Memory Categorization
Separating Attacker-Controlled Data

Matthias Neugschwandtner
Alessandro Sorniotti
Anil Kurmus

IBM Research - Zurich

16th Conference on Detection of Intrusions and Malware & Vulnerability Assessment; Gothenburg, June 19-20

Memory Safety - Approaches

- Ensure temporal and spatial memory safety
  - managed runtimes (Java)
  - native code (SoftBounds)
  - hardware support (MPX)
- Mitigate memory violations
  - control flow integrity
  - data flow integrity
- Runtime checks cause overhead
  - optimizations for performance-critical code
    - ASAP, SplitKernel, PartiSan, BinRec
- Optimize based on data!

Memory Categorization

Attacker-Controlled Data

- Untrusted data
  - Input read from Network

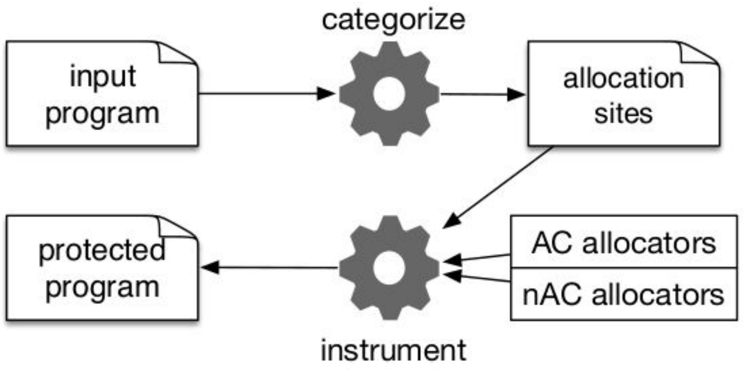
Non Attacker-Controlled Data

- Program internal data
  - Memory addresses
- Trusted data
  - Cryptographic material
  - Configuration read from disk

- Separate AC from nAC data
- Attacker only has access to their own data
- Loose form of memory safety by itself
- Enables mitigations based on selective hardening

Memory Categorization

I. Provide separate allocators
II. *Categorize*
decide which allocator should be used
III. *Instrument*
implement decision in program



Separate Allocators

- Stack allocators
  - nAC and AC allocators
- Heap allocators
  - nAC and AC allocators
  - “mixed” allocator
    - Complex data structures (list item: metadata + content, packet: header + payload)
    - Custom memory managers (single large allocated chunk of memory)
- Allocation sites
  - Location where allocator is invoked
  - Stack allocations
    - limited in scope to current function → intraprocedural
  - Heap allocations
    - long(er)-lived
    - depends on calling context → interprocedural
    - allocation wrappers, e.g. `xmalloc()`

Label Allocation Sites

I. Identify AC data sources
II. Track pointers backwards
III. Find allocation sites

```
1 char *cmalloc (int sz) {
2     if (sz == 0) return NULL;
3     return (char *)malloc(sz);
4 }

5 int main (int argc, char**argv) {
6     int fd = open (argv[1], O_RDONLY);
7     char *buf = cmalloc(10);
8     read(fd, buf, 10);
9 }
```

AC allocation site
Context: 7, 3

Static Analysis

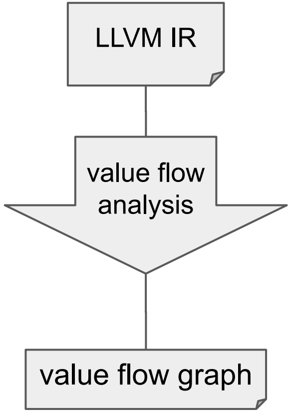
- Andersen's points-to analysis
  - field-sensitive, but context- and flow insensitive
  - field-sensitivity required for structs and classes with both AC & nAC fields
  - "partitioning" for SVF
- Sparse Value-Flow analysis
  - produces mSSA (memory single-static-assignment) form of the program
  - pointer dereference (load of address-taken variable) = USE
  - pointer assignment (store of address-taken variable) = DEF + USE
  - function callsite (for function operating on address-taken variable) = (DEF +) USE
- Sparse Value-Flow-Graph
  - combines SSA and mSSA to an interprocedural flow graph
  - nodes = variable definitions
  - edges = value flow dependencies
- Context-sensitive backward traversal through VFG

SVF: https://llvm.org/devmtg/2016-03/Presentations/SVF_EUROLLVM2016.pdf

Dynamic Analysis

- Fills in gaps of static analysis
  - e.g., because of dynamically loaded code, limits of points-to analysis
  - limited to heap allocations
- Intercept allocators
  - unwind call stack to obtain context information
  - allocate memory on “limbo” heap, annotate with context
- Intercept memory access
  - write access to limbo heap
  - categorize allocation context of corresponding memory region based on access

MemCat Compiler Pass

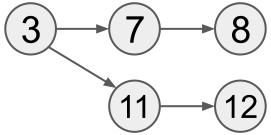


- Clang/LLVM LTO compiler pass
- Client for SVF
  - constructs value-flow-graph
  - value flows
    - direct: top-level pointers
    - indirect: address-taken pointers
    - interprocedural

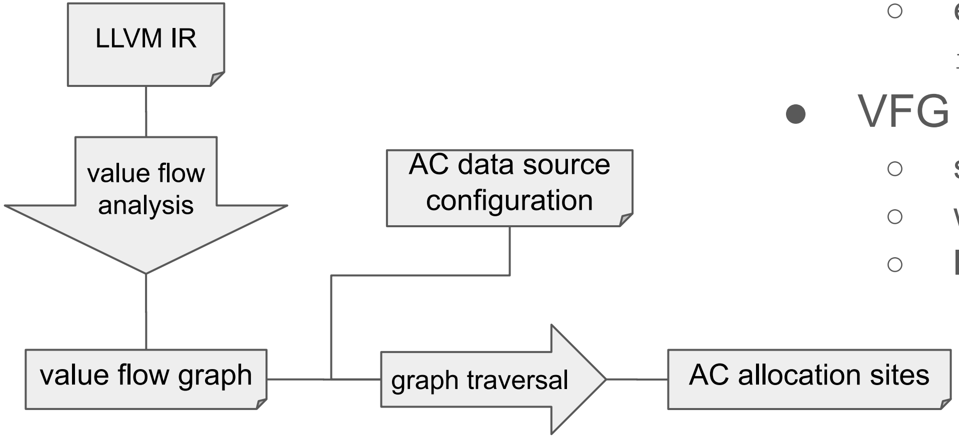
```
char *cmalloc (int sz) {
    if (sz == 0) return NULL;
    return (char *)malloc(sz);
}

void A () {
    int fd = open(...);
    char *buf = cmalloc(10);
    read(fd, buf, 10);
}

void B(char *foo) {
    char *tmp = cmalloc(20);
    strcpy(tmp, foo);
}
```



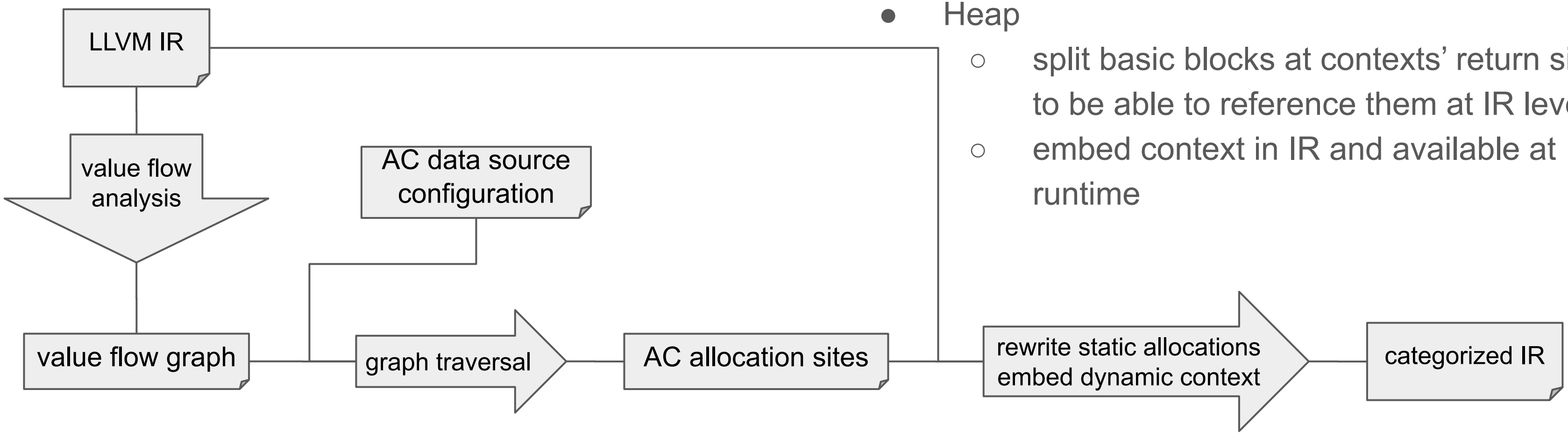
MemCat Compilation Pass



- Look for AC data sources
  - source function return values / output parameters
  - e.g., `fgetc, fgets, fread, fscanf, pread, read, recv*`
- VFG traversal
  - start from node representing source
  - worklist-style backward traversal
  - label encountered allocation sites
    - flag stack allocations
    - record context for heap allocations

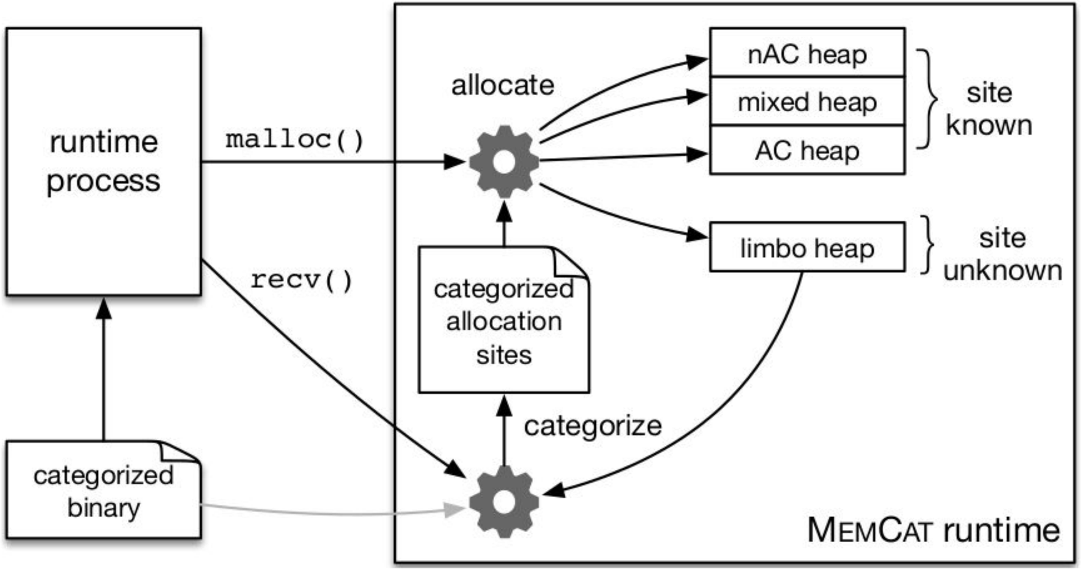
MemCat Compilation Pass

- Stack
  - rewrite allocations
  - safestack implementation
- Heap
  - split basic blocks at contexts’ return sites to be able to reference them at IR level
  - embed context in IR and available at runtime



MemCat Runtime

- Read categorized allocation sites from the binary
- Intercept allocators
  - site known → serve memory from corresponding heap
  - site not known → serve from limbo heap
- Intercept limbo heap writes
  - categorize based on data source (code) that is writing



MemCat Runtime

- Modified ptmalloc2
  - providing three arena pools
  - hardened allocator based on `mmap` + guard pages, mitigates
    - uninitialized data leaks
    - linear buffer overflows
    - double free
- Identifying context
  - stack unwinding, depth configurable
  - 8-byte context hash for fast matching
  - categorization cached across runs on disk

MemCat Runtime - Limbo Heap

- Limbo heap
  - read-only memory mappings
  - trap on access
    - remove protection
    - re-execute faulting instruction
    - categorize
    - reprotect
- Categorization termination heuristics
  - stop at program termination
  - stop after N writes
  - stop as soon as all bytes have been written
    - special handling of `memset` and `bzero`

MemCat Runtime - Indirect Categorization

- Intercept AC data sources
  - keep record of caller and targeted memory region
- additional check on limbo heap traps:
  - if caller in a record is part of the context AND
  - memory source matches record THEN
  - inherit categorization of the original record

Evaluation - Use Cases

Vulnerability	Type	Program	Categorization
CVE-2012-0920	use-after-free	Dropbear	AC
CVE-2014-0160	buffer overread	OpenSSL	mixed
CVE-2016-6309	use-after-free	OpenSSL	AC
CVE-2016-3189	use-after-free	bzip2	AC

Evaluation - Dropbear

- Small SSH server, part of busybox
- CVE-2012-0920
  - use-after free
  - allows for RCE by removing limitation on `char *forced_command`
- MemCat
  - configured to consider `read()` from network as AC
  - categorizes 4 allocation sites connected to `read_packet()` as AC at compile time
  - 3 allocation sites categorized at runtime as mixed
  - mitigates vulnerability because `forced_command` allocation resides on nAC heap

Evaluation - OpenSSL

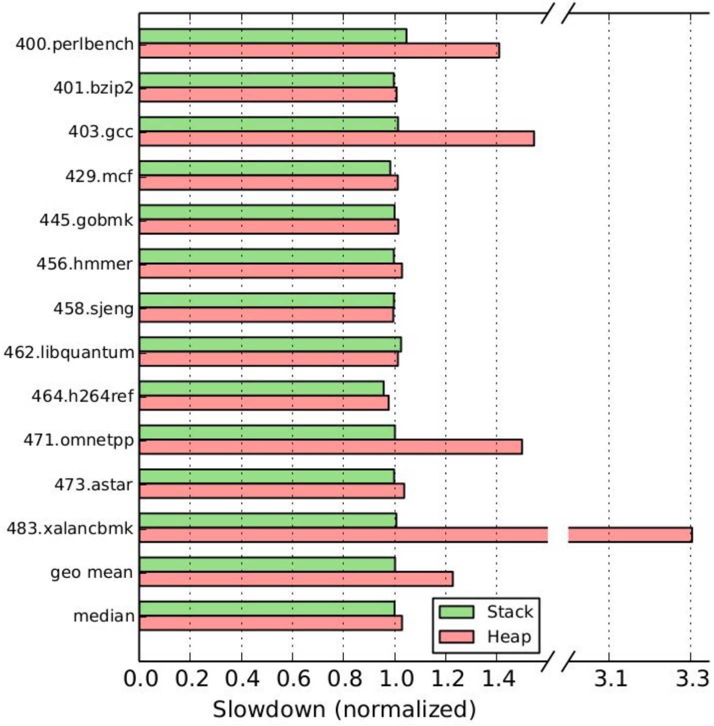
- CLI tool in server mode, perform TLS 1.2 handshake
  - performs all relevant operations (key agreement, hashing and (asymmetric) encryption, record parsing and I/O handling)
- MemCat compile time
  - 22 data sources providing AC input
  - Stack: 551 out of 3648 allocations AC
  - Heap: 1724 allocation sites AC
- MemCat runtime
  - categorization
    - 1st handshake: 1967 limbo, 5 AC, 38 mixed
    - 2nd handshake: 4 limbo, 5 AC, 39 mixed
  - 2.3% performance overhead on 2nd handshake

Evaluation - OpenSSL

- CVE-2016-6309 use-after-free
  - reallocation of the message-receive buffer leaves dangling pointers
  - allocation is AC → UAF limited to AC heap data (or entirely prevented)
- CVE-2014-0160 buffer overread (Heartbleed)
  - receive buffer is on AC heap → limited to AC (or entirely prevented)

Evaluation - Performance

- SpecINT 2006 CPU Clang/LLVM with LTO
- AC sources
  - `(f)read`
  - `recv(from)`
  - `(f)gets`
- `483.xalancbmk`
  - no points-to data for the pointer associated with the data source
- `462.libquantum`
  - does not use any of the preconfigured data sources



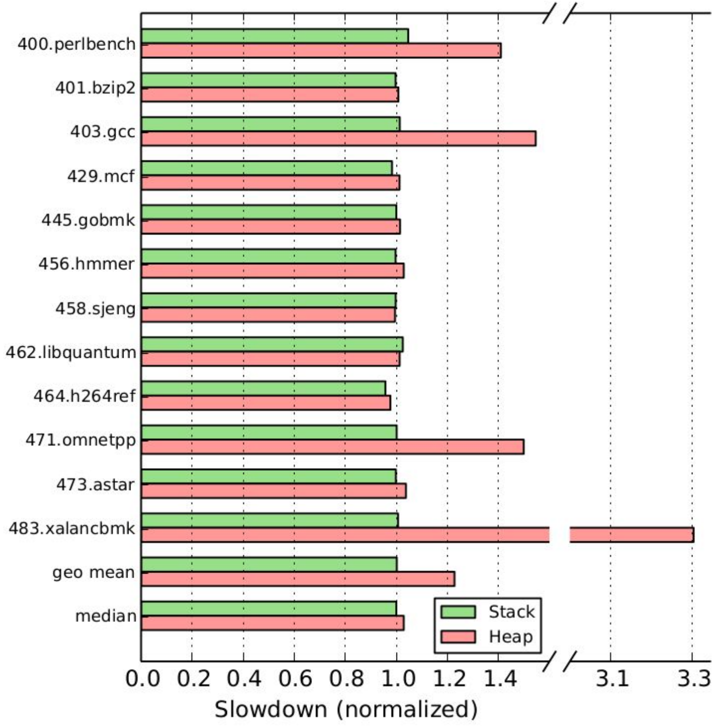
Evaluation - Performance

	compile time analysis			runtime allocations		
Benchmark	AC input	Stack AC	Heap AC	nAC	mixed	AC
perlbench	7	124	31	9185	9	15
bzip2	1	0	3	9	0	3
gcc	4	2	5	266404	1	0
mcf	3	1	1	6	0	0
gobmk	10	5	1	3672	0	0
hmmer	119	38	2525	83	1	65
sjeng	5	2	1	5	0	0
libquantum	0	0	0	7	0	0
h264ref	4	0	2	157	1	2
omnetpp	6	2	2	10305	0	0
astar	27	2	4	181	0	3
xalancbmk	1	0	0	4832	3	0

Evaluation - Performance

- Stack categorization: 0.1% overhead
- Heap categorization: 21% overhead
  - hardened heap can be implemented much more efficiently
  - higher overhead on benchmarks with many allocations or deep callstacks (limited to 20 frames)

Benchmark	allocation count	call stack depth
400.perlbench	$> 56M$	> 60
401.bzip2	28	4
403.gcc	$> 2.9M$	> 50
429.mcf	3	4
445.gobmk	$> 118K$	> 20
456.hmmer	$> 1M$	6
458.sjeng	4	4
462.libquantum	179	9
464.h264ref	5683	10
471.omnetpp	$> 267M$	10
473.astar	$> 1.1M$	6
483.xalancbmk	$> 135M$	> 6000



Discussion

- Inaccurate Categorization
    - MemCat errs on the safe side
        - nAC data might end up on AC heap
    - heap hardening can still protect nAC data on AC heap
        - worse in terms of performance, but not in terms of security
    - hardened mixed heap
- Sensitive AC data
    - multi-tenant setup requires multiple AC heaps to isolate tenants
- Propagating categorization
    - propagating categorization results using taint tracking
    - orthogonal, this work focuses on performing the initial categorization step

Conclusions

- Memory categorization
    - analyzes and labels memory allocation sites based on use in the program
    - separates attacker-controlled data
    - follows up on isolated heap by Microsoft and Adobe
- Provides loose form of memory safety on its own
- Enables selective hardening based on data
    - hardened allocators (electric fence, DieHarder, …)
    - selective instrumentation for (full) memory safety