Dangless Malloc

Mitigating dangling pointer bugs – thesis presentation

Gábor Kozár

2018 September 24

VU University Amsterdam, University of Amsterdam

Table of contents

Motivation: dangling pointers

Dangling pointers

```c
unsigned long *p = malloc(sizeof(unsigned long));
*p = 0xBADF00D;

// ... use p ...

free(p);

// ...

printf("Magic: %lu\n", *p);
```

Dangling pointers and memory re-use

```c
unsigned long *p = malloc(sizeof(unsigned long));
*p = 0xBADF00D;

// ... use p ...

free(p);

// ...

// perform another allocation
// memory can be re-used!
char *surprise = malloc(32);
strcpy(surprise, "MAGIC!!");

// ...

printf("Magic: %lx\n", *p); // 0x2121434947414d
```

Motivation: dangling pointers

As security vulnerability

Dangling pointers as security vulnerabilities

```c
char *x; size_t x_len;

void write_x(char *data) {
  size_t len = strlen(data);
  if (x_len <= len) {
    x_len = len + 1;
    x = realloc(x, x_len);
  }
  strcpy(x, data);
}

void process_x(void) {
  // ...
  free(x);
}
```

```c
struct {
  char p[10];
  int is_admin;
} *y;

void write_y(char *data) {
  if (!y)
    y = calloc(sizeof(*y));

  if (strlen(data) >= 10)
    return;

  strcpy(y->p, data);
  process_cmd(y->p, y->is_admin);
}
```

1. `write_x("dummy 1 + padding")`
2. `process_x()`
3. `write_y("dummy 2")`
4. `write_x("fill buff PWND")`
5. `write_y("drop_db")`

- allocate x, set x_len to big enough
- free x, it is now dangling
- allocate y, memory is re-used: x = y
- write to y, bypassing length check
- enjoy your modified y->is_admin

Dangling pointers as security vulnerabilities

```c
char *x; size_t x_len;

void write_x(char *data) {
  size_t len = strlen(data);
  if (x_len <= len) {
    x_len = len + 1;
    x = realloc(x, x_len);
  }
  strcpy(x, data);
}

void process_x(void) {
  // ...
  free(x);
}
```

```c
struct {
  char p[10];
  int is_admin;
} *y;

void write_y(char *data) {
  if (!y)
    y = calloc(sizeof(*y));

  if (strlen(data) >= 10)
    return;

  strcpy(y->p, data);
  process_cmd(y->p, y->is_admin);
}
```

1. `write_x("dummy 1 + padding")`
2. `process_x()`
3. `write_y("dummy 2")`
4. `write_x("fill buff PWND")`
5. `write_y("drop_db")`

- allocate x, set `x_len` to big enough
- free x, it is now dangling
- allocate y, memory is re-used: `x = y`
- write to y, bypassing length check
- enjoy your modified `y->is_admin`

Dangling pointers as security vulnerabilities

```c
char *x; size_t x_len;

void write_x(char *data) {
  size_t len = strlen(data);
  if (x_len <= len) {
    x_len = len + 1;
    x = realloc(x, x_len);
  }
  strcpy(x, data);
}

void process_x(void) {
  // ...
  free(x);
}
```

```c
struct {
  char p[10];
  int is_admin;
} *y;

void write_y(char *data) {
  if (!y)
    y = calloc(sizeof(*y));

  if (strlen(data) >= 10)
    return;

  strcpy(y->p, data);
  process_cmd(y->p, y->is_admin);
}
```

1. `write_x("dummy 1 + padding")`
2. `process_x()`
3. `write_y("dummy 2")`
4. `write_x("fill buff PWND")`
5. `write_y("drop_db")`

- allocate x, set `x_len` to big enough
- free x, it is now dangling
- allocate y, memory is re-used: `x = y`
- write to y, bypassing length check
- enjoy your modified `y->is_admin`

Dangling pointers as security vulnerabilities

```c
char *x; size_t x_len;

void write_x(char *data) {
  size_t len = strlen(data);
  if (x_len <= len) {
    x_len = len + 1;
    x = realloc(x, x_len);
  }
  strcpy(x, data);
}

void process_x(void) {
  // ...
  free(x);
}
```

```c
struct {
  char p[10];
  int is_admin;
} *y;

void write_y(char *data) {
  if (!y)
    y = calloc(sizeof(*y));

  if (strlen(data) >= 10)
    return;

  strcpy(y->p, data);
  process_cmd(y->p, y->is_admin);
}
```

1. `write_x("dummy 1 + padding")`
2. `process_x()`
3. `write_y("dummy 2")`
4. `write_x("fill buff PWND")`
5. `write_y("drop_db")`

- allocate x, set x_len to big enough
- free x, it is now dangling
- allocate y, memory is re-used: x = y
- write to y, bypassing length check
- enjoy your modified y->is_admin

Dangling pointers as security vulnerabilities

```c
char *x; size_t x_len;

void write_x(char *data) {
  size_t len = strlen(data);
  if (x_len <= len) {
    x_len = len + 1;
    x = realloc(x, x_len);
  }
  strcpy(x, data);
}

void process_x(void) {
  // ...
  free(x);
}
```

```c
struct {
  char p[10];
  int is_admin;
} *y;

void write_y(char *data) {
  if (!y)
    y = calloc(sizeof(*y));

  if (strlen(data) >= 10)
    return;

  strcpy(y->p, data);
  process_cmd(y->p, y->is_admin);
}
```

1. `write_x("dummy 1 + padding")`
2. `process_x()`
3. `write_y("dummy 2")`
4. `write_x("fill buff PWND")`
5. `write_y("drop_db")`

- allocate x, set `x_len` to big enough
- free x, it is now dangling
- allocate y, memory is re-used: `x = y`
- write to y, bypassing length check
- enjoy your modified `y->is_admin`

Dangling pointers as security vulnerabilities

```c
char *x; size_t x_len;

void write_x(char *data) {
  size_t len = strlen(data);
  if (x_len <= len) {
    x_len = len + 1;
    x = realloc(x, x_len);
  }
  strcpy(x, data);
}

void process_x(void) {
  // ...
  free(x);
}
```

```c
struct {
  char p[10];
  int is_admin;
} *y;

void write_y(char *data) {
  if (!y)
    y = calloc(sizeof(*y));

  if (strlen(data) >= 10)
    return;

  strcpy(y->p, data);
  process_cmd(y->p, y->is_admin);
}
```

1. `write_x("dummy 1 + padding")`
2. `process_x()`
3. `write_y("dummy 2")`
4. `write_x("fill buff PWND")`
5. `write_y("drop_db")`

- allocate x, set `x_len` to big enough
- free x, it is now dangling
- allocate y, memory is re-used: `x = y`
- write to y, bypassing length check
- enjoy your modified `y->is_admin`

Dangling pointers as security vulnerabilities

```c
char *x; size_t x_len;

void write_x(char *data) {
  size_t len = strlen(data);
  if (x_len <= len) {
    x_len = len + 1;
    x = realloc(x, x_len);
  }
  strcpy(x, data);
}

void process_x(void) {
  // ...
  free(x);
}
```

```c
struct {
  char p[10];
  int is_admin;
} *y;

void write_y(char *data) {
  if (!y)
    y = calloc(sizeof(*y));

  if (strlen(data) >= 10)
    return;

  strcpy(y->p, data);
  process_cmd(y->p, y->is_admin);
}
```

1. `write_x("dummy 1 + padding")`
2. `process_x()`
3. `write_y("dummy 2")`
4. `write_x("fill buff PWND")`
5. `write_y("drop_db")`

- allocate x, set `x_len` to big enough
- free x, it is now dangling
- allocate y, memory is re-used: `x = y`
- write to y, bypassing length check
- enjoy your modified `y->is_admin`

Overview: memory allocation

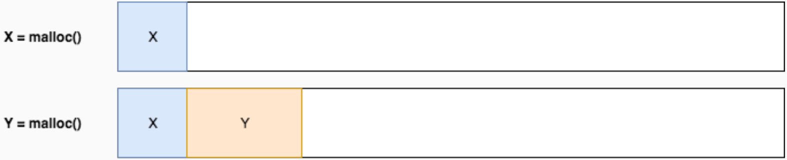
Overview: memory allocation

Normal memory allocation

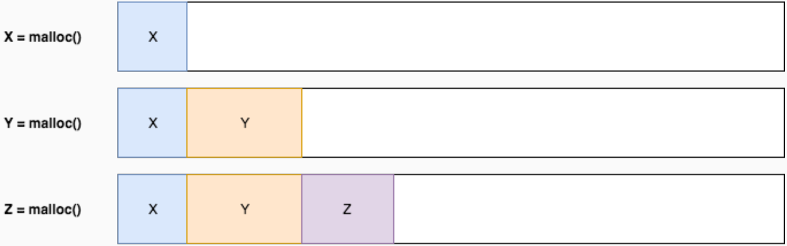
Normal memory allocation – step by step

X = malloc()

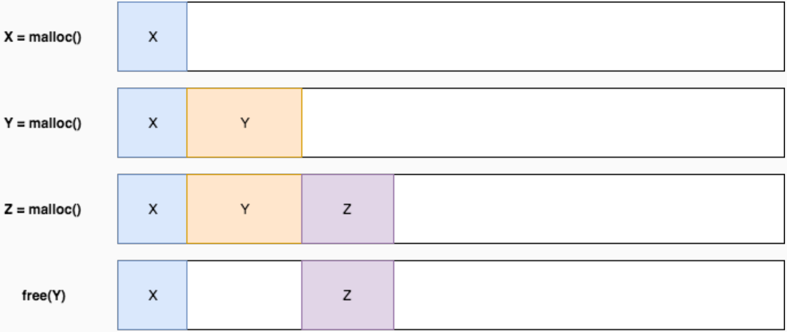
Normal memory allocation – step by step



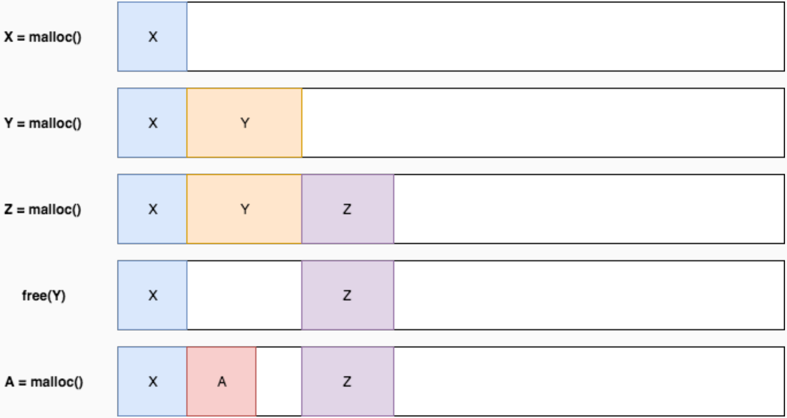
Normal memory allocation – step by step



Normal memory allocation – step by step



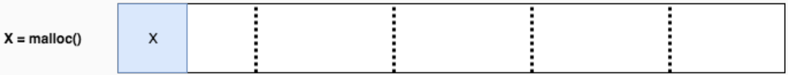
Normal memory allocation – step by step



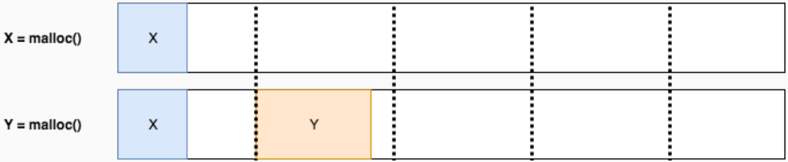
Overview: memory allocation

"Dangless" allocation – simplified

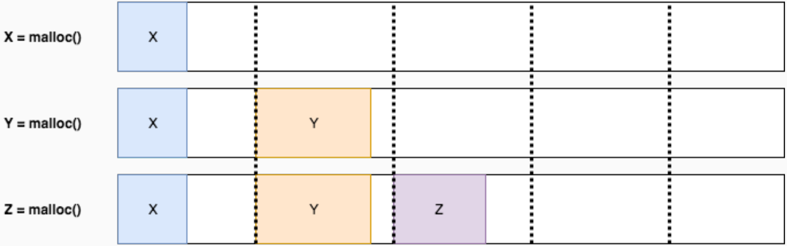
Dangless memory allocation – simplified



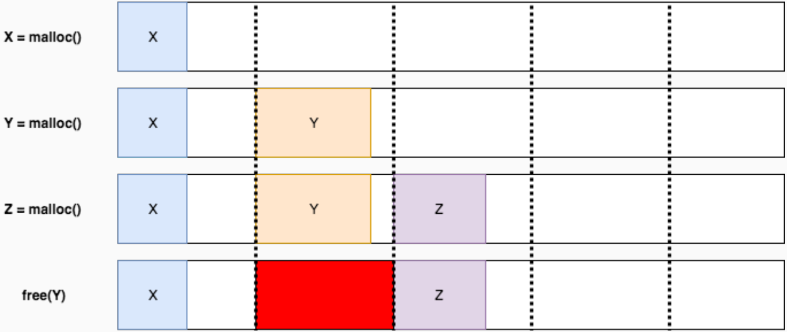
Dangless memory allocation – simplified



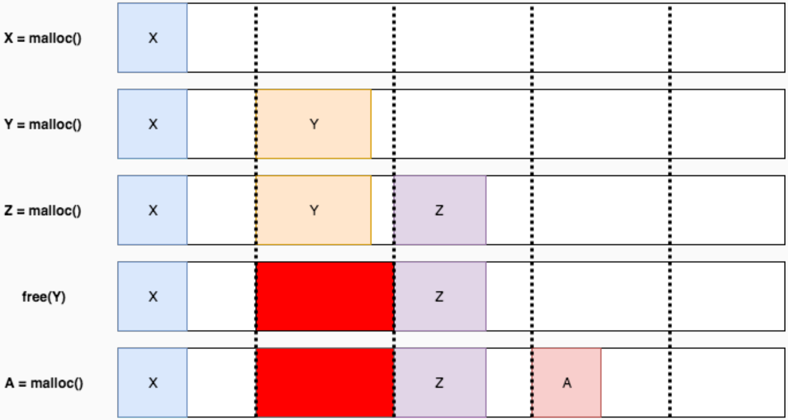
Dangless memory allocation – simplified



Dangless memory allocation – simplified



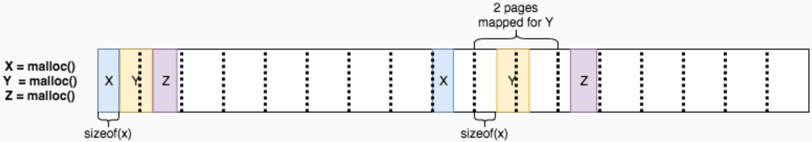
Dangless memory allocation – simplified



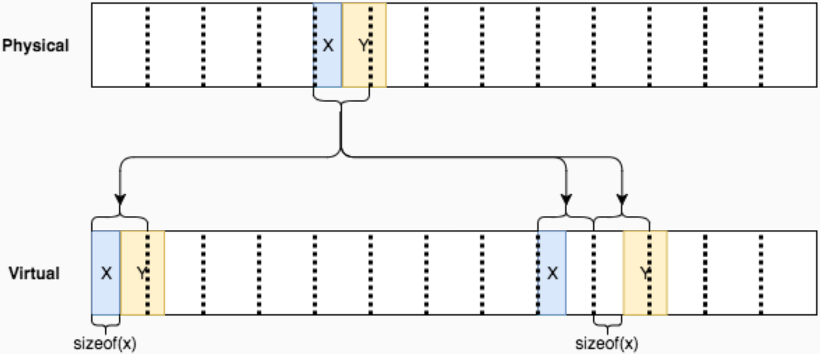
Overview: memory allocation

"Dangless" allocation – the full story

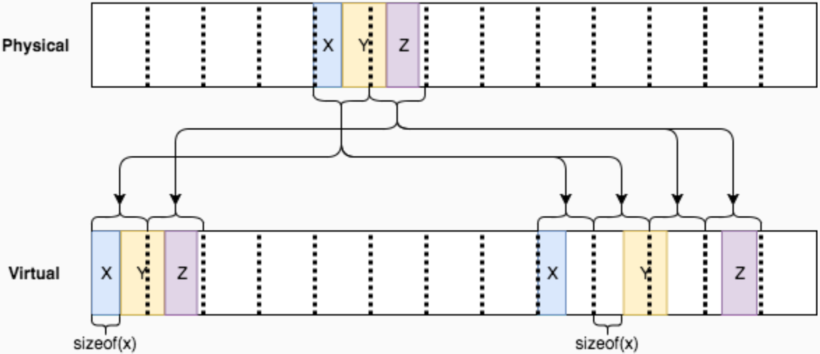
Dangless memory allocation – full memory view



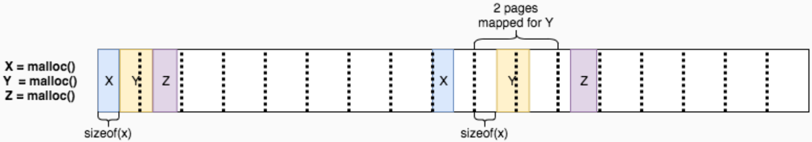
Dangless memory allocation – virtual remapping



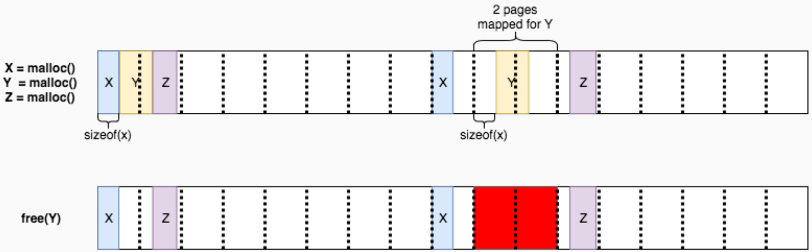
Dangless memory allocation – virtual remapping



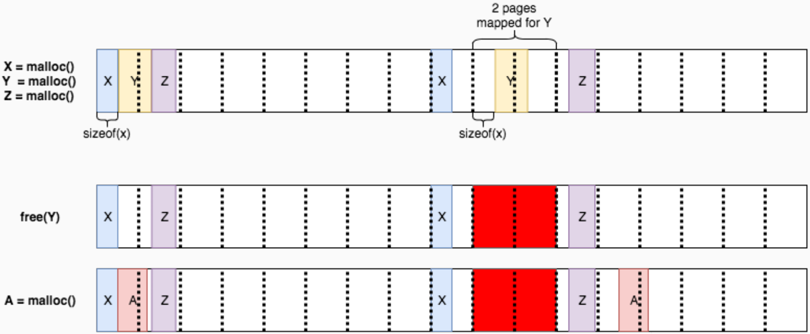
Dangless memory allocation – step by step



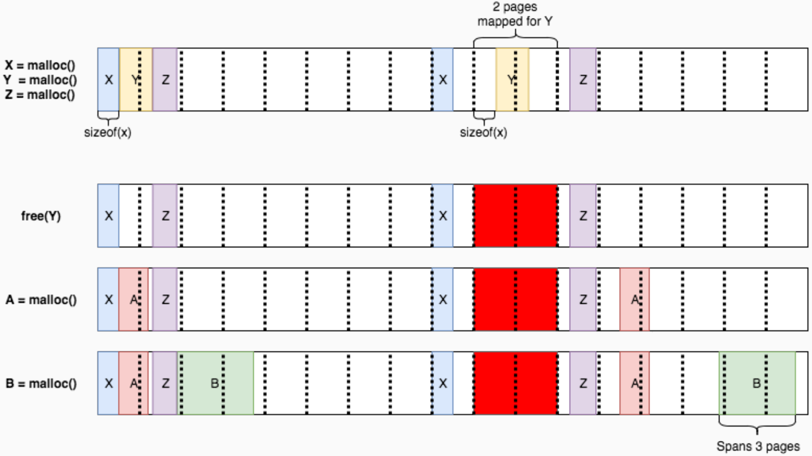
Dangless memory allocation – step by step



Dangless memory allocation – step by step



Dangless memory allocation – step by step



The difficulty with the Dangless scheme

Difficulties with the this scheme:

- Virtual aliasing happens through page-tables
- Ring 0 (kernel) privileges are required
- System calls (`mremap()`, `mprotect()`): significant overhead

Instead: use a light-weight virtual environment

The difficulty with the Dangless scheme

Difficulties with the this scheme:

- Virtual aliasing happens through page-tables
- Ring 0 (kernel) privileges are required
- System calls (`mremap()`, `mprotect()`): significant overhead

Instead: use a light-weight virtual environment

Dangless: implementation

Light-weight virtualization via Dune

- Dune: library and kernel module (based on KVM)
- Low-overhead virtualization
- Just call `dune_init_and_enter()`
- Virtual environment: ring 0 privileges
    - **Manipulate page tables**
    - Handle interrupts
    - Handle system calls
- `vmcall` to syscall the host kernel

Light-weight virtualization via Dune

- Dune: library and kernel module (based on KVM)
- Low-overhead virtualization
- Just call `dune_init_and_enter()`
- Virtual environment: ring 0 privileges
    - **Manipulate page tables**
    - Handle interrupts
    - Handle system calls
- `vmcall` to syscall the host kernel

Light-weight virtualization via Dune

- Dune: library and kernel module (based on KVM)
- Low-overhead virtualization
- Just call `dune_init_and_enter()`
- Virtual environment: ring 0 privileges
  - **Manipulate page tables**
  - Handle interrupts
  - Handle system calls
- `vmcall` to syscall the host kernel

The Dangless scheme: allocations

Allocations (`malloc()`, `calloc()`, etc):

- Forward to "system allocator" (`dlsym(RTLD_NEXT, "malloc")`)
- If recursive call: passthrough
- If not in ring 0: passthrough
- Virtual page allocator: allocate pages
- Create the virtual memory mapping
- Return the remapped address

The Dangless scheme: deallocations

Deallocations (`free()`, sometimes `realloc()`):

- If recursive call: passthrough
- If not in ring 0: passthrough
- If not remapped: passthrough
- Resolve canonical address (page-table walk)
- Invalidate virtual alias pages (clear the "present" bit)
- Call "system allocator" (`dlsym(RTLD_NEXT, "free")`)

Dangless: implementation

`vmcall` with remapped pointers

Caveat: `vmcall` with remapped pointers

- Virtualization: host (Linux) memory `!=` guest (libdune) memory
- Manipulating page tables in guest: no effect on host
- Dangless' virtual aliases: unknown to host

Caveat: `vmcall` with remapped pointers

- Virtualization: host (Linux) memory `!=` guest (libdune) memory
- Manipulating page tables in guest: no effect on host
- Dangless' virtual aliases: unknown to host

Caveat: `vmcall` with remapped pointers

- Virtualization: host (Linux) memory != guest (libdune) memory
- Manipulating page tables in guest: no effect on host
- Dangless' virtual aliases: unknown to host

```
char *filename = malloc(32);
// ...

int fd = open(filename, O_CREAT | O_WRONLY);
// ...
```

Fixing up `vmcall` arguments

- *linux-syscallmd*: Python script, parse Linux *syscall.h*
- Patched Dune: `vmcall` pre- and post-hooks
- Pre-hook: replace virtual alias pointer arguments with canonical

Fixing up `vmcall` arguments

- *linux-syscallmd*: Python script, parse Linux *syscall.h*
- Patched Dune: `vmcall` pre- and post-hooks
- Pre-hook: replace virtual alias pointer arguments with canonical

```
void dangless_vmcall_prehook(
    uint64_t *syscallno,
    uint64_t args[],
    uint64_t *retaddr
);

void dangless_vmcall_posthook(uint64_t result);
```

Fixing up `vmcall` arguments

- *linux-syscallmd*: Python script, parse Linux *syscall.h*
- Patched Dune: `vmcall` pre- and post-hooks
- Pre-hook: replace virtual alias pointer arguments with canonical

```
void dangless_vmcall_prehook(
    uint64_t *syscallno,
    uint64_t args[],
    uint64_t *retaddr
);

void dangless_vmcall_posthook(uint64_t result);
```

Fixing up nested pointers

```c
ssize_t readv (int fd, const struct iovec *v, int n);
ssize_t writev(int fd, const struct iovec *v, int n);

struct iovec {
    void  *iov_base; /* Starting address */
    size_t iov_len;  /* Number of bytes */
};
```

Fixing up nested pointers

Other examples:

- `execve()`: arrays of pointers (`argv`, `environ`)
- `recvmsg()`: complex `struct msghdr` (incl. `struct iovec`)

Performance evaluation

SPEC 2006

- SPEC2006: benchmarking suite
- Both CPU- and memory-intensive benchmarks
- Performance and memory overhead both relevant
- Baseline: vanilia & Dune only

Benchmarks missing

- Previous OOM-killed: `400.perlbench`, `403.gcc`, `433.milc`
- Previous 4 GB exceeded: `447.dealII`, `471.omnetpp`, `482.sphinx3`, `483.xalancbmk`
- Segfault: `471.omnetpp`
- Included: 12 / 19 benchmarks

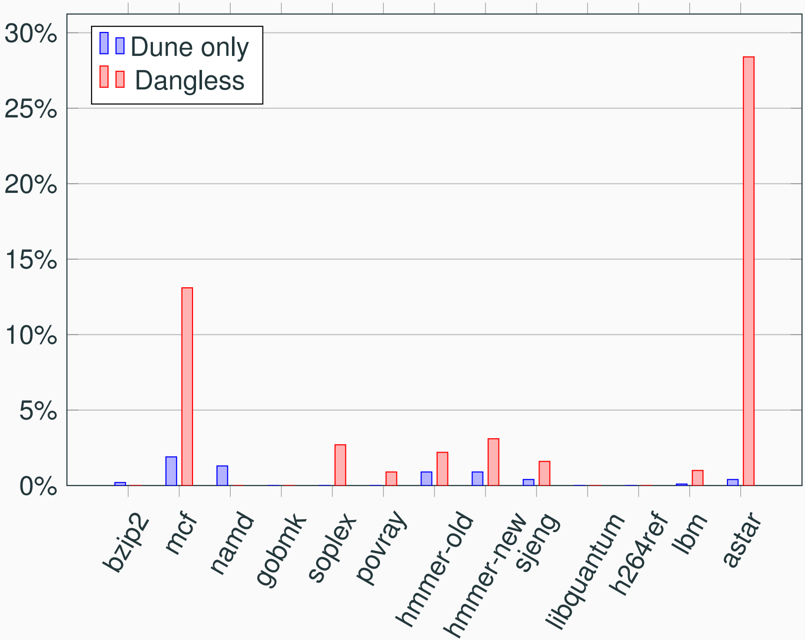
Benchmarks missing

- Previous OOM-killed: `400.perlbench`, `403.gcc`, `433.milc`
- Previous 4 GB exceeded: `447.dealII`, `471.omnetpp`, `482.sphinx3`, `483.xalancbmk`
- Segfault: `471.omnetpp`
- Included: 12 / 19 benchmarks

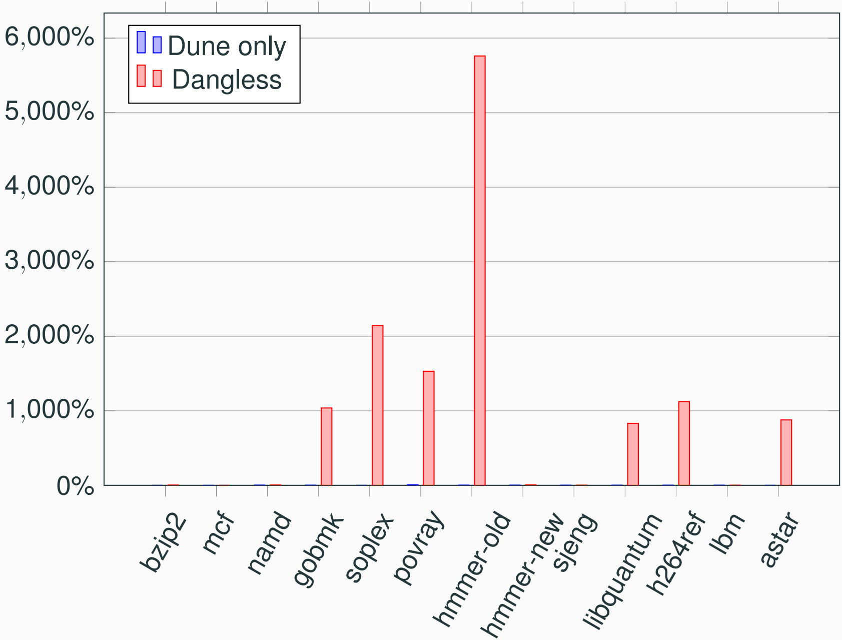
```
    }
} else /*if (result < 0) {
    LOG("failed to determine whether %p was remapped: assume not (
    result %d)\n", p, result);
} else {
    LOG("vremap_resolve returned %d, assume no remapping\n", result);
}*/

sysfree(original_ptr);
```

Figure 1: . . . . .

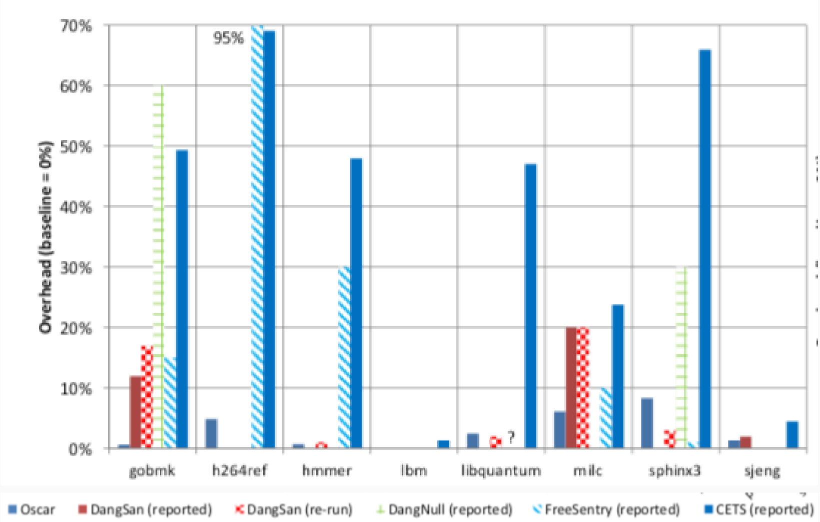
SPEC2006 – performance overhead



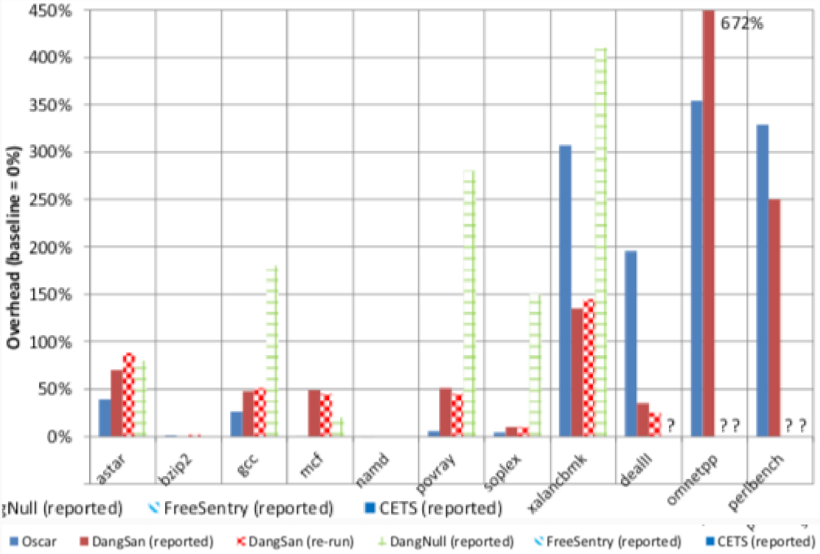
SPEC2006 – memory overhead



Overhead of previous technologies



Overhead of previous technologies



Performance – summary

- Discovery: not `free()`-ing makes memory usage go up a lot!
- Final results pending
  - Teaser: 456.hmmer: 5760.4% => 3.9%
- Previously geometric means (on subset):
  - Performance overhead: 3.5%
  - Memory overhead: 406.7%
- Performance compares favourably to previous techniques (Oscar)

Performance – summary

- Discovery: not `free()`-ing makes memory usage go up a lot!
- Final results pending
  - Teaser: 456.hmmer: 5760.4% => 3.9%
- Previously geometric means (on subset):
  - Performance overhead: 3.5%
  - Memory overhead: 406.7%
- Performance compares favourably to previous techniques (Oscar)

Conclusion

Summary

- Use-after-free (dangling pointer) bugs are a security risk
- Detecting and defending against them has traditionally been impractical and/or inefficient
- The virtual aliasing scheme was originally proposed more than a decade ago (2006), and re-proposed again recently (2017)
- Light-weight virtualization provides another way to manage performance overhead

Limitations & possible future work

- Limitations of Dune:
  - Not thread-safe
  - Signal processing
  - Tied to kernel versions
- `clone()` and `fork()` not supported
- Virtual memory could eventually be reused: GC

Thank you for your attention!