);`) of all the memory leaks you can easily add an `if(...) {...}` test in your code to understand why any given leak takes place (even on a list of millions of allocated blocks).

- The "***used / total / resident***" count per block-size lets you to evaluate the proportion of leaks and regular allocations, and find how much "*resident*" memory they actually occupy at a given time. That's the same purpose as the "*15 largest leaks*", but with a complete global picture of all your heaps' memory usage – not a limit imposed by the tool.

- The "*fragmentation and of-total*" percentages give you insights about how important any class-size of memory leaks (or allocations) is for the program – at any given time.

- With the ability to trace all memory leaks (or allocations) from within your code, you can decide which portion(s) of your program to trace (instead of searching forever in the gazillions of tracked memory allocations – or missing the one you are searching because it is not among the *"15 largest memory leaks"* reported by your leak-tracing tool).

- SLIMalloc lets you enable/disable the randomness of block addresses (at run time, on a per-heap basis – unlike Linux Address Space Layout Randomization enabled at compile time by `gcc -fPIE`, for Position Independent Executable) so that debugging and tracing are much easier (block addresses are always the same – when you need it).

- The SLIMalloc leaks-tracking feature also lets you traverse (print, export, free) the list of leaks, on-the-fly, at any point in time, globally or on a per-heap basis, for the whole program or just a chosen part... to take action *at run time* (and not only after the facts).

Speed matters, but usability has this unique power to make your daily life great or miserable.

All features should automate everything that can be done for end-users so they don't have to try searching inaccessible information, and waste their time in endless conjectures.

The ***trimming option***, the only "common feature"(!) of all allocators, is enabled because:

1. the less tasks you accomplish, the fastest you are, so enabling the only common feature that all allocators support is fair and informative (reflecting design and code quality).
   But, here again, some do much less than others – yet they call it by the same word.

2. TCmalloc does it with *./configure --enable-aggressive-decommit-by-default*, JEmalloc always purges unused areas (lazy trimming), and others can enable/disable it at runtime: GLibC and MIMalloc via global options, and SLIMalloc via a per-heap flag [1].

3. trimming reveals how much memory (and how fast) each allocator is able to release to the OS during and after a large workload – a key decision criteria when several applications share a single machine (databases and caching servers are memory-hungry).

Yet, to check if SLIMalloc performs well whatever the enabled options, we will also make the same benchmarks without trimming.

So, as you have seen now, comparing things is difficult – because you need enough knowledge to know how different are the things you want to compare.

II. SPEED / 2. Finding a Pertinent Test (and Redefining “Pertinence”)

We need a benchmark capable of isolating the CPU and memory related parts of a server process ***while excluding all the other bottlenecks***: TCP/IP stack, unrelated OS functions, etc.
We must measure only what we want to test – not use tests that introduce new bottlenecks!

The 2007 **Ebizzy** benchmark has been written and documented by **Intel Corp** (a CPU vendor):

> *“Ebizzy is designed to replicate a common web search application server workload. A lot of search applications have the basic pattern:*
>
> 1. *get a request to find a certain record,*
> 2. *index into the chunk of memory that contains it,*
> 3. *copy it into another chunk, then, look it up via binary search.*
>
> *The interesting parts of this workload are:*
>
> - *large working set,*
> - *data alloc/copy/free cycle,*
> - *unpredictable data access patterns.*
>
> *The records per second should be as high as possible, and the system time as low as possible.”*

Simulating Search Engines, Web application servers, Cache servers and Database servers should make a lot of sense for the largest datacenter operators (here Google, Microsoft and Facebook).

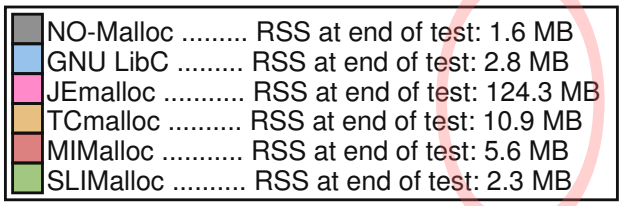
Over the years, Intel Ebizzy has been considered so relevant that it has been included in:

> *“The Linux Test Project, a joint project started by SGI, developed and maintained by IBM, Cisco, Fujitsu, SUSE, Red Hat and others, has a goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux."*

Now we have a pertinent benchmark, we must measure the Ebizzy overhead without memory allocations in its processing loop:

> To avoid GLibC allocator interference we allocate all memory upfront with *mmap()*, a tactic that fits Intel Ebizzy but not all benchmarks (because their processing loop may dynamically allocate much more memory than available in the system).
>
> This delivers an efficiency difficult to match for general-purpose allocators that have to allocate and free memory repeatedly in the *multi-thread* Ebizzy processing loop!
>
> This reference test will show us (1) the cost of dynamic allocations, and (2) how much better (or worse) each allocator handles instruction-cache and data-cache thrashing [19].
>
> Beyond the code and data working sets, OS-page fragmentation alone may cause TLB thrashing if the VM working set does not fit into the Translation Lookaside Buffer (TLB).
>
> Result: smaller allocator implementations get a boost if they are at least as fast (or faster).

This ***"bare to the bone" reference*** (the state of available kernel memory and activity is variable and beyond our usermode control), is now one of the data series of all our Ebizzy charts:



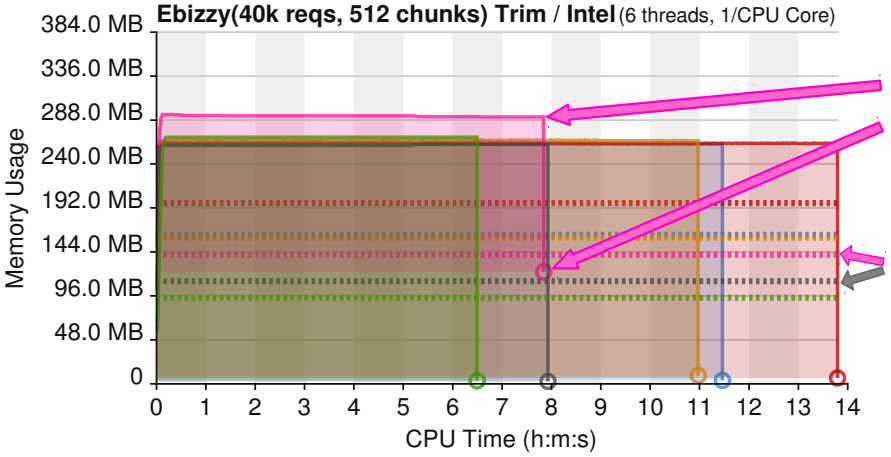
In our charts, a tail vertical data line, if any, indicates a ***how much memory programs have released at the end of a test*** – the RSS (Resident Set Size – actual memory usage), highlighted by a circle for better visibility.

We highlight this released memory with a per-allocator semi-transparent colored area (all the space left blank below this area represents the remaining memory still used by the program).

Given `n` = number of samples of an allocator, `N` = number of samples of the slowest allocator, `s` = sum of *n* memory usage samples, and `r` = ending RSS, the dotted horizontal lines show:

```
(s + (r * n/3)) / (N + n/3); // scoring speed+memory-usage (ignoring security)
```

When they do no trimming, TCmalloc and MIMalloc are faster by **up to 33%**. This also explains why JEmalloc always does merely ~half of SLIMalloc and GLibC's trimming:



To gain speed, JEmalloc ***consumes*** more RAM and ***releases*** much less RAM than all others.

Our dotted-line scores ***reflect*** this with a better rank for NO-Malloc than for JEmalloc (despite a very similar execution time – *this time*).

Trimming has substantial value – when done completely and efficiently.
But it has less value ***if*** the trimming you are doing slows-down the OS and applications [19].

The same goes for security (which must be ***real*** and ***efficient***).

What matters is to have the choice, so users can pick the best performance, security, and features.

Yet, the other memory allocators tested here are all based on the same defective design – hence their similar defects hurting performance, on the top of their common security gaps.

And these problems persist in 2023, despite the publication of SLIMalloc in 2020.

If "the markets" ensure the success of the worst products, users must put their money elsewhere.

II. SPEED / 3. Finding a Relevant Unit of Measure

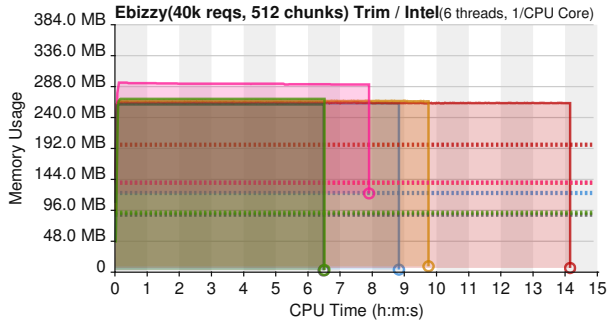
All programs have variable execution times but some allocators are less constant than others, already "without background tasks" – and even more when the machine workload is raised by a demanding task (like re-encoding a video, a process using 80% of all the CPU Cores for hours).

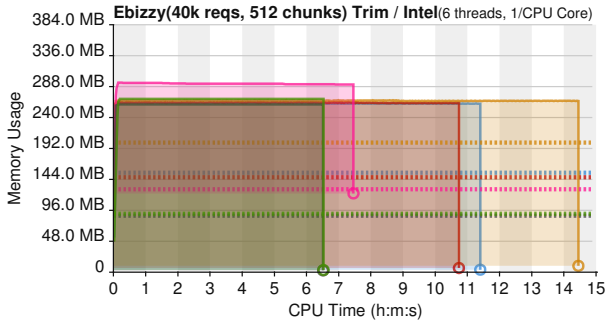
Yet, benchmarks ignore this parameter – like if computer programs were *perfectly* constant and isolated from interferences (previous and concurrent workloads, OS kernel changing states, etc.).

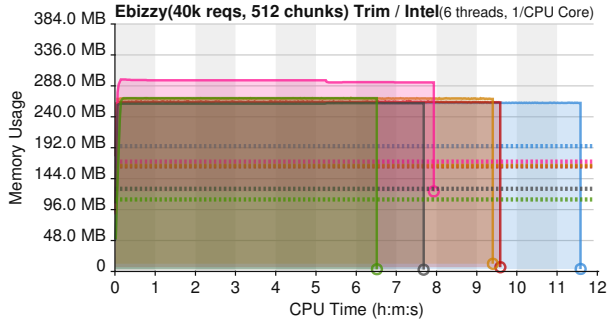
If ten consecutive benchmarks cannot provide the same exact results, then, certainly, another unit of measure is required to reflect what is happening, in the real life.

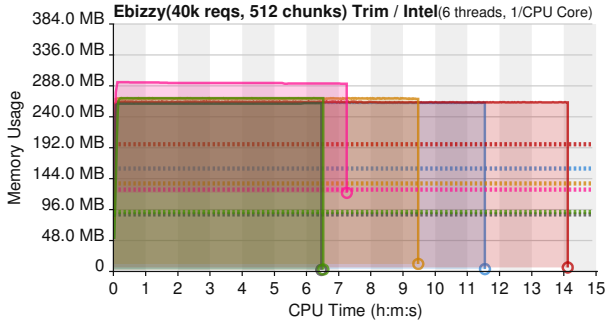
That's what ***statistics*** [13] provide – and they require ***a fair amount of (long-enough) test runs***.

These charts show the execution time (and memory consumption) for the Ebizzy test without, and then with background tasks (for the last two). Most ***vary a lot*** – so their ***profile*** can be built:

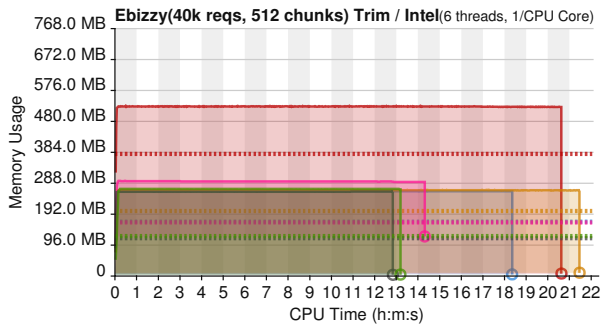


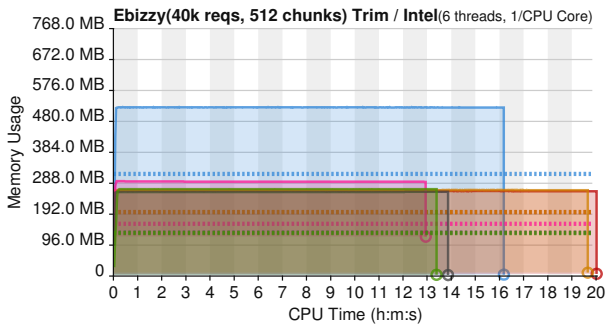






Tests with CPU/RAM intensive ***erratic background tasks*** generate equally ***erratic results***:

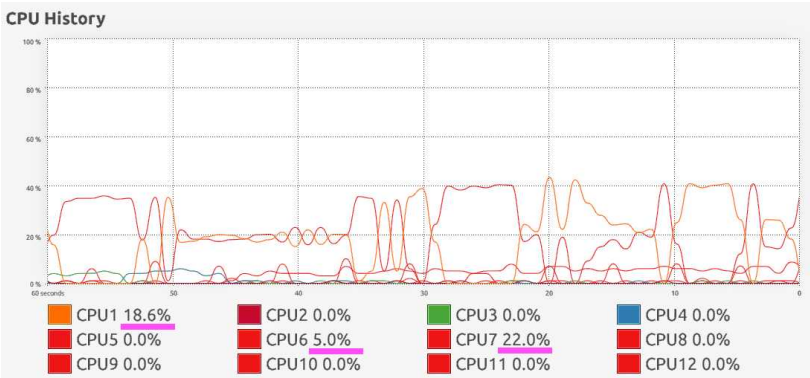




To check if we could reduce these variations, we have gradually increased the delay between each program launch – the results are slightly different but don't change the allocators behaviors:
(six consecutive 10 test-runs of all 6 allocators, total duration: 60 minutes, no background tasks)

```
0 ms     ebizzy-libc  mean: 8.678 median: 8.745 max-min:4.222 variance:1.313213 ave-dev:0.6146 std-dev:1.1460
         ebizzy-mim   mean:12.324 median:12.394 max-min:1.914 variance:0.347761 ave-dev:0.4564 std-dev:0.5897
         ebizzy-tc    mean:10.488 median: 9.939 max-min:4.604 variance:1.970903 ave-dev:1.0447 std-dev:1.4039
         ebizzy-je    mean: 6.736 median: 6.605 max-min:1.376 variance:0.188500 ave-dev:0.2973 std-dev:0.4342
         ebizzy-none  mean: 6.687 median: 6.590 max-min:1.230 variance:0.141300 ave-dev:0.2131 std-dev:0.3759
         ebizzy-slim  mean: 6.475 median: 6.477 max-min:0.016 variance:0.000029 ave-dev:0.0045 std-dev:0.0054

100 ms   ebizzy-libc  mean: 8.280 median: 7.159 max-min:4.373 variance:3.131708 ave-dev:1.4444 std-dev:1.7697
         ebizzy-mim   mean:12.671 median:12.351 max-min:4.791 variance:1.926828 ave-dev:0.9310 std-dev:1.3881
         ebizzy-tc    mean: 9.723 median: 9.418 max-min:1.390 variance:0.304272 ave-dev:0.4718 std-dev:0.5516
         ebizzy-je    mean: 6.699 median: 6.575 max-min:0.918 variance:0.127245 ave-dev:0.2682 std-dev:0.3567
         ebizzy-none  mean: 6.684 median: 6.447 max-min:1.469 variance:0.233320 ave-dev:0.3385 std-dev:0.4830
         ebizzy-slim  mean: 6.471 median: 6.469 max-min:0.025 variance:0.000056 ave-dev:0.0054 std-dev:0.0075

250 ms   ebizzy-libc  mean: 9.983 median:10.596 max-min:2.654 variance:1.706292 ave-dev:1.2202 std-dev:1.3063
         ebizzy-mim   mean:12.988 median:12.529 max-min:3.330 variance:1.359300 ave-dev:0.9947 std-dev:1.1659
         ebizzy-tc    mean:11.275 median:11.440 max-min:3.924 variance:1.339614 ave-dev:0.8996 std-dev:1.1574
         ebizzy-je    mean: 7.102 median: 6.628 max-min:1.450 variance:0.445353 ave-dev:0.6189 std-dev:0.6673
         ebizzy-none  mean: 7.053 median: 6.844 max-min:1.314 variance:0.332450 ave-dev:0.5016 std-dev:0.5766
         ebizzy-slim  mean: 6.475 median: 6.473 max-min:0.025 variance:0.000051 ave-dev:0.0048 std-dev:0.0072

500 ms   ebizzy-libc  mean: 9.414 median: 8.815 max-min:4.416 variance:2.684432 ave-dev:1.4016 std-dev:1.6384
         ebizzy-mim   mean:12.475 median:12.529 max-min:2.520 variance:0.703143 ave-dev:0.6428 std-dev:0.8385
         ebizzy-tc    mean:10.585 median: 9.949 max-min:5.596 variance:2.633603 ave-dev:1.0804 std-dev:1.6228
         ebizzy-je    mean: 7.470 median: 7.849 max-min:1.387 variance:0.305503 ave-dev:0.4664 std-dev:0.5527
         ebizzy-none  mean: 6.731 median: 6.596 max-min:1.505 variance:0.239643 ave-dev:0.3498 std-dev:0.4895
         ebizzy-slim  mean: 6.472 median: 6.471 max-min:0.015 variance:0.000025 ave-dev:0.0040 std-dev:0.0050

1 sec    ebizzy-libc  mean: 9.084 median: 8.832 max-min:4.619 variance:3.279747 ave-dev:1.4354 std-dev:1.8110
         ebizzy-mim   mean:12.532 median:12.411 max-min:2.589 variance:0.521370 ave-dev:0.4850 std-dev:0.7221
         ebizzy-tc    mean: 9.970 median: 9.861 max-min:1.832 variance:0.398112 ave-dev:0.4714 std-dev:0.6310
         ebizzy-je    mean: 7.079 median: 6.939 max-min:1.428 variance:0.374597 ave-dev:0.5295 std-dev:0.6120
         ebizzy-none  mean: 6.778 median: 6.532 max-min:1.463 variance:0.251937 ave-dev:0.3964 std-dev:0.5019
         ebizzy-slim  mean: 6.474 median: 6.474 max-min:0.020 variance:0.000029 ave-dev:0.0038 std-dev:0.0054

2 sec    ebizzy-libc  mean: 8.762 median: 8.736 max-min:4.314 variance:2.423943 ave-dev:1.0740 std-dev:1.5569
         ebizzy-mim   mean:12.061 median:12.218 max-min:1.388 variance:0.200091 ave-dev:0.3789 std-dev:0.4473
         ebizzy-tc    mean:10.340 median:10.135 max-min:3.438 variance:1.220981 ave-dev:0.8588 std-dev:1.1050
         ebizzy-je    mean: 6.740 median: 6.532 max-min:1.368 variance:0.213691 ave-dev:0.3342 std-dev:0.4623
         ebizzy-none  mean: 6.630 median: 6.522 max-min:1.474 variance:0.206617 ave-dev:0.2571 std-dev:0.4546
         ebizzy-slim  mean: 6.473 median: 6.474 max-min:0.012 variance:0.000012 ave-dev:0.0027 std-dev:0.0034
```

Above, a delay of zero has worked best for almost all allocators so it does not seem conclusive, if it's even pertinent: all computers run many programs concurrently (the OS alone runs dozens of usermode and kernel tasks) so an erratic background CPU and RAM latencies is inevitable [20] (“System Monitor” CPU Core activity on a supposedly *“idle”* Linux Desktop PC):

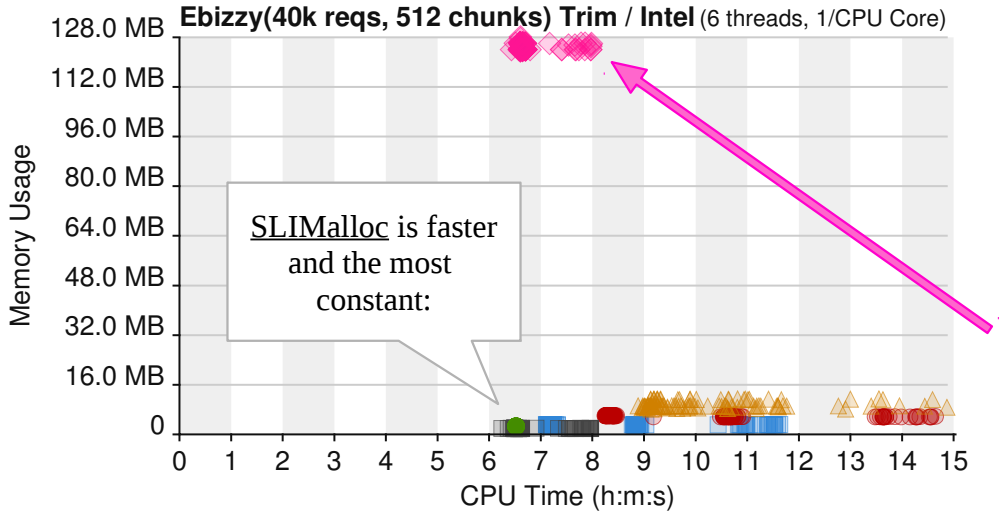


Constant allocators that much better tolerate previous and concurrent workloads have certainly more value than the less resilient kind!

That's what the ***variance*** highlights – in SLIMalloc's case, by several orders of magnitude.

100 test-runs of all 6 allocators (without background tasks) confirm the previous results of the six above tests. Dotted charts show the circles (ending time+RSS) of the previous line/area charts:

```
250 ms   ebizzy-libc   mean: 8.629 median: 8.756 max-min:4.502 variance:2.348933 ave-dev:1.2220 std-dev:1.5326
         ebizzy-mim    mean:10.399 median:10.518 max-min:6.371 variance:4.524115 ave-dev:1.6752 std-dev:2.1270
         ebizzy-tc     mean:10.219 median: 9.683 max-min:5.955 variance:2.015242 ave-dev:1.1150 std-dev:1.4196
         ebizzy-je     mean: 6.692 median: 6.532 max-min:1.541 variance:0.171069 ave-dev:0.2919 std-dev:0.4136
         ebizzy-none   mean: 6.765 median: 6.520 max-min:1.717 variance:0.283991 ave-dev:0.4383 std-dev:0.5329
         ebizzy-slim   mean: 6.467 median: 6.467 max-min:0.019 variance:0.000018 ave-dev:0.0034 std-dev:0.0042
```



Charts of allocator spread profiles are more intuitive than the statistics.

JEmalloc trims ~half to ***save time*** (increase speed) and to ***reduce its variability***.

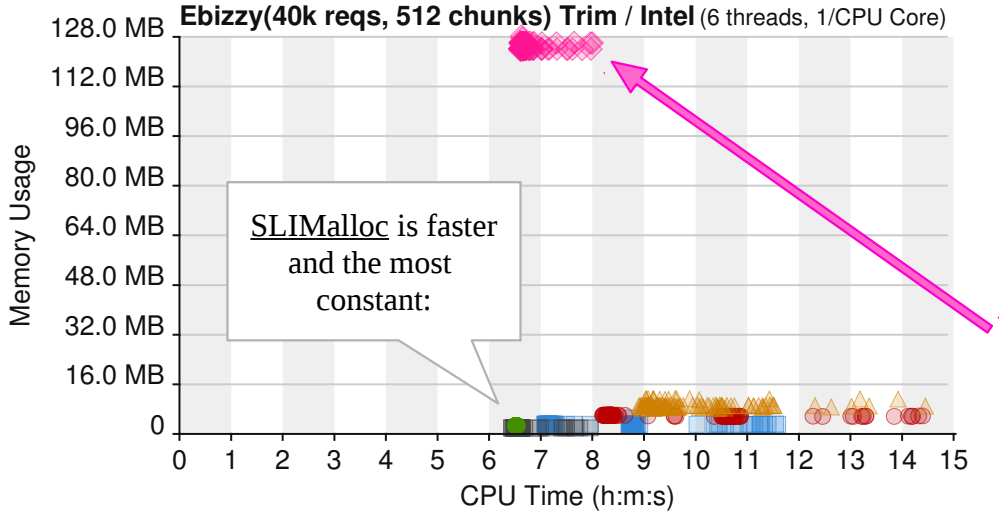
That's a 1 hour 25 minutes test.

OK. But, was this group of ***600 consecutive tests lucky for some*** and ***unlucky for others***?

Let's check this with three times more tests done later (at different daytimes and without reboot). They are sorted by growing SLIMalloc variance to help distinguish any test incoherences.

The results are so consistent that we did not have to adjust the comments' arrows and text:

```
250 ms   ebizzy-libc   mean: 8.332 median: 8.661 max-min:4.498 variance:1.898960 ave-dev:1.1390 std-dev:1.3780
         ebizzy-mim    mean: 9.845 median: 9.555 max-min:6.166 variance:3.239778 ave-dev:1.5134 std-dev:1.7999
         ebizzy-tc     mean:10.008 median: 9.557 max-min:5.525 variance:1.382394 ave-dev:0.9111 std-dev:1.1758
         ebizzy-je     mean: 6.675 median: 6.546 max-min:1.393 variance:0.103357 ave-dev:0.2123 std-dev:0.3215
         ebizzy-none   mean: 6.586 median: 6.489 max-min:1.512 variance:0.093341 ave-dev:0.1775 std-dev:0.3055
         ebizzy-slim   mean: 6.469 median: 6.468 max-min:0.034 variance:0.000034 ave-dev:0.0041 std-dev:0.0059
```



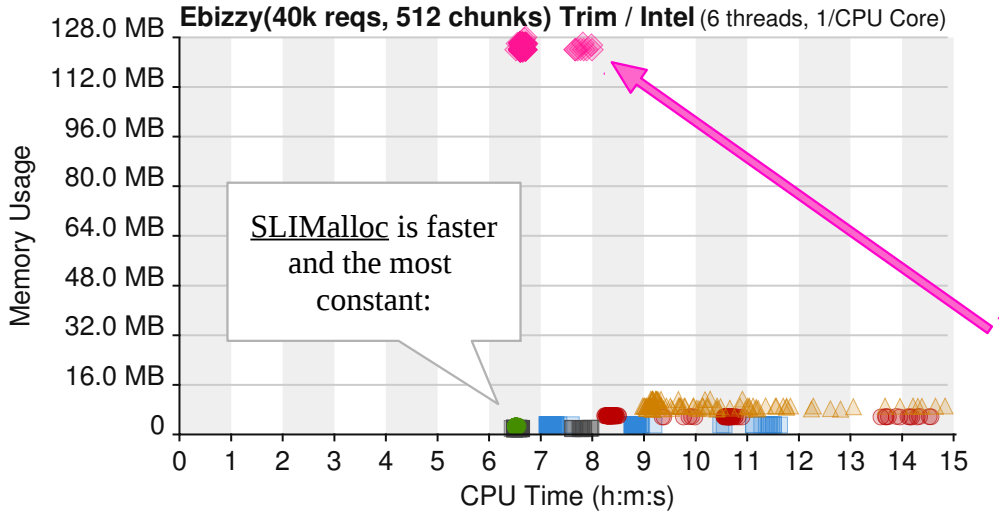
Charts of allocator spread profiles are more intuitive than the statistics.

JEmalloc trims ~half to ***save time*** (increase speed) and to ***reduce its variability***.

That's a 1 hour 23 minutes test.

Things change – but allocator behavior is ***familiar and recognizable*** (time, ram, distribution).

```
250 ms   ebizzy-libc   mean: 8.216 median: 7.294 max-min:4.488 variance:1.792758 ave-dev:1.1142 std-dev:1.3389
         ebizzy-mim    mean: 9.800 median: 8.443 max-min:6.290 variance:3.653523 ave-dev:1.5575 std-dev:1.9114
         ebizzy-tc     mean:10.196 median: 9.601 max-min:5.829 variance:2.224022 ave-dev:1.1615 std-dev:1.4913
         ebizzy-je     mean: 6.623 median: 6.526 max-min:1.464 variance:0.117059 ave-dev:0.1925 std-dev:0.3421
         ebizzy-none   mean: 6.635 median: 6.517 max-min:1.519 variance:0.170202 ave-dev:0.2636 std-dev:0.4126
         ebizzy-slim   mean: 6.466 median: 6.465 max-min:0.055 variance:0.000051 ave-dev:0.0045 std-dev:0.0072
```



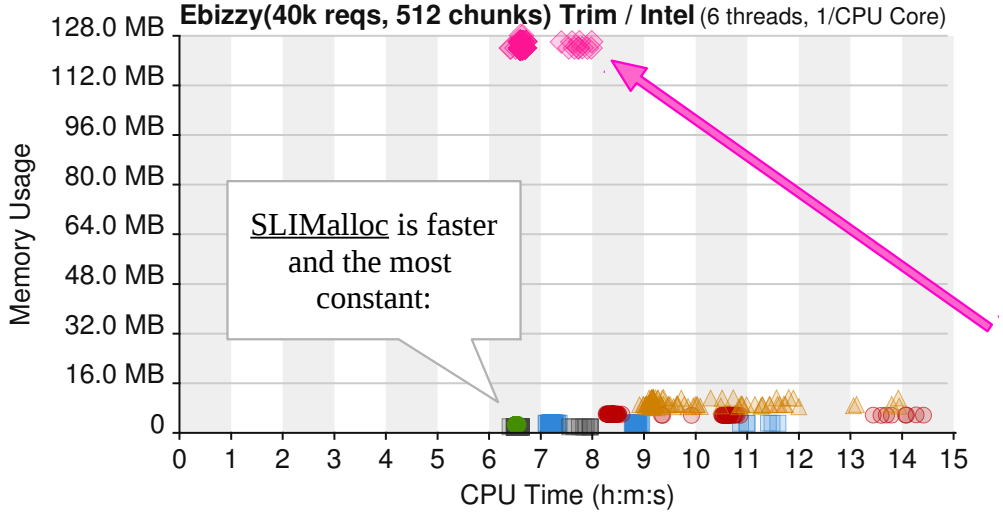
Charts of allocator spread profiles are more intuitive than the statistics.

JEmalloc trims ~half to ***save time*** (increase speed) and to ***reduce its variability***.

That's a 1 hour 23 minutes test.

These behavior ***variations merely confirm*** each allocator's "personality".

```
250 ms   ebizzy-libc   mean: 7.995 median: 7.207 max-min:4.466 variance:1.247309 ave-dev:0.9595 std-dev:1.1168
         ebizzy-mim    mean: 9.495 median: 8.389 max-min:6.118 variance:2.688220 ave-dev:1.3623 std-dev:1.6396
         ebizzy-tc     mean: 9.876 median: 9.201 max-min:5.005 variance:1.466731 ave-dev:0.9565 std-dev:1.2111
         ebizzy-je     mean: 6.624 median: 6.490 max-min:1.573 variance:0.127011 ave-dev:0.2172 std-dev:0.3564
         ebizzy-none   mean: 6.577 median: 6.445 max-min:1.552 variance:0.126568 ave-dev:0.1891 std-dev:0.3558
         ebizzy-slim   mean: 6.468 median: 6.467 max-min:0.048 variance:0.000062 ave-dev:0.0047 std-dev:0.0079
```



Charts of allocator spread profiles are more intuitive than the statistics.

JEmalloc trims ~half to ***save time*** (increase speed) and to ***reduce its variability***.

That's a 1 hour 22 minutes test.

Here we have compared the ***statistics of data acquired during 2,760 program executions lasting 6 hours and 36 minutes*** – not just ***a bunch of test runs***. That's a world of difference.

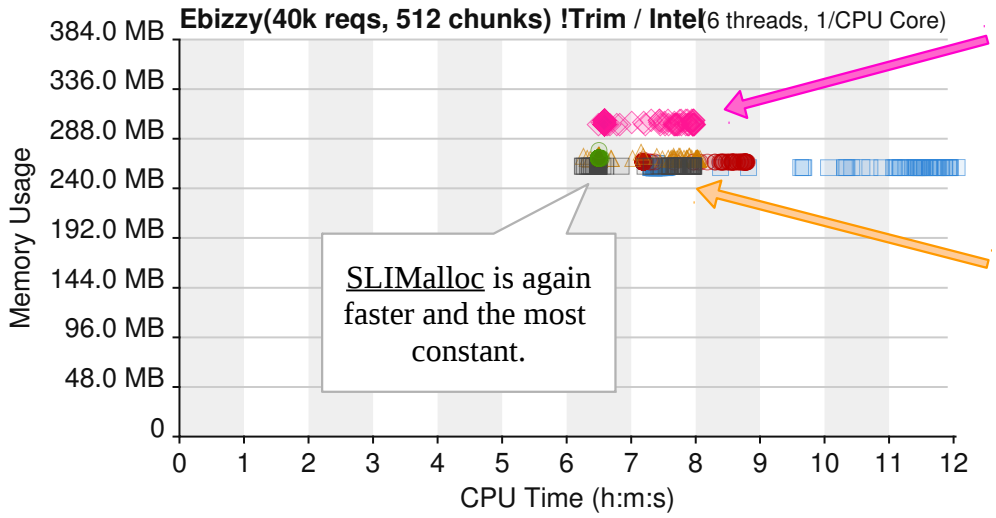
We see ***subtle and large variations***, as well as per-allocator ***recognizable behavior patterns and performance*** from which we can learn to understand what is going on – to make progress.

Reality is the only criteria now ***we have written a tool to acquire and visualize a lot of data***.

We have demonstrated that SLIMalloc does efficient and fast trimming – a feature that TCmalloc authors seriously call *"a bad idea"*. Google could safely add that TCmalloc (like most) is faster ***without trimming*** – and that *then* our tests' outcomes would be quite different.

There's only one way to know (now we show the peak RSS instead of the ending RSS value):

```
250 ms   ebizzynt-libc   mean: 9.089 median: 7.469 max-min:4.728 variance:3.760846 ave-dev:1.8632 std-dev:1.9393
         ebizzynt-mim    mean: 7.800 median: 7.312 max-min:1.608 variance:0.484058 ave-dev:0.6717 std-dev:0.6957
         ebizzynt-tc     mean: 6.995 median: 6.459 max-min:1.798 variance:0.444855 ave-dev:0.6330 std-dev:0.6670
         ebizzynt-je     mean: 7.005 median: 6.518 max-min:1.477 variance:0.346944 ave-dev:0.5614 std-dev:0.5890
         ebizzynt-none   mean: 6.887 median: 6.444 max-min:1.721 variance:0.394055 ave-dev:0.5861 std-dev:0.6277
         ebizzynt-slim   mean: 6.452 median: 6.452 max-min:0.048 variance:0.000033 ave-dev:0.0030 std-dev:0.0057
```



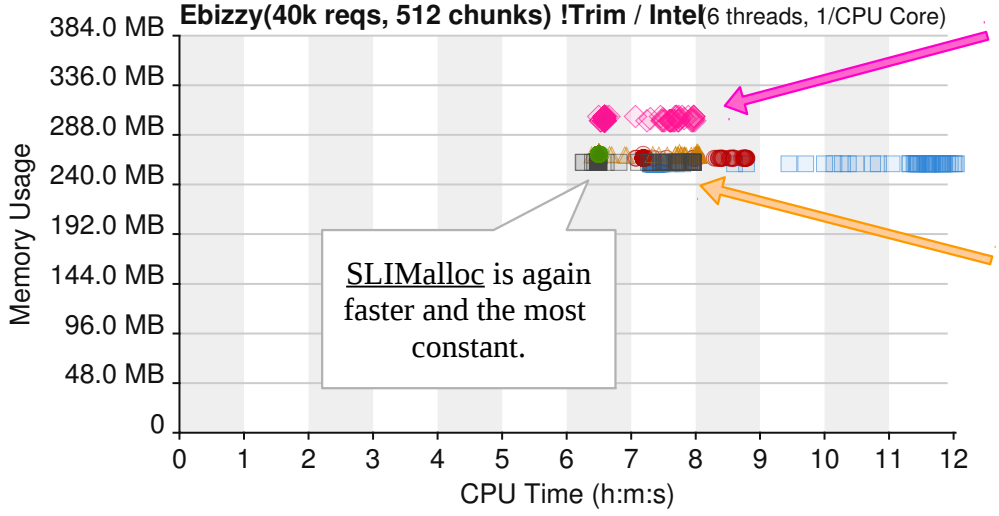
JEmalloc in a no-trimming test ***is still as variable*** (still trimming).

MIMalloc and TCmalloc (even more) are now faster and ***much less variable***.

That's a 1 hour 17 minutes test.

These consecutive 100 test-runs (of all 6 allocators) confirm the above 100 test-runs results:

```
250 ms   ebizzynt-libc   mean: 9.144 median: 7.564 max-min:4.718 variance:4.000300 ave-dev:1.9259 std-dev:2.0001
         ebizzynt-mim    mean: 7.625 median: 7.139 max-min:1.724 variance:0.439390 ave-dev:0.6172 std-dev:0.6629
         ebizzynt-tc     mean: 6.958 median: 6.456 max-min:1.661 variance:0.447683 ave-dev:0.6275 std-dev:0.6691
         ebizzynt-je     mean: 6.902 median: 6.490 max-min:1.480 variance:0.304615 ave-dev:0.5172 std-dev:0.5519
         ebizzynt-none   mean: 6.882 median: 6.444 max-min:1.700 variance:0.372877 ave-dev:0.5663 std-dev:0.6106
         ebizzynt-slim   mean: 6.451 median: 6.451 max-min:0.041 variance:0.000021 ave-dev:0.0027 std-dev:0.0045
```



JEmalloc in a no-trimming test ***is still as variable*** (still trimming).

MIMalloc and TCmalloc (even more) are now faster and ***much less variable***.

That's a 1 hour 16 minutes test.

Trimming was exacerbating the allocators' defects that remain very visible without trimming.

Conclusion. Many test runs (the good, the bad, the ugly) and their statistics demonstrate that:

1. The only possible way to explain that all vendors claim to have "*the fastest product*" is that most benchmarks are shamelessly biased and/or "cherry-picked" for self-promotion or product-placement (a business only because the audience can be easily abused).

2. But ***how can we prove that we don't do the same thing*** for SLIMalloc?
   By giving end-users a practical and indisputable way to check the facts.

   > *For centuries, international trading has secured transactions between total strangers by releasing a payment initially blocked in an escrow – after the received goods have been verified and match the vendor specifications accepted by the buyer.*

   As consecutive benchmarks fail to give identical results, our published "specifications" must involve ***a large number of test runs*** (absent from almost all other publications).

   The above statistics of *consecutive test runs* widely differ, yet they show unique capacity. We will show different tests (than Intel Ebizzy) to cover an even more complete reality.

   SLIMalloc licensees have to pay only if third-parties can duplicate our published results.

   Our promise is that the statistically-relevant measures we have published are trustworthy, and we are confident enough about our work to guaranty licensees that they will get what they have ordered from us (a much welcome exception in this industry).

3. We have created a program to ***run many benchmarks***, collect all the results in files, and then ***show all test execution times and statistics with per-allocator line and dotted charts*** at the steps of ***10, 50, and 100 test runs*** (each running the all the tested allocators) during hours or weeks, as required.

   We also generate each allocator's execution times' ***mean***, ***median***, ***difference range***, ***variance***, ***average and standard deviations*** to complement the dotted charts revealing if the execution-time ***variance*** of an allocator is:

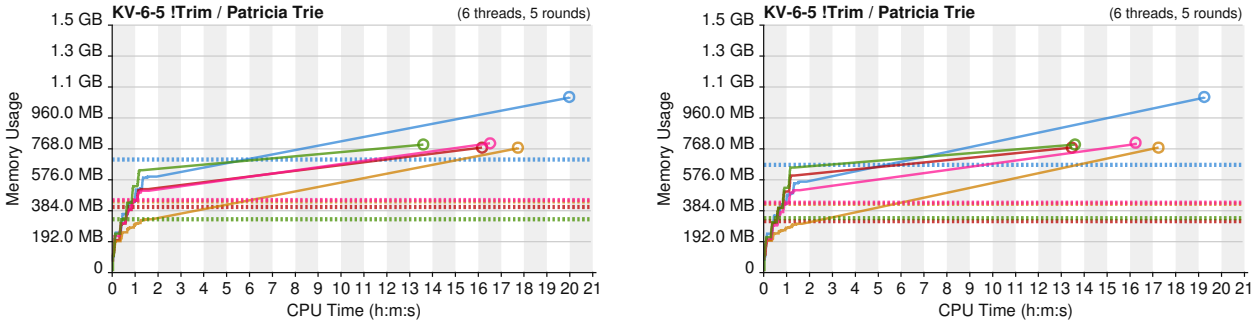
   - shared-by-all (due to other tasks running in the background). Here, all allocators show at the same time values diverging from the best (shortest execution time, lowest memory usage) and median (more common, darkest area). A larger distance from the median indicates a higher sensibility to the external perturbation (OS, background tasks, etc.),
     or,
   - per-memory-allocator (due to its design and implementation). Here, while some allocators are constant, others show a spread of bad values diverging from the best. The distance from the most constant allocator reveals the inability of other allocators to deliver top and stable performance.

The next (and final speed-related) step is about different "real-life" and "synthetic" tests.

II. SPEED / 4. More Tests

A Key-Value Store

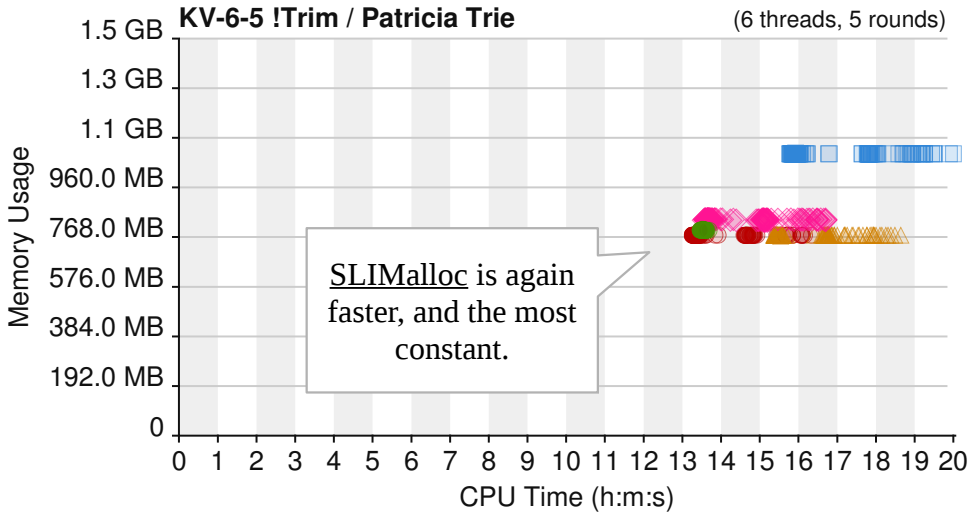
The Patricia Trie is known for its efficiency and low memory overhead as a data structure. Here, we used it with even more many random-length keys than for [1], now using the 95 MB `enwik8` archive [23] where 1,128,023 CR-terminated paragraphs are added to the KV Store, searched (top to bottom and then in reverse order), checked, and freed – 5 times, by 6 concurrent threads.



These consecutive 100 test-runs (of all allocators) now demonstrate familiar patterns but this time TCmalloc and MIMalloc have a much better variance than JEmalloc (which, for the Ebizzy benchmark, was the third best after NO-Malloc and SLIMalloc):

```
250 ms   kv5-libc    mean:17.024 median:16.129 max-min:4.224 variance:1.768371 ave-dev:1.2276 std-dev:1.3298
         kv5-mim     mean:14.043 median:13.384 max-min:3.078 variance:0.928687 ave-dev:0.8658 std-dev:0.9637
         kv5-tc      mean:16.348 median:16.567 max-min:3.289 variance:0.946467 ave-dev:0.8485 std-dev:0.9729
         kv5-je      mean:14.568 median:14.820 max-min:3.154 variance:1.104606 ave-dev:0.9359 std-dev:1.0510
         kv5-slim    mean:13.460 median:13.458 max-min:0.229 variance:0.001429 ave-dev:0.0277 std-dev:0.0378
```

NO-Malloc has been removed because it does not give much insights. And, in contrast to [1], trimming has been disabled – so the "Memory Usage" in the chart below is the ***peak RSS*** value:



TCmalloc has a good variance, but poor performance, this time.

MIMalloc, in this test, outperforms JEmalloc – both in overall speed and variance.

That's a 2 hour 40 minutes test.

To better understand how this test puts allocator under pressure, here is the SLIMalloc memory usage (complete block-size break-down) after the test has been done.

This is the “default” heap for the program (hence the GLibC TLS 288-byte allocations, and 2 large blocks for the file loaded from disk and the paragraph-pointers buffer).

This heap also acts as a “worker thread” (hence the 16-byte and 32-byte block allocations also done by all the thread heaps below):

```
--- heap[0] ---------------------------
  block-size[   used    total  resident] frag of-total (212.6 MB)
          16[      5        5    131072]  19%   0%
          24[      2        2     87381]  49%   0%
          32[3562225  3562225   3604480]   0%  41% ====================
         128[      0        1     16384]   1%   0%
         288[      4        5      7281]  19%   0%
     9024184[      0        1         0]   0%   4% ==
   100000008[      0        1         0]   0%  45% ======================
  3562236 small-block(s) in use (108.7 MB), resident-pages: 118.0 MB
```

The following are “thread” heaps (here only allocating KV structures pointing to the loaded file):

```
--- heap[1] ---------------------------
  block-size[   used    total  resident] frag of-total (108.7 MB)
          16[      5        5    131072]  19%   0%
          32[3562225  3562225   3604480]   0%  67% ==================================
  3562230 small-block(s) in use (108.7 MB), resident-pages: 112.0 MB

--- heap[2] ---------------------------
  block-size[   used    total  resident] frag of-total (108.7 MB)
          16[      5        5    131072]  19%   0%
          32[3562225  3562225   3604480]   0%  67% ==================================
  3562230 small-block(s) in use (108.7 MB), resident-pages: 112.0 MB

--- heap[3] ---------------------------
  block-size[   used    total  resident] frag of-total (108.7 MB)
          16[      5        5    131072]  19%   0%
          32[3562225  3562225   3604480]   0%  67% ==================================
  3562230 small-block(s) in use (108.7 MB), resident-pages: 112.0 MB

--- heap[4] ---------------------------
  block-size[   used    total  resident] frag of-total (108.7 MB)
          16[      5        5    131072]  19%   0%
          32[3562225  3562225   3604480]   0%  67% ==================================
  3562230 small-block(s) in use (108.7 MB), resident-pages: 112.0 MB
  RSS: 686.2 MB

--- heap[5] ---------------------------
  block-size[   used    total  resident] frag of-total (108.7 MB)
          16[      5        5    131072]  19%   0%
          32[3562225  3562225   3604480]   0%  67% ==================================
  3562230 small-block(s) in use (108.7 MB), resident-pages: 112.0 MB
```

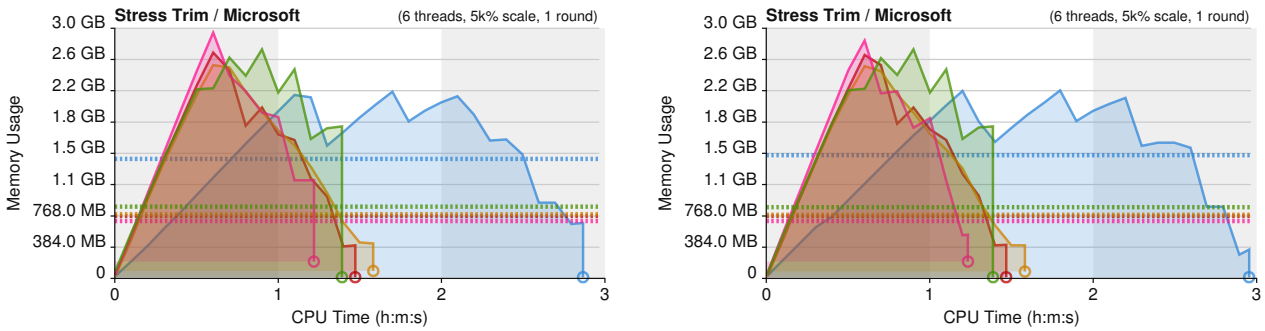
A memory allocator that *“consistently”* (here for tens of thousands of test runs over a period of time lasting more than 10 hours) shines in very different benchmarks (that work well or not so well depending on the allocator's architecture) has certainly more value than those who can only claim to occasionally perform with carefully-chosen tests and cherry-picked test results.

This is even more true as SLIMalloc delivers never-seen-before yet very desirable features, including today's highest higher standards of security – usable in production.

The Microsoft Stress Test

This artificial *"stress test"* from Microsoft Research validates memory allocators with aggressive small/large reallocations. We used it as a benchmark in our first paper [1] and then noted that most allocators had problems with it (slow execution times and crashes).

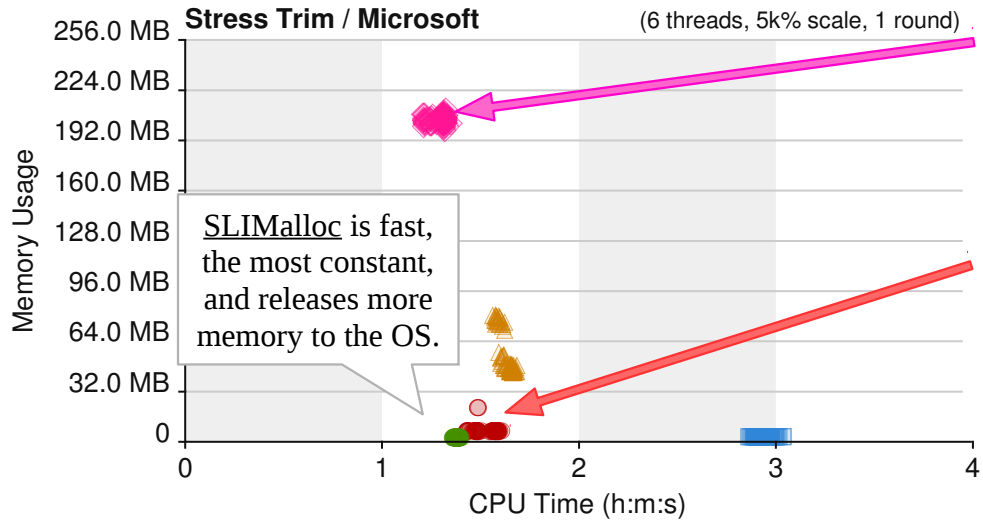
Microsoft, Google and Facebook have improved in this test. SLIMalloc was a bit faster and using less memory [1] than today, due to its new features. Despite this "*considerable performance overhead*", SLIMalloc is again fast and by far the most constant:



These consecutive 100 test-runs (of all allocators) show again that SLIMalloc is much more constant than all others (its variance is orders of magnitude lower):

```
250 ms   str-libc   mean:2.936  median:2.931  max-min:0.178  variance:0.001291  ave-dev:0.0286  std-dev:0.0359
         str-mim    mean:1.530  median:1.564  max-min:0.177  variance:0.002969  ave-dev:0.0509  std-dev:0.0545
         str-tc     mean:1.632  median:1.648  max-min:0.124  variance:0.001046  ave-dev:0.0286  std-dev:0.0323
         str-je     mean:1.273  median:1.291  max-min:0.141  variance:0.001477  ave-dev:0.0316  std-dev:0.0384
         str-slim   mean:1.375  median:1.373  max-min:0.048  variance:0.000089  ave-dev:0.0077  std-dev:0.0094
```

Again, JEmalloc releases far less memory to the OS (than all others) to gain more speed:



JEmalloc favors speed (again) at the expense of not releasing memory to the OS.

MIMalloc now passes its own test much better than in 2020.

That's a 22 minutes and 15 seconds test.

```
GLibC ..................     2.3 MB ending RSS (the second best after SLIMalloc)
MIMalloc ...............     5.9 MB ending RSS (with 3 "accidents" at 20.8 MB)
TCmalloc ...............    78.5 MB ending RSS (with a wide range of results)
JEmalloc ...............   206.5 MB ending RSS (with the usual least effort)
SLIMalloc ..............     1.9 MB ending RSS (the best and most constant)
```

II. SPEED / 5. Conclusion

Performance ***variations*** are due to the allocator (and obviously not to the alleged *"task scheduler's randomness"* or to *"background tasks"* as they do not affect SLIMalloc as much as others). These variations reveal that ***the bottleneck is the memory allocator*** rather than the OS kernel.

Our test machine has been upgraded as compared to [1]:

HW: 6-Core MacPro (Intel Xeon CPU W3680 @ 3.33GHz), 48 GB RAM DDR3 1333 MHz
OS: Ubuntu 14.04.2 LTS, GLIBC v2.19 (v2.26-2.32 builds fail:"*too old: GNU ld*")

Memory allocators were downloaded on October 12, 2022 from their public repository (except SLIMalloc and GLibC) and linked statically to the executable (except for GLibC and NO-Malloc: statically linking the whole LibC would have artificially reduced their RSS – a measure that stays relevant only if we compare allocators using the same application).

What we have done in this technical report is just measuring... conflicts of interests.

"Cherry-picking" is the technique of selecting data matching the thesis you want to demonstrate, and ignoring all the rest (sadly, a widely used practice nowadays).

> A variation (heavily used in public contracts) imposes complex norms, certifications, and capital thresholds only available to international publicly-traded companies – whether the promoted vendor's product or service has merits or not (it often is not even delivered).
>
> On the top of wasting resources that would be game-changers elsewhere these entrenched insidious practices are endangering the very basis of all human activities:
>
> - Critical Infrastructure (energy grid, drinkable water, telecoms, transportation),
> - Automotive (remotely spied, hijacked and stolen cars, charging stations, etc.),
> - MedTech (health sensors, insulin pumps, pace-makers, medical appliances),
> - FinTech (payment/trading/exchange/compensation devices and platforms),
> - Defense (all connected equipment is vulnerable, can be used against owners).

Since SLIMalloc, in only 1.5 years, has redefined – *at the benefit of all end-users* – the standards of performance and security, acquiring a license is certainly the quickest, easiest and less hazardous way to be *faster, safer, and more capable*.

This is true for national and local governments, for large and small companies, and for individuals.

References

[1] *"SLIMalloc: a Safer, Faster, and more Capable Heap Allocator"* (June 2020)
http://twd.ag/archives/twd_slim_1-pager.pdf http://twd.ag/archives/slim.pdf

[2] Ken Thompson, *"Reflections on Trusting Trust"*,
Communications of the ACM, volume 27, number 8, pages 761-763 (August 1984)
https://cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf

[3] The Hacker News, *"Last Years Open Source - Tomorrow's Vulnerabilities"* (2022)
https://thehackernews.com/2022/11/last-years-open-source-tomorrows.html

[4] Mitre, *"2022 CWE Top 25 Most Dangerous Software Weaknesses"* (2022)
https://cwe.mitre.org/data/definitions/1387.html

[5] Statista, *"Size of Cyber Security Market Worldwide from 2019 to 2030"* (2022)
https://www.statista.com/statistics/1256346/worldwide-cyber-security-market-revenues/

[6] 20th National Information Systems Security Conference, PMO RCAS, Boeing IS,
"Software Encryption in the DoD" (1997)
https://csrc.nist.gov/csrc/media/publications/conference-paper/1997/10/10/proceedings-of-the-20th-nissc-1997/documents/543.pdf

[7] The Economist, Special report | Vulnerabilities,
"Zero-day game - Wielding a controversial cyber-weapon" (July 10, 2014)
"*How do you protect what you want to exploit?" asks Scott Charney, an executive at Microsoft. He highlights a dilemma. Intelligence agencies look for programming mistakes in software so they can use them to spy on terrorists and other targets. But if they leave open these security holes, known in tech jargon as 'vulnerabilities', they run the risk that hostile hackers will also find and exploit them.*"
https://www.economist.com/special-report/2014/07/10/zero-day-game

[8] Reuters, *"A scramble at Cisco exposes uncomfortable truths about U.S. cyber defense"*
(March 29, 2017)
"*Across the federal government, about 90 percent of all spending on cyber programs is dedicated to offensive efforts, including penetrating the computer systems of adversaries, listening to communications and developing the means to disable or degrade infrastructure.*"
https://www.reuters.com/article/us-usa-cyber-defense-idUSKBN17013U

[9] The Washington Post,
"Obama's secret struggle to punish Russia for Putin's election assault" (June 23, 2017)
"Obama also approved a previously undisclosed covert measure that authorized planting cyberweapons in Russia's infrastructure, the digital equivalent of bombs that could be detonated if the United States found itself in an escalating exchange with Moscow."
https://www.washingtonpost.com/graphics/2017/world/national-security/obama-putin-election-hacking/?hpid=hp_hp-banner-high_russiaobama-banner-7a:homepage/story&utm_term=.17b842c4a03c

[10] Wikipedia, *"Linus' Law"* (2002)
https://en.wikipedia.org/wiki/Linus's_law

[11] ZDNet, *"Microsoft: 70 percent of all security bugs are memory safety issues"* (2019)
https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

[12] Wired, *"Apple's T2 Security Chip Has an Unfixable Flaw"* (2020)

https://www.wired.com/story/apple-t2-chip-unfixable-flaw-jailbreak-mac/

[13] NIST, *"e-Handbook of Statistical Methods – Scale, Variability, or Spread"* (2022)
https://www.itl.nist.gov/div898/handbook/eda/section3/eda356.htm

[14] Wikipedia, *"Security Theater"* (2005-2022)
"the practice of taking security measures that are considered to provide the feeling of improved security while doing little or nothing to achieve it."
https://en.wikipedia.org/wiki/Security_theater

[15] TU Dresden, University of Edinburgh, University of Neuchâtel, "*Intel MPX Explained*" (2017) https://arxiv.org/abs/1702.00719

Wikipedia, *"Intel MPX"*
https://en.wikipedia.org/wiki/Intel_MPX

[16] European Union, *"Sustainable development in the European Union, Monitoring report on progress towards the SDGs in an EU context"* (2020 edition)
ISBN 978-92-76-17443-1 doi:10.2785/555257

[17] The Hacker News, "*Google patches Chrome security issues (most are memory errors), the fourth actively exploited type-confusion flaw that Google has addressed since the start of the year. It's also the ninth zero-day flaw in Chrome attackers have exploited in the wild in 2022. These vulnerabilities could be weaponized by threat actors to perform out-of-bounds memory access, or lead to a crash and arbitrary code execution exploited to pass arbitrary code and gain control over a victim's system.*" (2022)
CVE-2022-0609 *- Use-after-free in Animation*
CVE-2022-1096 *- Type confusion in V8*
CVE-2022-1364 *- Type confusion in V8*
CVE-2022-2294 *- Heap buffer overflow in WebRTC*
CVE-2022-3723 *- Type confusion in V8*
CVE-2022-4135 *- Heap buffer overflow in GPU"*
https://thehackernews.com/2022/12/google-rolls-out-new-chrome-browser.html

[18] The Economist, "*The Internet of things (to be hacked)"* (Jul 12, 2014)
"To avoid lurid headlines about car crashing, insulin overdoses and houses burning, tech firms will surely have to embrace higher standards."
http://www.economist.com/node/21606829/print

The Economist, "*Market failures - Not my problem*" (July 12, 2014)
"*There's a market failure in cyber-security, made worse by the trouble firms have in getting reliable information about the threats they face.*"
http://www.economist.com/node/21606422/print

[19] Wikipedia, "*Thrashing (computer science)*"
https://en.wikipedia.org/wiki/Thrashing_(computer_science)

[20] Netflix / Brendan Gregg, "*p1bench - perturbation benchmark*" (2018)
https://github.com/brendangregg/p1bench

[21] ZeroHedge, "*Our Parasitic Generation*" (Dec 12, 2022)
"*What happens when a later parasitic generation mocks but still consumes what it inherits but cannot create. [...] All seem testaments to our incompetence, arrogance, and ignorance. […] Our great universities, once the most esteemed in the world, grow burdened with commissars, their faculties are weaponized, and their students have never been more confident in their abilities, and with so little reason for that confidence.*"
https://www.zerohedge.com/political/victor-davis-hanson-our-parasitic-generation

[22] The Hacker News, "*Researchers Demonstrate How EDR (Endpoint Detection and Response) and Antivirus Can Be Weaponized Against Users*" (Dec 12, 2022)
"Microsoft Defender and Defender for Endpoint, Trend Micro Apex One, Avast and AVG Antivirus"
https://thehackernews.com/2022/12/researchers-demonstrate-how-edr-and.html

[23] Matt Mahoney, "*Large Text Compression Benchmark, About the Test Data*" (2022)
http://www.mattmahoney.net/dc/textdata.html

[24] Multinationals Observatory, "*GAFAM Nation. The web of influence of the web giants in France*" (Dec 13, 2022)
"*Rapidly increasing lobbying expenses, poaching of high ranking civil servants, contacts at the Elysée Palace, financial partnerships with media, think tanks and research institutions... Diving into the formidable machinery of lobbying and influence deployed in France by the web giants Google, Amazon, Facebook, Apple and Microsoft.*"
https://multinationales.org/fr/enquetes/gafam-nation/

[25] The Economist, "*Robber barons and silicon sultans*" (Jan 3rd, 2015)
https://www.economist.com/briefing/2015/01/03/robber-barons-and-silicon-sultans

The Economist, "*A world of robber barons*" (Feb 22, 2014)
https://www.economist.com/sites/default/files/20140222_companies.pdf

[26] Le Figaro, "*2022, the highest tax year in thirty years!*"
https://www.lefigaro.fr/vox/economie/agnes-verdier-molinie-2022-l-annee-record-des-prelevements-obligatoires-depuis-trente-ans-20221017
https://www.ifrap.org/budget-et-fiscalite/2022-lannee-record-des-prelevements-obligatoires-depuis-trente-ans

République Française, "*Rapport économique et financier*" (Oct 3, 2022) page 87
https://www.tresor.economie.gouv.fr/Articles/5b9dd056-db45-43ca-973d-2320c5157d59/files/7d5a8ecc-fc65-4a02-960d-000aa71a191b

[27] Michael Hudson, "*Killing the Host, How Financial Parasites and Debt Bondage Destroy the Global Economy*" (2015)
https://medium.com/the-capital/financialization-of-the-economy-a-parasite-economy-500b90947209

[28] U.S. National Security Agency, "*NSA Releases Guidance on How to Protect Against Software Memory Safety Issues*" (Nov 10, 2022)
https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/

[29] Mark Nottingham, "*Internet protocols are changing*" (Dec 12, 2017)
https://blog.apnic.net/author/mark-nottingham/
TWD's analysis, in 5 points:
1. The HTTP/1.1 Web protocol allowed people to filter which servers users want to be connected to: network administrators used DNS white-lists for friends and blacklists for foes. Each server required at least one dedicated TCP connection (easily identifiable and blocked).

2. Google has written HTTP/2 with DoH (DNS over HTTPS) to completely bypass DNS queries so that (1) users can no-longer know who they are connected to and (2) administrators can no-longer filter infiltration or exfiltration done via HTTPS (the Web). And with HTTP/2 multiplexing, one single HTTP/2 connection can involve thousands of hidden servers and cloud services that end-users have no way to merely list and even less selectively allow or block.

3. People can check who they are connected to and selectively allow the hosts they trust by using the following Web browser add-on: https://github.com/gorhill/uMatrix

4. All Web browsers are crippled by vulnerabilities and even those claiming to respect (or defend!) privacy secretly maintain encrypted connections to *constantly* spy end-users: https://www.researchgate.net/publication/349979628_Web_Browser_Privacy_What_Do_Browsers_Say_When_They_Phone_Home

5. Web browsers – like Certificate Authorities, Operating Systems, and thousands of private companies and government agencies – can, at any time, enforce their own *"root certificates"* (a complete bypass of all the security supposedly provided by SSL/TLS certificates) and their own certificate verification procedure. They can (and do) play games and are routinely compromised by third-parties (ie: the year 1999 "*Flame malware*").

[30] aBetterInternet.org NGO funded by Google, Amazon, Facebook, Microsoft, Cisco, Mozilla, IBM, EFF, OVH and many others: https://www.abetterinternet.org/sponsors/

"*Using C/C++ is bad for society, bad for your reputation, and bad for your customers*."
[...]
"*A recent study found that 60-70% of vulnerabilities in iOS and macOS are memory safety vulnerabilities. Microsoft estimates that 70% of all vulnerabilities in their products over the last decade have been memory safety issues. Google estimated that 90% of Android vulnerabilities are memory safety issues. An analysis of 0-days that were discovered being exploited in the wild found that more than 80% of the exploited vulnerabilities were memory safety issues.*"
https://www.memorysafety.org/docs/memory-safety/

"*Our first goal is to move the Internet's security-sensitive software infrastructure to memory safe code. Many of the most critical software vulnerabilities are memory safety issues in C and C++ code.* [...] *Using memory safe languages eliminates the entire class of issues.* [...]
Our second goal is to change the way people think about memory safety. Today it's considered perfectly normal and acceptable to deploy software written in languages that aren't memory safe, like C and C++, on a network edge, despite the overwhelming evidence for how dangerous this is. Our hope is that we can get people to fully recognize the risk and view memory safety as a requirement for software in security-sensitive roles.
[...]
We believe we have a strong competency in identifying work that is both high impact and efficiently achievable. Our aim is for funding entrusted to us to provide a strong return on investment in terms of making the Internet's software infrastructure safer for everyone."
https://www.memorysafety.org/about/

[31] StackOverflow, "*Why is C so fast, and why aren't other languages as fast or faster*"
https://stackoverflow.com/questions/418914/why-is-c-so-fast-and-why-arent-other-languages-as-fast-or-faster

[32] Microsoft Network, "*Windows Update hijacked to infect PCs with malware*"

(Jan 28, 2022)
https://www.msn.com/en-us/news/technology/windows-update-hijacked-to-infect-pcs-with-malware/ar-AATfany

Windows Report, "*Windows Updates are used to spread malware by Lazarus hackers*"
https://windowsreport.com/windows-update-lazarus-malware/

BC, "*Windows Update can be abused to execute malicious programs*" (Oct 12, 2020)
https://www.bleepingcomputer.com/news/security/windows-update-can-be-abused-to-execute-malicious-programs/

ITIGIC, "*Hackers Use Windows Update to Sneak Malware onto Users*"
https://itigic.com/hackers-use-windows-update-to-sneak-malware-onto-users/

CNET, "*Flame virus can hijack PCs by spoofing Windows Update*" (June 5, 2012)
https://www.cnet.com/tech/tech-industry/flame-virus-can-hijack-pcs-by-spoofing-windows-update/

TechSpot, "*Flame malware subverts Windows Updates, infects networked PCs*"
https://www.techspot.com/news/48886-flame-malware-subverts-windows-updates-infects-networked-pcs.html

Ars Technica, "*Flame malware hijacks Windows Update to spread from PC to PC*"
https://arstechnica.com/information-technology/2012/06/flame-malware-hijacks-windows-update-to-propogate/

[33] Thomas Roulet / University of Cambridge, "*Sins for Some, Virtues for Others*" (Oct. 2018)
"*Our findings show that the more banks are disapproved for their wrongdoings, the more likely they are to be selected to join a syndicate. This study suggests that the coverage of misconduct can actually act as a positive signal providing banks with incentives to engage in what is broadly perceived as professional misconduct.*"
https://www.researchgate.net/publication/328615870_Sins_for_some_virtues_for_others_Media_coverage_of_investment_banks%27_misconduct_and_adherence_to_professional_norms_during_the_financial_crisis
https://journals.sagepub.com/doi/epub/10.1177/0018726718799404

[34] "*Exploiting Memory Corruption Vulnerabilities in the Java Runtime*" (2011)
Joshua J. Drake, Black Hat Abu Dhabi

Mitre, "*Search CVE (Common Vulnerabilities and Exposures) List*" (January 5, 2023)
https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=.net .NET (2002) **1995 CVE Records**
https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=java Java (1995) **2538 CVE Records**
https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=glibc GLibC (1987) **174 CVE Records**

[35] GNU, "*Object Size Checking Built-in Functions*"
"*GCC implements a limited buffer overflow protection mechanism that can prevent some buffer overflow attacks by determining the sizes of objects into which data is about to be written and preventing the writes when the size isn't sufficient.*"
https://gcc.gnu.org/onlinedocs/gcc/Object-Size-Checking.html

[36] Wikipedia, "*Peter Principle*" (1969)
"In a hierarchy, every employee tends to rise to his level of incompetence."
https://en.wikipedia.org/wiki/Peter_principle