Hardware Memory Tagging

to make C/C++ memory safe(r)

iSecCon'18

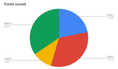
Kostya Serebryany kcc@google.com

December 2018

Agenda

- Memory Safety: Industry-Wide Crisis
  - ASAN
- Memory Tagging vs Memory Safety
  - **ARM v8.5 MTE**
  - SPARC ADI
  - LLVM HWASAN
- Call to Action

C & C++ memory safety is a mess

- Use-after-free / buffer-overflow / uninitialized memory
- > 50% of High/Critical security bugs in Chrome & Android
  - Lots of scary  to follow
- Not only security vulnerabilities
  - crashes, data corruption, developer productivity
- AddressSanitizer (ASAN) is not enough
  - Hard to use in production
  - Not a security mitigation

Dynamic Tools @ Google (who I am)

- 2008-2011: deploying Valgrind/Memcheck at Google
- 2011: implemented AddressSanitizer (ASAN)
  - Deployed for Google server-side (250 MLOC C++), Chrome, Android, ...
  - Available on Linux, Windows, OSX, *BSD, x86, ARM, MIPS, SPARC, Power, …
  - Also used by Apple, Facebook, Samsung, Sony, Mozilla, Oracle, …
- 2013: implemented Linux Kernel AddressSanitizer (KASAN)
- Responsible for finding ~ 50K memory safety bugs since 2008
- Advocating for a hardware implementation since 2012
- 2018: Published a study of Memory Tagging, implemented a SW prototype

- GWP-ASAN, Control Flow Integrity, fuzzing tools & services, data race detection, ...

C and C++ memory (un)safety 101

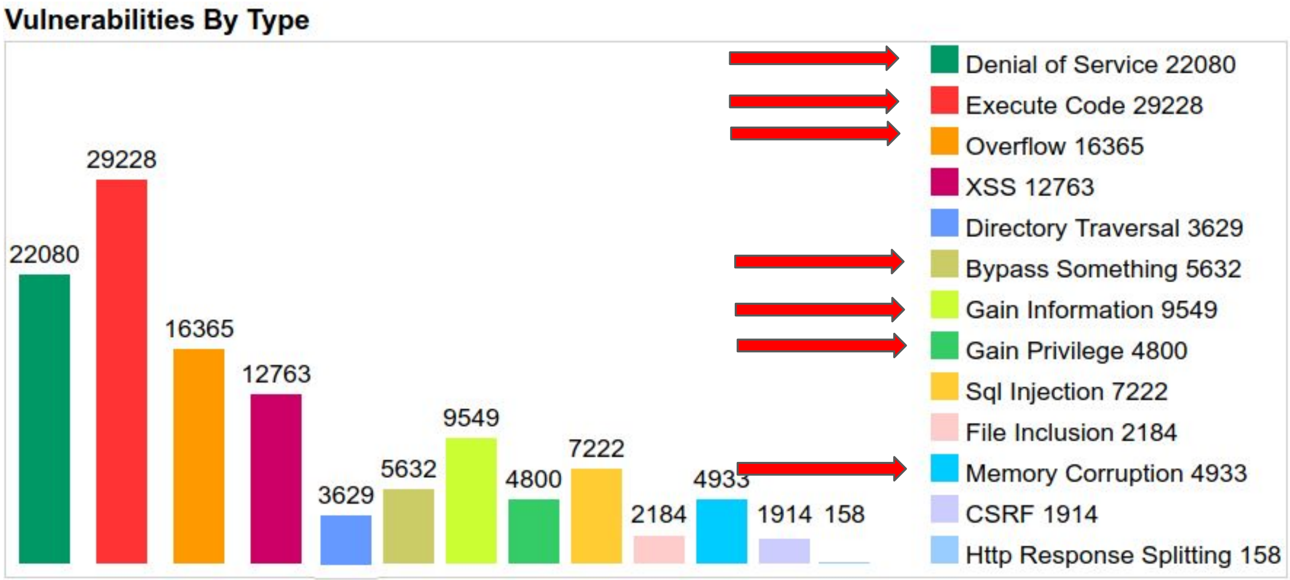
- Root Causes (Vulnerabilities)
  - Read/write out-of-bounds (OOB): heap, stack, globals
  - Read/write after-free (UAF): heap, stack
  - Read of uninitialized memory
  - Integer overflow, type confusion, data race, etc often cause OOB

- Consequences (Exploits)
  - Remote code execution
  - Information leak
  - Privilege escalation
  - VM escapes/cross-VM info leaks
  - ...
  - Safety / reliability bugs (silent data corruption, sporadic crashes)

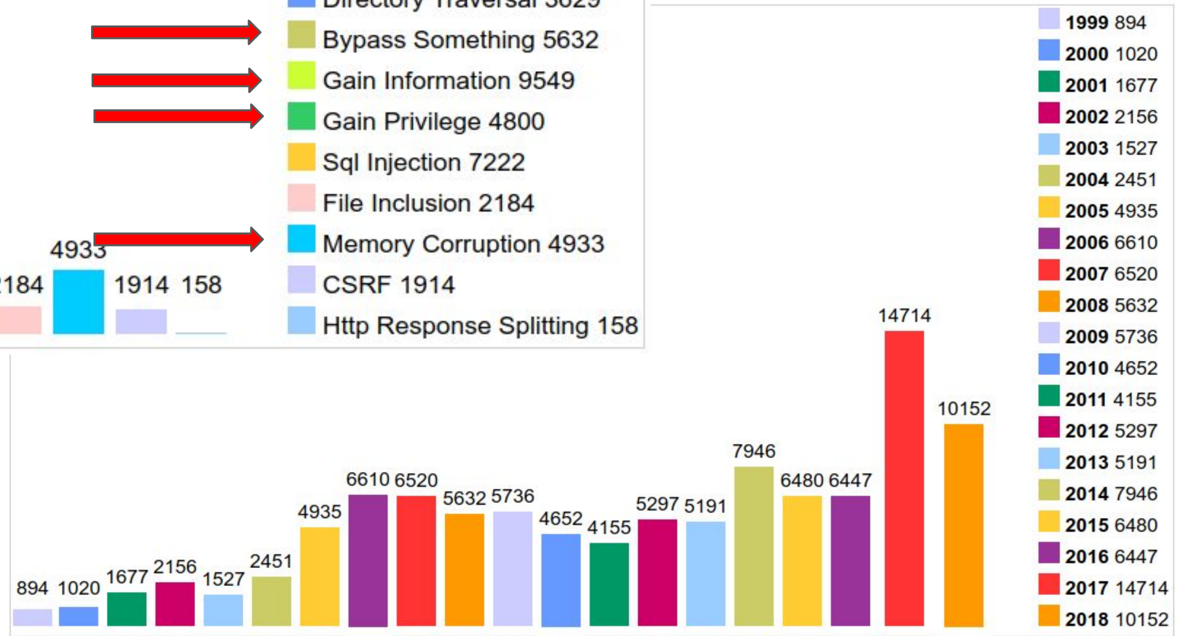
Tip of the iceberg

Bugs with names, logos, websites, wikipedia, blogs

CVEs (cvedetails.com)



By year



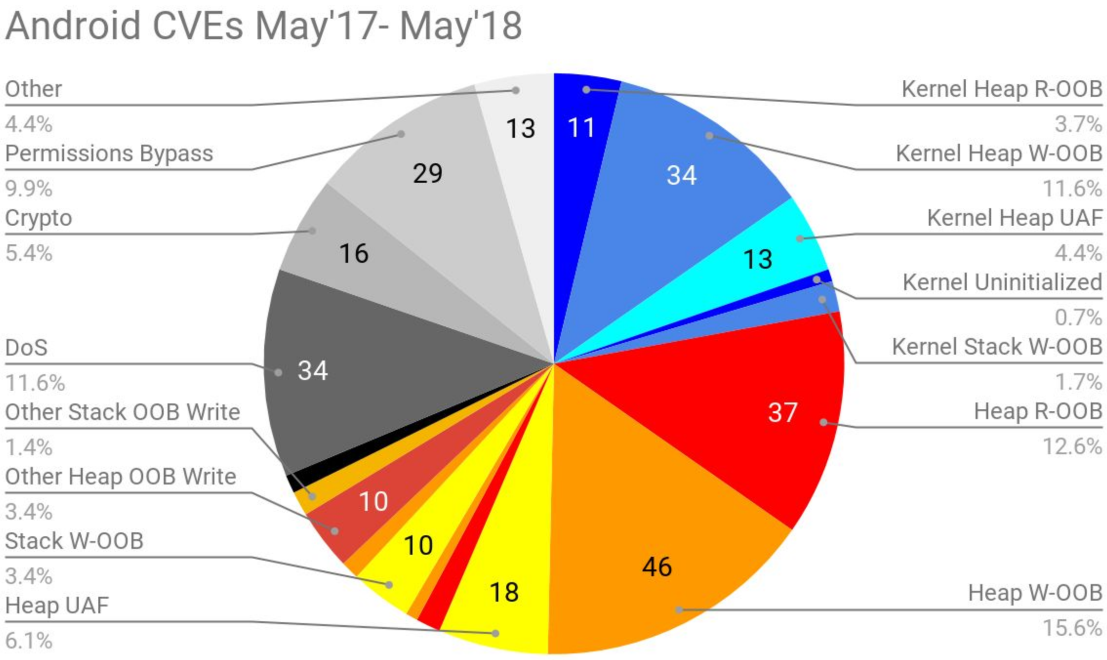
Classification by the consequence, not by the root cause

Most of the bugs are not CVEs

- Some are detected before the release
    - Affect the development cost and time
- Many SW vendors don’t submit CVEs
- Worst case: a bug is unknown to the vendor
    - But sold on the black market
    - Or just silently corrupts the data
    - Heartbleed was not known to the vendor for 2+ years

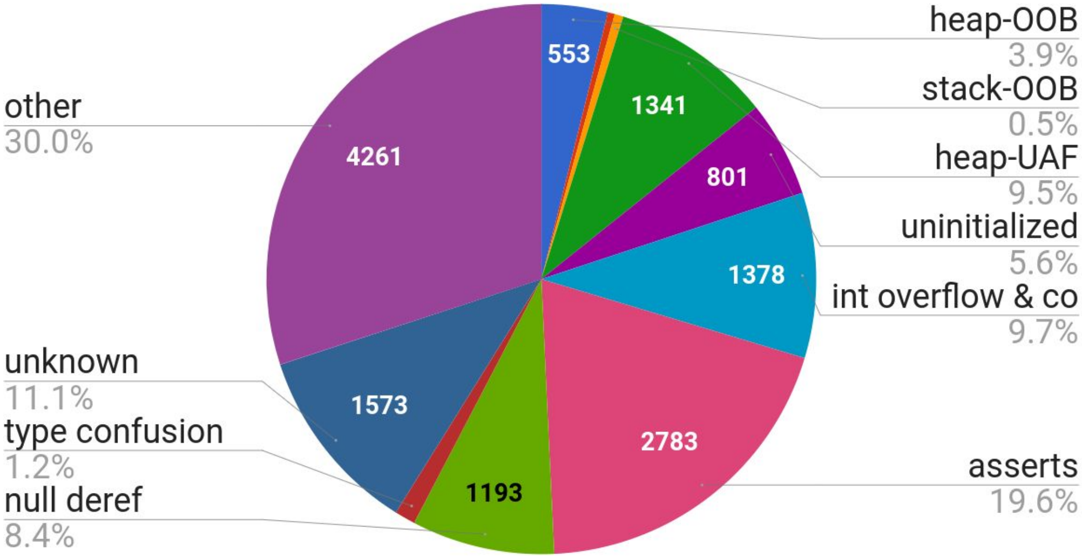
Memory Safety Horror Stories

Android CVEs (*)



(*) Source: High/Critical CVEs, May 2017- May 2018

Chrome bugs (*)



* 14K bugs found internally

* still, $4M [bug rewards] paid

* ChromeOS [pwnium chain]: 1-byte OOB => RCE under root

MT covers ~25% of all bugs

(*) Source: bugs found by Chrome's internal fuzzing since ~ 2011

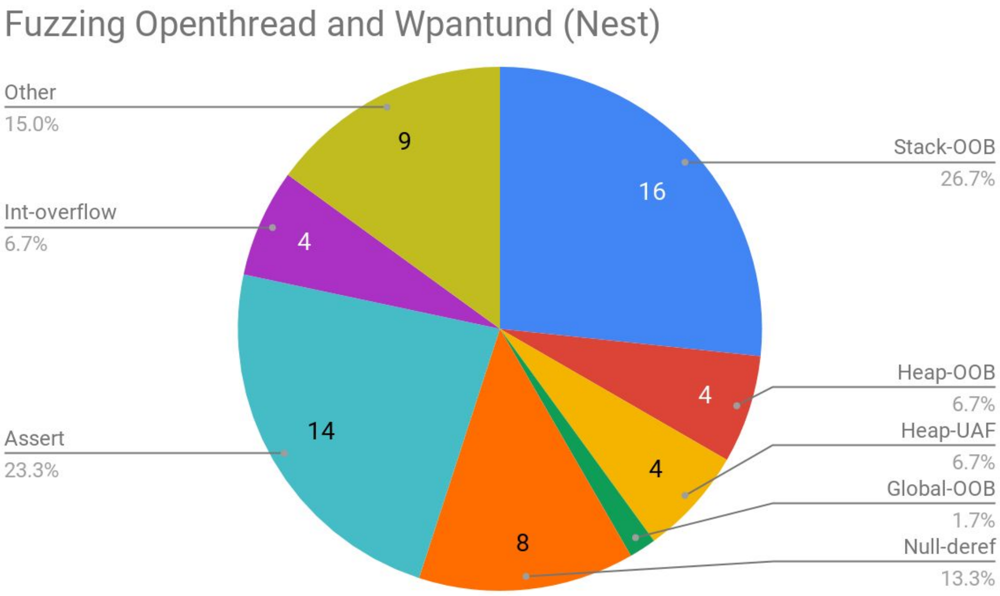
Chrome Releases July 24, 2018

MT covers all High CVEs

[$5000][850350] High CVE-2018-6153: Stack buffer overflow in Skia. *Reported by Zhen Zhou ...*
[$3000][848914] High CVE-2018-6154: Heap buffer overflow in WebGL. *Reported by Omair on 2018-06-01*
[$N/A][842265] High CVE-2018-6155: Use after free in WebRTC. *Reported by Natalie Silvanovich...*
[$N/A][841962] High CVE-2018-6156: Heap buffer overflow in WebRTC. Reported by Natalie Silvanovich ...
[$N/A][840536] High CVE-2018-6157: Type confusion in WebRTC. Reported by Natalie Silvanovich ...
[$2000][841280] Medium CVE-2018-6158: Use after free in Blink. *Reported by Zhe Jin(金哲)...*
[$2000][837275] Medium CVE-2018-6159: Same origin policy bypass in ServiceWorker.*Reported by Jun Kokatsu ...*
[$1000][839822] Medium CVE-2018-6160: URL spoof in Chrome on iOS. *Reported by evi1m0 ...*
[$1000][826552] Medium CVE-2018-6161: Same origin policy bypass in WebAudio.*Reported by Jun Kokatsu ...*
[$1000][804123] Medium CVE-2018-6162: Heap buffer overflow in WebGL. *Reported by Omair on 2018-01-21*
[$500][849398] Medium CVE-2018-6163: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-06-04*
[$500][848786] Medium CVE-2018-6164: Same origin policy bypass in ServiceWorker.*Reported by Jun Kokatsu*
[$500][847718] Medium CVE-2018-6165: URL spoof in Omnibox. *Reported by evi1m0 of Bilibili Security ...*
[$500][835554] Medium CVE-2018-6166: URL spoof in Omnibox. *Reported by Lnyas Zhang on 2018-04-21*
[$500][833143] Medium CVE-2018-6167: URL spoof in Omnibox. *Reported by Lnyas Zhang on 2018-04-15*
[$500][828265] Medium CVE-2018-6168: CORS bypass in Blink. *Reported by Gunes Acar and Danny Y. Huang of Princeton University, ...*
[$500][394518] Medium CVE-2018-6169: Permissions bypass in extension installation .*Reported by Sam P on 2014-07-16*
[$TBD][862059] Medium CVE-2018-6170: Type confusion in PDFium. *Reported by Anonymous on 2018-07-10*
[$TBD][851799] Medium CVE-2018-6171: Use after free in WebBluetooth. *Reported by amazon@mimetics.ca on 2018-06-12*
[$TBD][847242] Medium CVE-2018-6172: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-05-28*
[$TBD][836885] Medium CVE-2018-6173: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-04-25*
[$N/A][835299] Medium CVE-2018-6174: Integer overflow in SwiftShader. *Reported by Mark Brand of Google Project Zero on 2018-04-20*
[$TBD][826019] Medium CVE-2018-6175: URL spoof in Omnibox. *Reported by Khalil Zhani on 2018-03-26*
[$N/A][666824] Medium CVE-2018-6176: Local user privilege escalation in Extensions.*Reported by Jann Horn of Google Project Zero on 2016-11-18*

Every 6-8 weeks on https://chromereleases.googleblog.com, since ~ 2011

IoT (*)

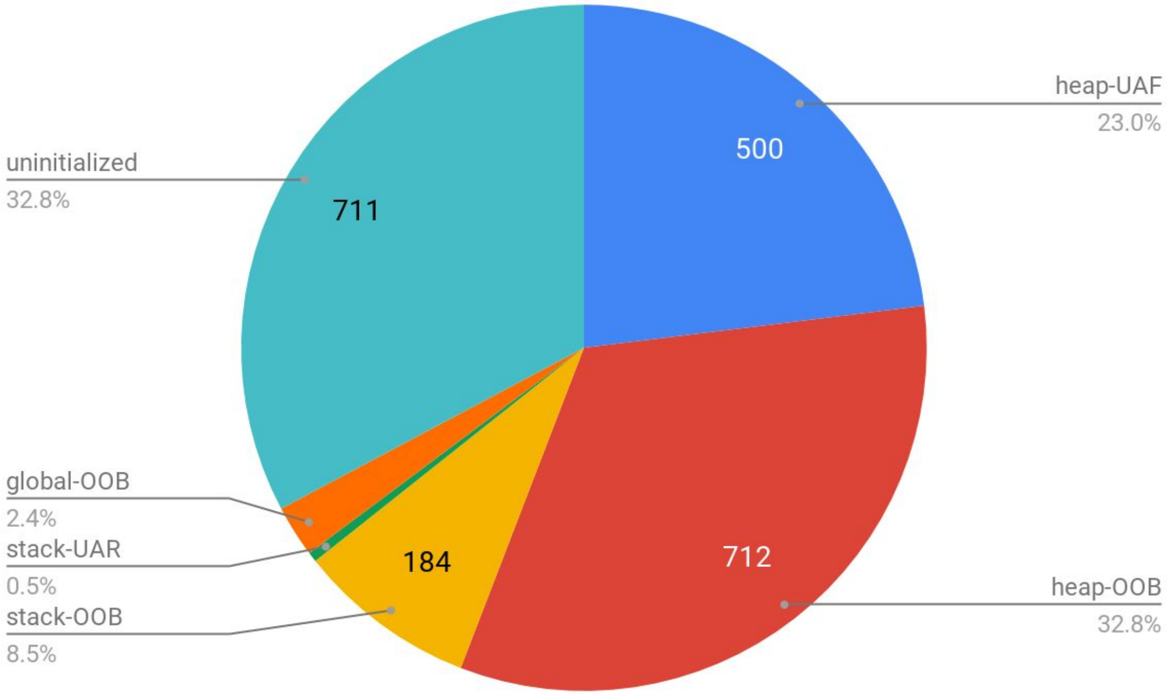


- Data is harder to find
- IoT has more shallow problems
  - Default passwords
  - Open ports
  - Can’t disable telnet
- Today, usually 32-bit anyway, MT does not apply

MT covers ~ 40% of all bugs

(*) Fuzzing [openthread and wpantund](#) (Nest) on OSS-Fuzz

Datacenter (*)

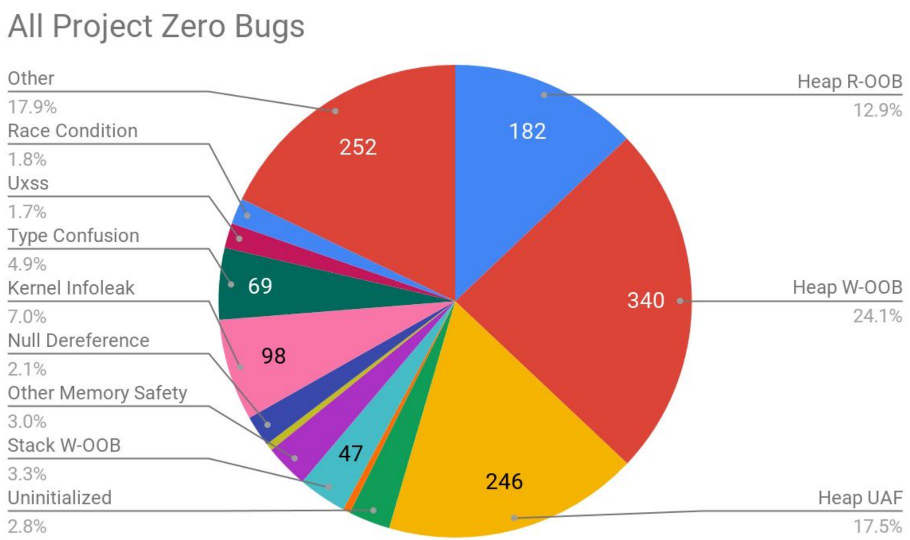


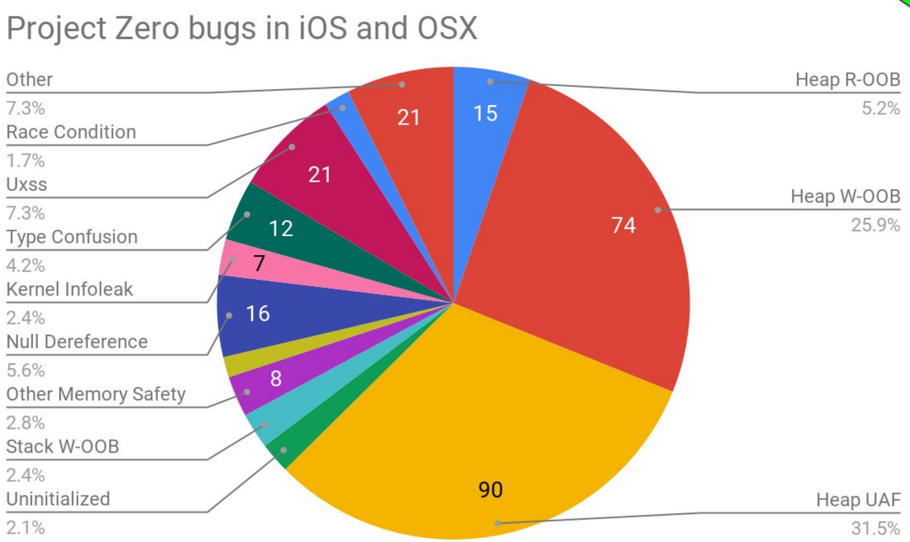
(*) Google's internal bug database, bugs found with ASan/MSan during July 2017 - July 2018. This data is far from complete.

Not any better outside of Google

Project Zero bug reports (*)

MT covers >60% reports

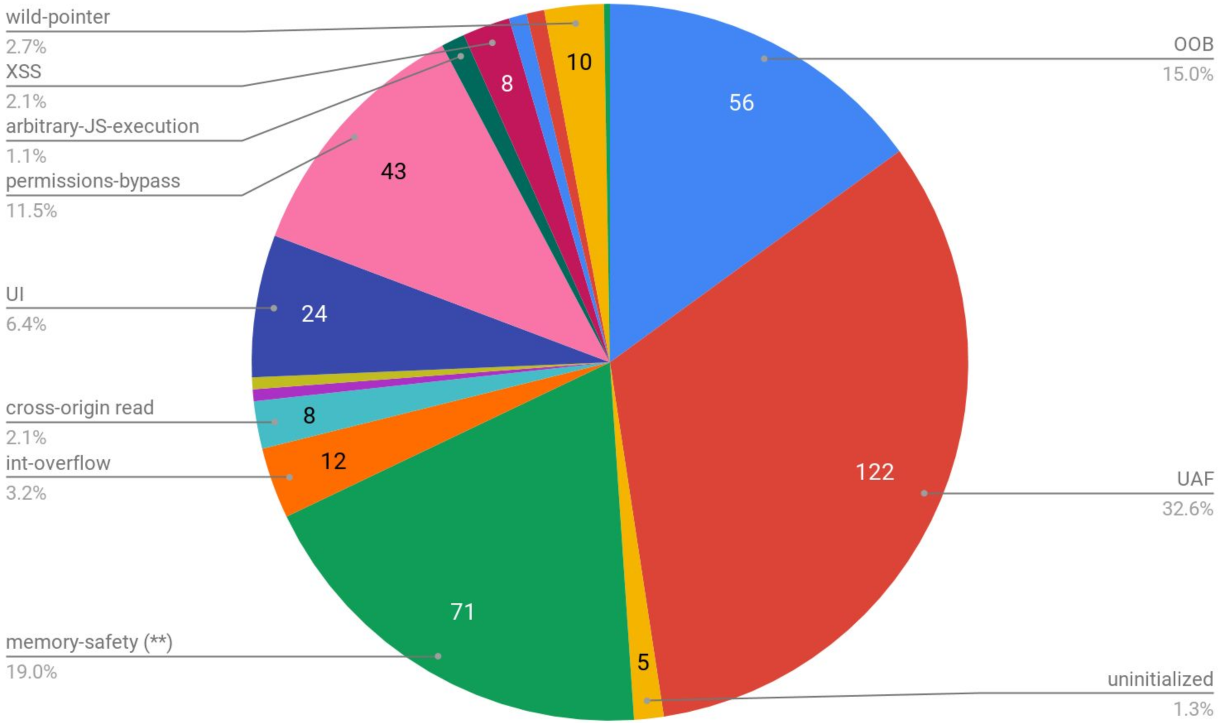




(*) Data source: all issues in project zero issue tracker as of 25th July 2018

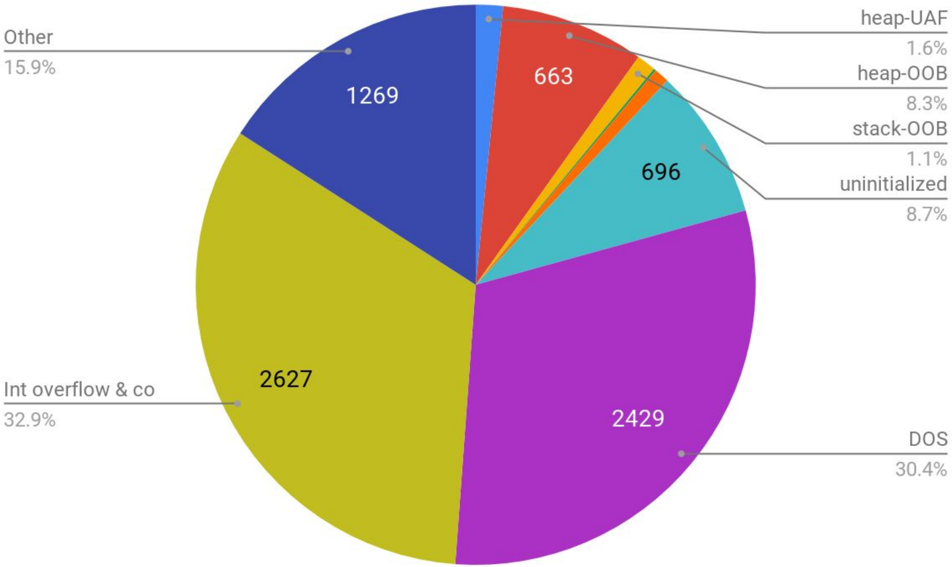
Mozilla CVEs (*)

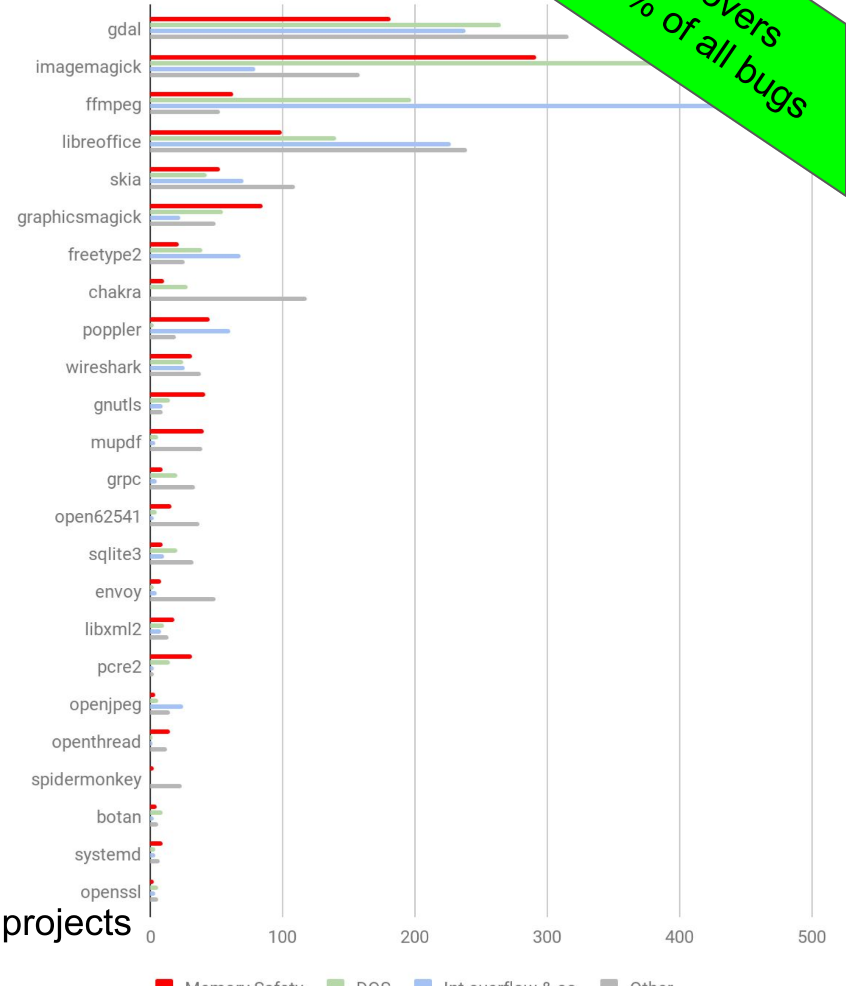
MT covers >60% CVEs



(*) Data source: bugs linked to CVEs fixed in Mozilla releases, July 2017 - July 2018.
(**) Uncategorized (restricted-access bugs, not enough data in CVEs)

OSS-Fuzz reports (*)





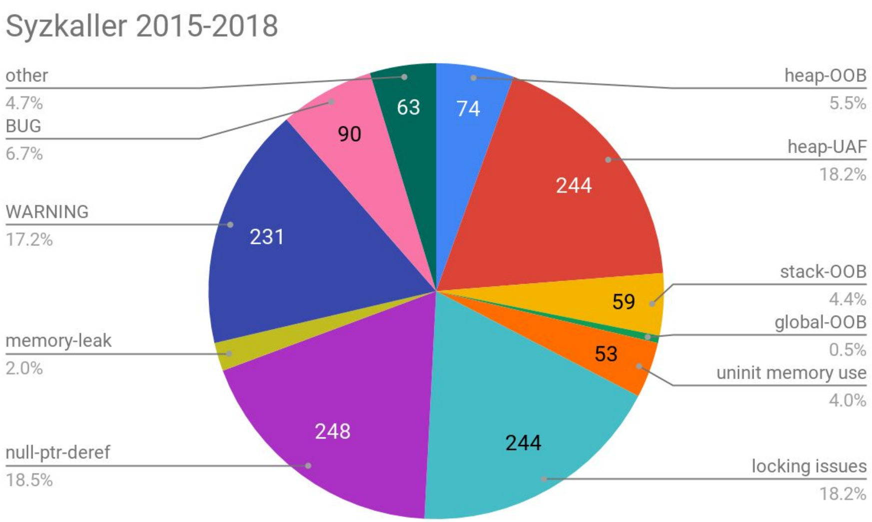
MT covers ~25% of all bugs

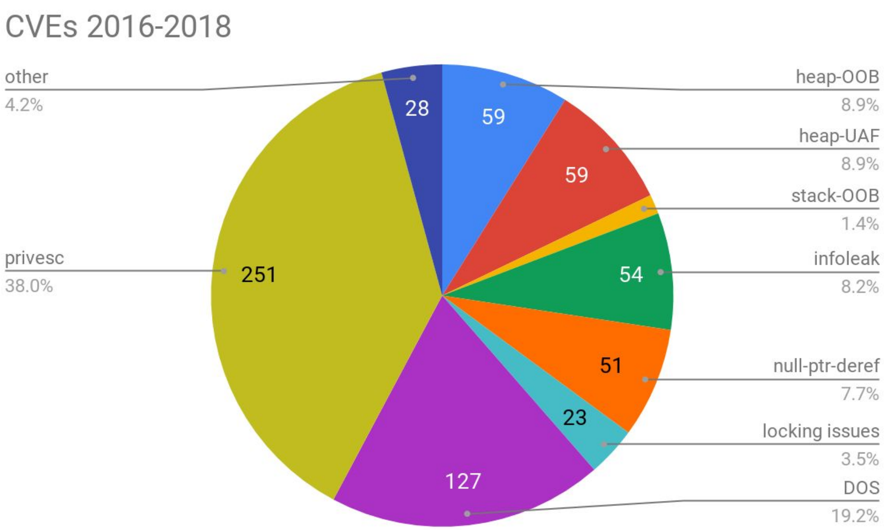
(*) Data source: OSS-Fuzz bug tracker, covering 100+ OSS projects

Not horrified enough?

Linux kernel bugs: syzkaller & CVEs (*) (**)

MT covers ~50% CVEs & ~25% bugs





(*) Source: syzkaller findings [1], [2], [3], [4]; CVEs from 2016-2018 [1], [2]

(**) Classification is non-trivial: an integer overflow can lead to a heap-OOB that allows to escalate privileges

Memory safety bugs:

- Largest portion of all security bugs
- Cloud, desktop, mobile, IoT
- Userspace, kernels & below

IoT = Internet Of Targets

ASAN

ASAN is a testing tool

- Continuous integration: pre- & post- submit testing
- Continuous automated fuzzing (see Usenix Security'17)
- Responsible for the majority of findings since 2011

ASAN in production is painfully hard

- More code size:
  - Explodes the datacenter infrastructure
  - Hard to ship to users

- More CPU:
  - Causes cluster management algorithmic problems (lame ducks)
  - More battery use

- **More RAM:**
  - Different cluster configuration
  - Doesn’t fit on client devices

Many are desperate enough to use ASAN in prod

- Google server-side: multiple teams have “ASAN production canaries”
  - Constant flow of P0 bugs not detected in testing
- Chrome: shipped SyzyASAN (Windows) for 5 years
  - ~900 actionable bug reports
- (New) Mozilla nightly ASAN build
  - Users get paid for unique bug reports
  - NOT SCAM: Browse the Web, earn money :)
  - 10% of all critical/high security bugs (5 months period)
  - 50% bugs reported once (i.e. these are rare bugs)
  - RAM is the major bottleneck

Also: GWP-ASAN

- Guarded pages + low frequency sampling
- Crowd-sourced bug detection in production
- Will find *most* heap bugs *eventually*

- But:
  - May take months to discover a bug in production
  - Unlikely to catch rare bugs
  - Not a security mitigation
  - Not suitable for stack

Memory Tagging: ARM MTE

ARM Memory Tagging Extension (MTE)

- Announced by ARM on 2018-09-17

- Doesn't exist in hardware yet
  - Will take several years to appear

- "Hardware-ASAN on steroids"
  - RAM overhead: 3%-5%
  - CPU overhead: (*hoping for)* low-single-digit %

ARM Memory Tagging Extension (MTE)

- 64-bit only
- Two types of tags
  - Every aligned 16 bytes of memory have a 4-bit tag stored separately
  - Every pointer has a 4-bit tag stored in the top byte
- LD/ST instructions check both tags, raise exception on mismatch
- New instructions to manipulate the tags

Allocation: tag the memory & the pointer

- Stack and heap

- Allocation:
  - Align allocations by 16
  - Choose a 4-bit tag (random is ok)
  - Tag the pointer
  - Tag the memory (optionally initialize it at no extra cost)

- Deallocation:
  - Re-tag the memory with a different tag

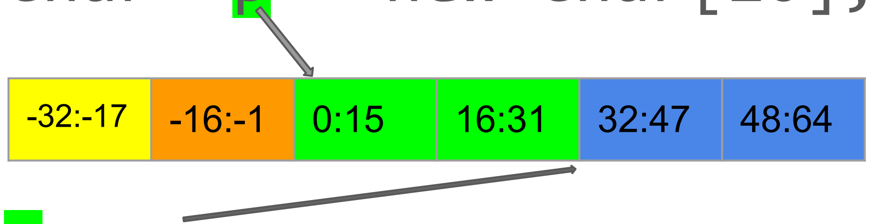
Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007ffffff1240
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007fffffff1240
```




```
p[32] = … // heap-buffer-overflow ≠
```

Heap-use-after-free

```
char *p = new char[20]; // 0xa007fffffff1240
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

Heap-use-after-free

```
char *p = new char[20]; // 0xa007fffffff1240
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

```
delete [] p;  // Memory is retagged ⇒
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

```
p[0] = ...  // heap-use-after-free ≠
```

Probabilities of bug detection

```
int *p = new char[20];

p[20]              // undetected, same granule (*)

p[32], p[-1]       // 93%-100% (15/16 or 1)

p[100500]          // 93% (15/16)

delete [] p; p[0]  // 93% (15/16)
```

Buffer overflows within a 16-byte granule

- Typically, not security bugs if heap/stack is 16-byte aligned in production
- Still, logical bugs
- Only so-so solutions for testing:
  - Malloc may optionally align right (tricky on ARM, more tricky on x86_64)
  - Put magic value on malloc, check on free (detects only overwrites, with delay)
  - Tag the last granule with a different tag, handle in the signal handler (SLOW)

MTE overhead

- Extra logic inside LD/ST (fetching the memory tag)
  - Software can’t do much to improve it (???)
- Tagging heap objects
  - CPU: malloc/free become O(size) operations
- Tagging stack objects (optional, but desirable)
  - CPU: function prologue becomes O(frame size)
  - Stack size: local variables aligned by 16
  - Code size: extra instructions per function entry/exit
  - Register pressure: local variables have unique tags, not as simple as [SP, #offset]

Usage models

- Testing in lab
  - Better & cheaper than ASAN
- **Testing in production** aka crowdsourced bug detection
  - possibly with per-process or per-allocation sampling
  - good deduplication of bug reports
- Always-on **security mitigation**
  - with per-process knobs
- MTE is a general purpose tool - other types of usage are likely to appear
  - Infinite hardware watchpoints
  - Race detection (like in DataCollider)
  - Garbage collection

Is probabilistic detection OK for security mitigation?

- Enough retries may allow an MTE bypass in some cases (e.g. UAF)

- BUT:
    - Software could block the restarts on first MTE report (i.e. no retries)
    - The vendors gets actionable bug report on first failed attempt

Google Project Zero on MTE

… we would expect the impact of such a hard mitigation on the number of available, reliably exploitable bugs would be higher than that of other soft mitigation techniques such as CFI, or fine-grained ASLR, which **target** exploitation techniques rather than the **existence of exploitable vulnerabilities in the first place**.

Legacy code

- MTE will work on legacy code w/o recompilation
  - Libc-only change
  - Will find and mitigate heap OOB & UAF (~90% of all bugs)

No more uses of uninitialized memory

- Tagging the memory during allocation also initializes it
  - MTE always-on => no more uninitialized memory
  - MTE only during testing => uninitialized memory remains

- Can initialize all memory today, at ~ the same cost as full MTE

Overhead

- RAM: 3% - 5% (measured)
- Code Size: 2%-4% (measured)
- CPU: 0% - 5% (*estimated*)
- Power: ?

But also savings

- Many existing and near-future mitigations become fully redundant
  - Stack Protector
  - Parts of Fortify
  - Hardened and specialized allocators
- Other mitigations become less critical
  - Control Flow Integrity
- Estimated performance wins from MTE are similar to its overheads

Memory Tagging: LLVM HWASAN

LLVM HWASAN (HardWare ASAN)

- Same logic as ARM MTE but 8-bit tags, **relies on ARM top-byte-ignore**
- Tag checking via compiler instrumentation, 16:1 shadow; similar to ASAN
- Today: fully instrumented Android userspace: just works, reports real bugs
- Next steps: kernel and apps

```
// int foo(int *a) { return *a; }  // clang -O2 --target=aarch64-linux -fsanitize=hwaddress -c load.c
     0:       08 00 00 90     adrp    x8, <__hwasan_shadow>
     4:       08 01 40 f9     ldr     x8, [x8]         // shadow base (to be resolved by the loader)
     8:       09 dc 44 d3     ubfx    x9, x0, #4, #52  // shadow offset
     c:       28 69 68 38     ldrb    w8, [x9, x8]     // load shadow tag
    10:       09 fc 78 d3     lsr     x9, x0, #56      // extract address tag
    14:       3f 01 08 6b     cmp     w9, w8           // compare tags
    18:       61 00 00 54     b.ne    24               // jump on mismatch
    1c:       00 00 40 b9     ldr     w0, [x0]         // original load
    20:       c0 03 5f d6     ret
    24:       40 20 21 d4     brk     #0x902           // trap
```

HWASAN vs ASAN

- HWASAN:
  - **Much smaller RAM overhead**: 6% vs 2x
  - Detection of buffer overflows far from bounds
  - Detection of use-after-free long after deallocation
- ASAN:
  - More precise 1-byte buffer-overflow detection
  - More portable (32-bit, non-aarch64)

HWASAN is infeasible on x86_64 (no top-byte-ignore)

- Compiler needs to remove the address tag before every load/store
- Nearly impossible to deploy anywhere

Memory Tagging: SPARC ADI

SPARC ADI - HW implementation, since 2016

- 4 bit tags per 64 bytes
- higher RAM consumption due to 64-byte alignment
- Heap-only, expensive for stack
- Little attention due to a small niche
- Strong feedback from those who tried

ADI: precise vs imprecise

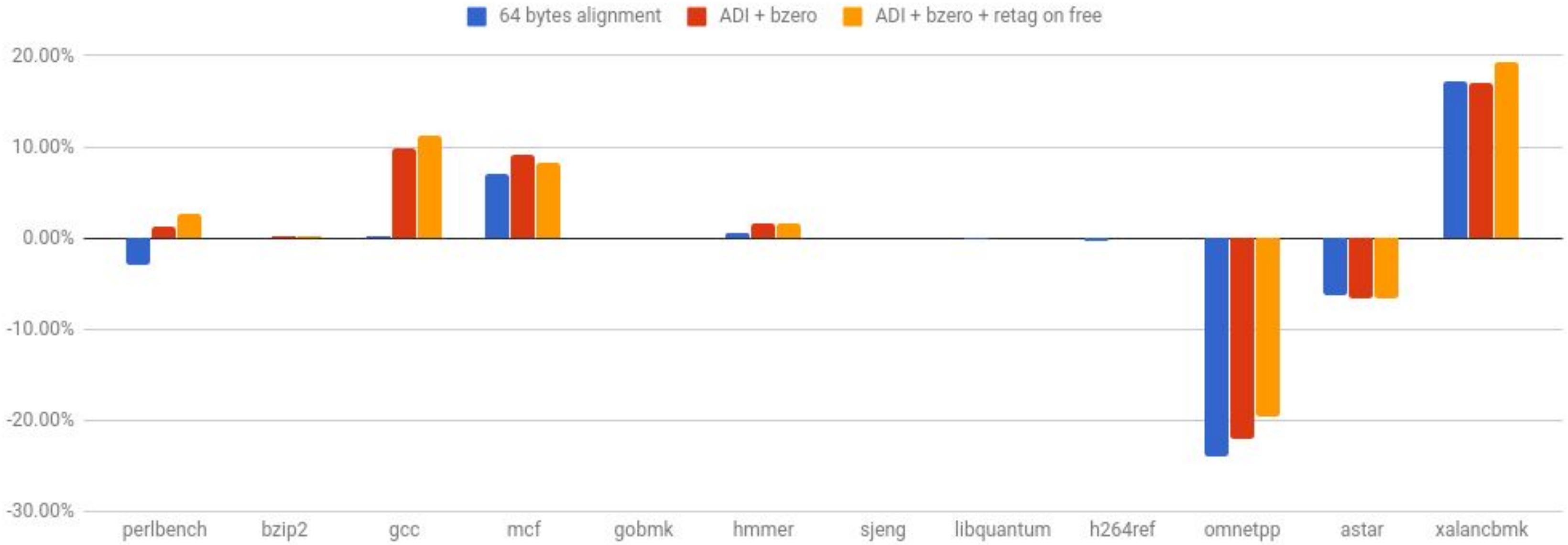
- Precise mode:
  - Tag mismatch on store causes immediate trap
  - Expensive, great for debugging

- Imprecise mode
  - Tag mismatch on store causes a trap some time later
  - Very low overhead

- Loads are always precise

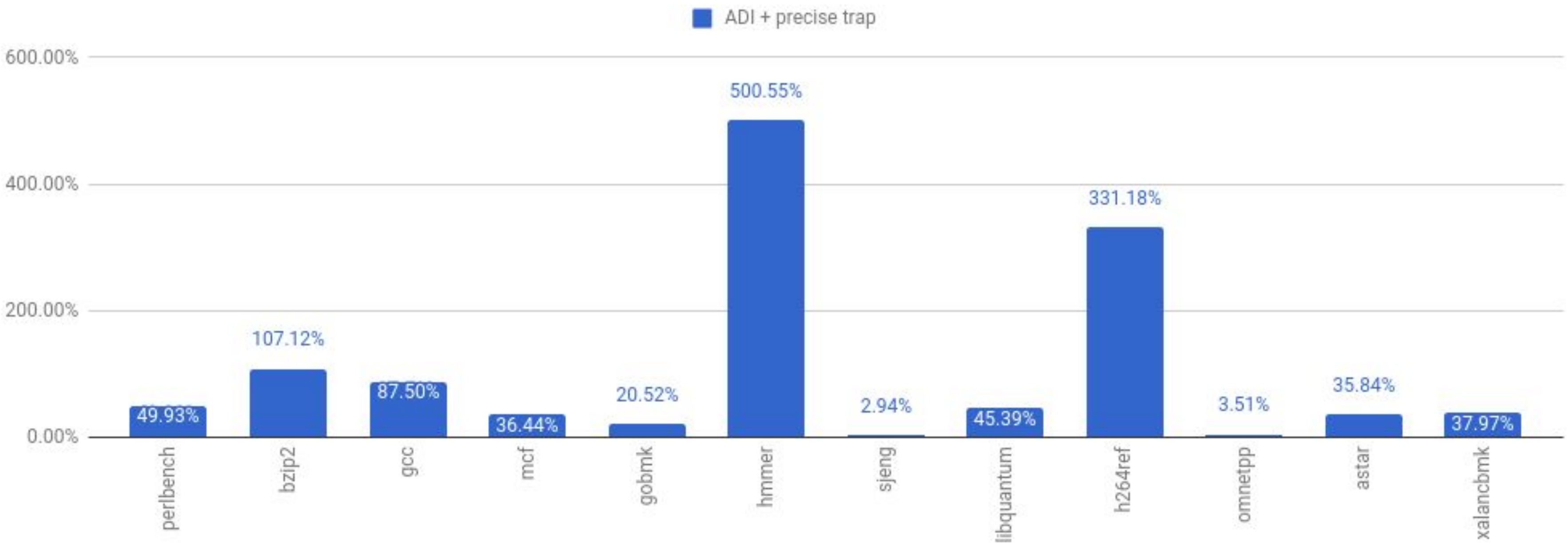
ADI overhead (imprecise)

Overhead: 64-byte alignment and (less) tagging memory on malloc



ADI overhead (precise)

Stores become very expensive



Why 16-byte granules and not 64?

Heap

- Typical Google Server app
  - Default alignment: 8 for operator new, 16 for malloc
  - Allocations dominantly tiny, < 128 bytes
    - 40 bytes: 20%, 48 bytes: 15%, 30 bytes: 15%, 56 bytes: 10%, **64 bytes: 7%**
  - Forcing 16-byte alignment costs ~2% extra RAM
  - **Forcing 64-byte alignment costs 17% extra RAM - unacceptable for production**

- Chrome Browser
  - Default alignment: 16
  - Heavily dominated by tiny allocations
  - **64-byte alignment causes 25%+ extra RAM - unacceptable for production**

Stack

- Typical Google Server app
  - 64K stacks
  - Thousands of threads, can’t increase the stack size
  - 16-byte alignment: ~5% frame size growth, tolerable
  - 64-byte alignment: ~25% frame size growth, unacceptable

- Chrome Browser
  - 1M+ stacks
  - Hundreds of threads
  - 16-byte alignment: ~4% frame size growth, tolerable
  - 64-byte alignment: ~31% frame size growth, unacceptable

16-byte granules is the sweet spot

- 64-byte granules / 4 bit tags (SPARC ADI)
  - < 1% RAM for tag storage
  - 17%-25% RAM for heap over-alignment
  - 25%-31% stack size growth
- 16-byte granules / 4 bit tags (ARM MTE)
  - 3% RAM for tag storage
  - 0%-2% RAM for heap over-alignment
  - 4%-5% stack size growth
- 8-byte granules / 4 bit tags (hypothetical)
  - 6% RAM for tag storage
  - 0% other overhead

Alternatives to Memory Tagging? (spoiler: none)

- Memory safe languages (Rust? Java? Go? Swift? C#?)
  - Safer than C/C++, but not really safe. And C/C++ won't disappear any time soon
  - Safety is achieved by run-time checks (slow) and, except Rust, by GC (more RAM)
  - May cause further fragmentation, increase development costs
- Fat pointers (MPX, CHERI): too much RAM, doesn't directly address UAF, harder to deploy
- "Constant Vigilance" and better testing (ASAN & Co): sure, but not enough
- Keep piling up soft mitigations (CFI, ASLR, XOM, hardened malloc,...)
  - Already cost more than MTE, but "defence-in-depth"
- Other hardware proposals, e.g. REST or LowFat. Weaker than MTE

Call to Action

- Top-byte-ignore (4+ bits is ok)
  - Material improvements in testing process (ASAN => HWASAN)
  - Wider use of production canaries (server & client)
  - Preparation for memory tagging
  - Already running HWASAN on Android Pixel 2/3, i.e. X86_64 is far behind
- Memory Tagging @ <50% CPU overhead (*16 byte granules*)
  - Drastic improvements in production canaries
  - Niche deployments in production
  - Deployed for subsets of Chrome code and/or users
- Memory Tagging @ <10% CPU overhead
  - Shipping memory-safe Chrome to everyone becomes feasible
  - Applicable to large subsets of security-critical data-center code
- Memory Tagging @ <2% CPU overhead
  - Not worth discussing now, give us the first three, please!

Summary

- Memory Safety bugs are the largest source of security & correctness bugs
- Hardware Memory Tagging is the only solution on the horizon
- Top-byte-ignore is the first easy step, very useful by itself

backup

Complement: Control Flow Integrity (CFI)

- Forward-CFI: prevent virtual/indirect calls hijacking
  - LLVM's CFI: partially deployed in Android P, more to go
  - BTI / ENDBRANCH (aka Landing Pads): may improve performance, unlikely improve security
- Backward-CFI: prevent return address hijacking
  - LLVM's Shadow Call Stack: investigating for Android Q
  - PAC: may be slightly more secure and/or faster (pros and cons)
  - CET (shadow call stack): probably the strongest solution

- CFI is great defence-in-depth complement to MTE, very nice to have!
  - **MTE w/o CFI is much better than CFI w/o MTE**
- CFI is not sufficient w/o MTE
  - Blocks only *some* of the exploitation techniques
  - Doesn't address the root cause
  - Doesn't help stability

New MTE instructions (docs, LLVM patch)

IRG Xd, Xn

Copy Xn into Xd, insert a random 4-bit tag into Xd

ADDG Xd, Xn, #<immA>, #<immB>

Xd := Xn + #immA, with address tag modified by #immB.

bit manipulations with the address tag

STG [Xn], #<imm>

Set the memory tag of [Xn] to the tag(Xn)

STGP Xa, Xb, [Xn], #<imm>

Store 16 bytes from Xa/Xb to [Xn] and set the memory tag of [Xn] to the tag(Xn)

storing the memory tag