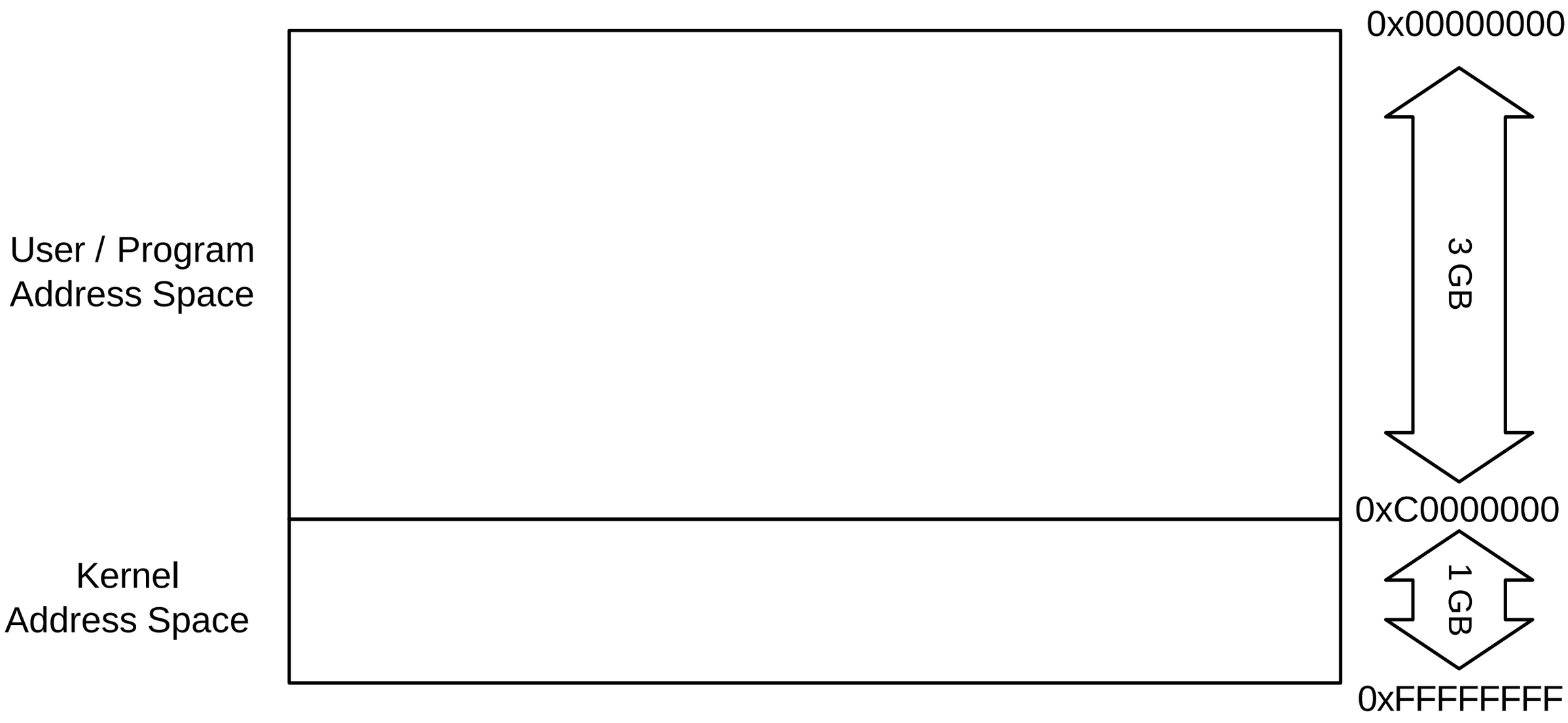
Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels

Meng Xu, Chenxiong Qian, Kangjie Lu[+], Michael Backes*, Taesoo Kim

Georgia Tech | *University of Minnesota*[+] | *CISPA, Germany**

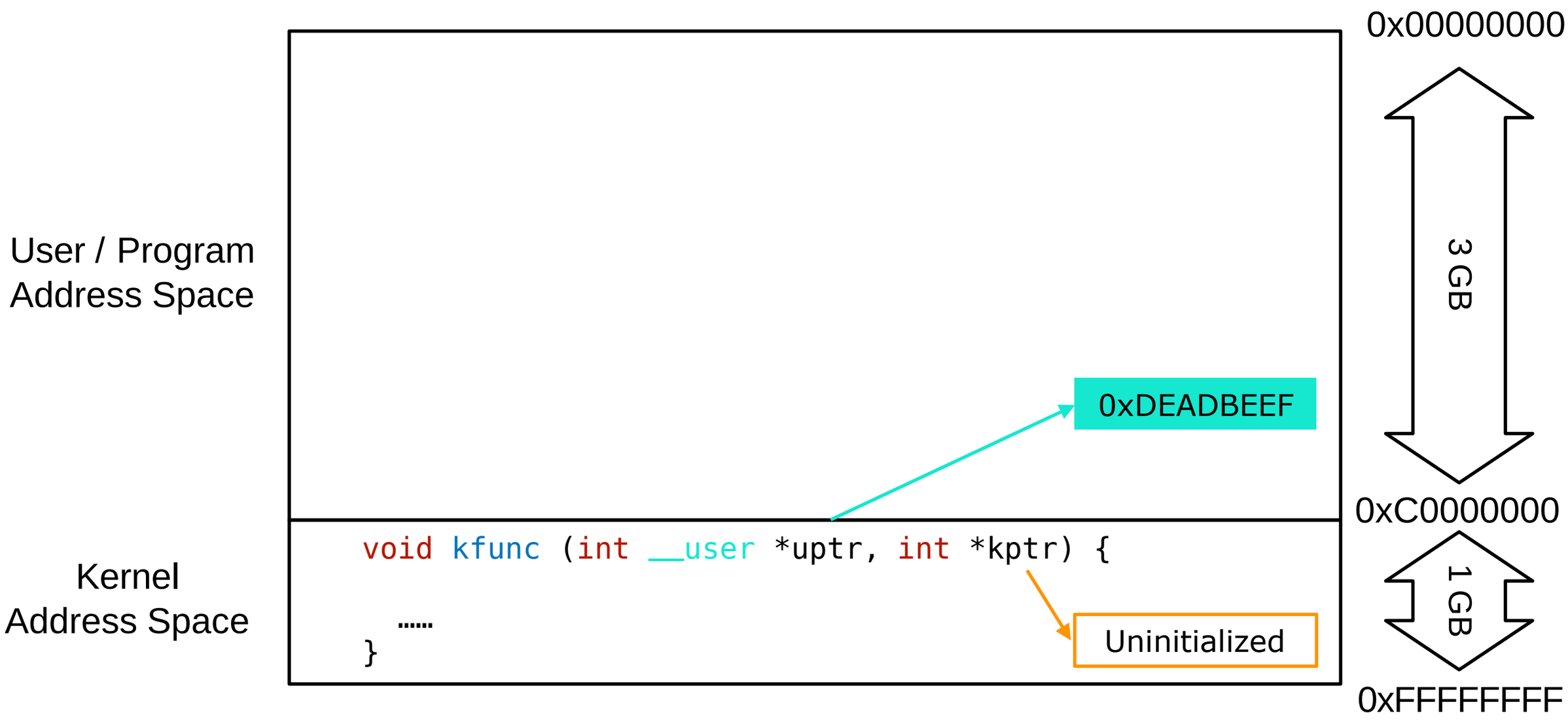
What is Double-Fetch?

Address Space Separation



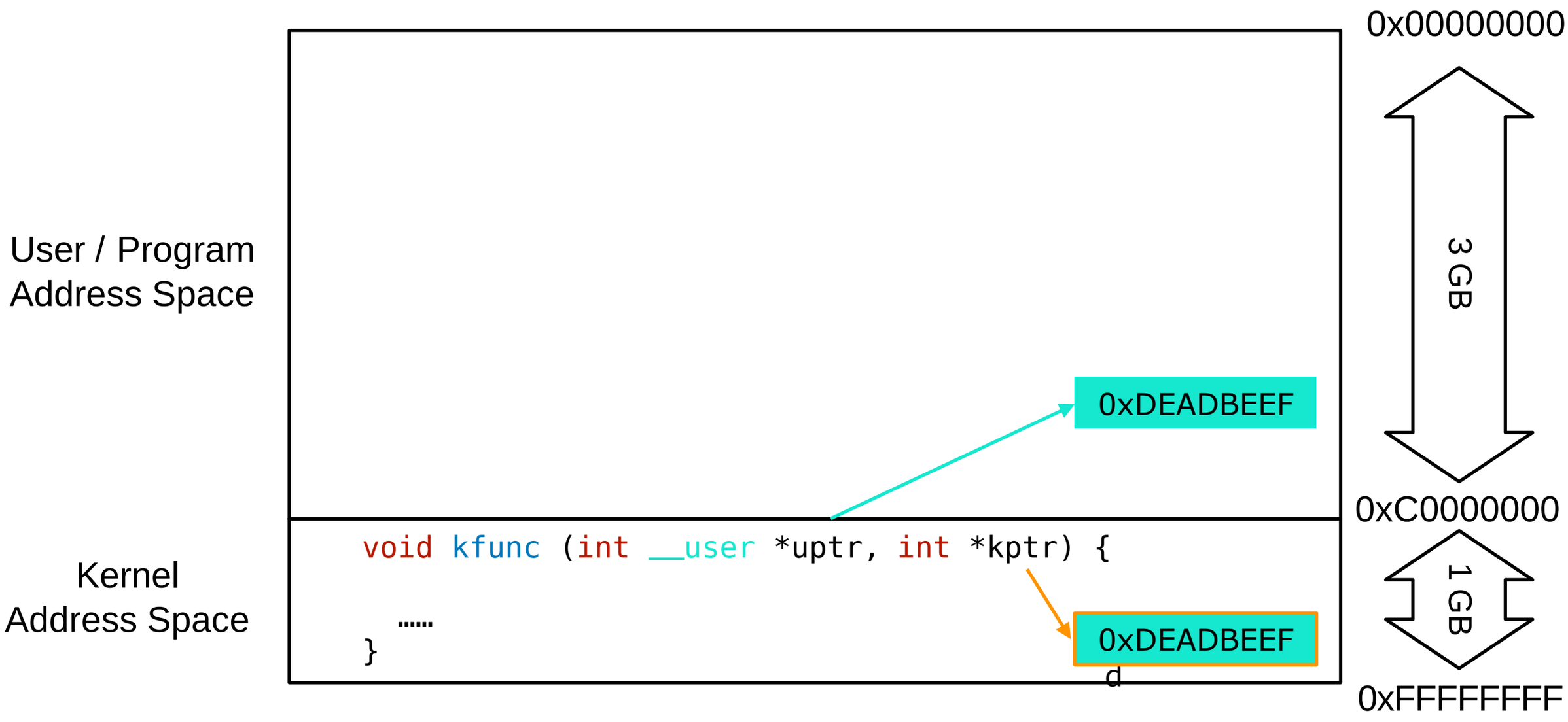
A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

No Dereference on Userspace Pointers



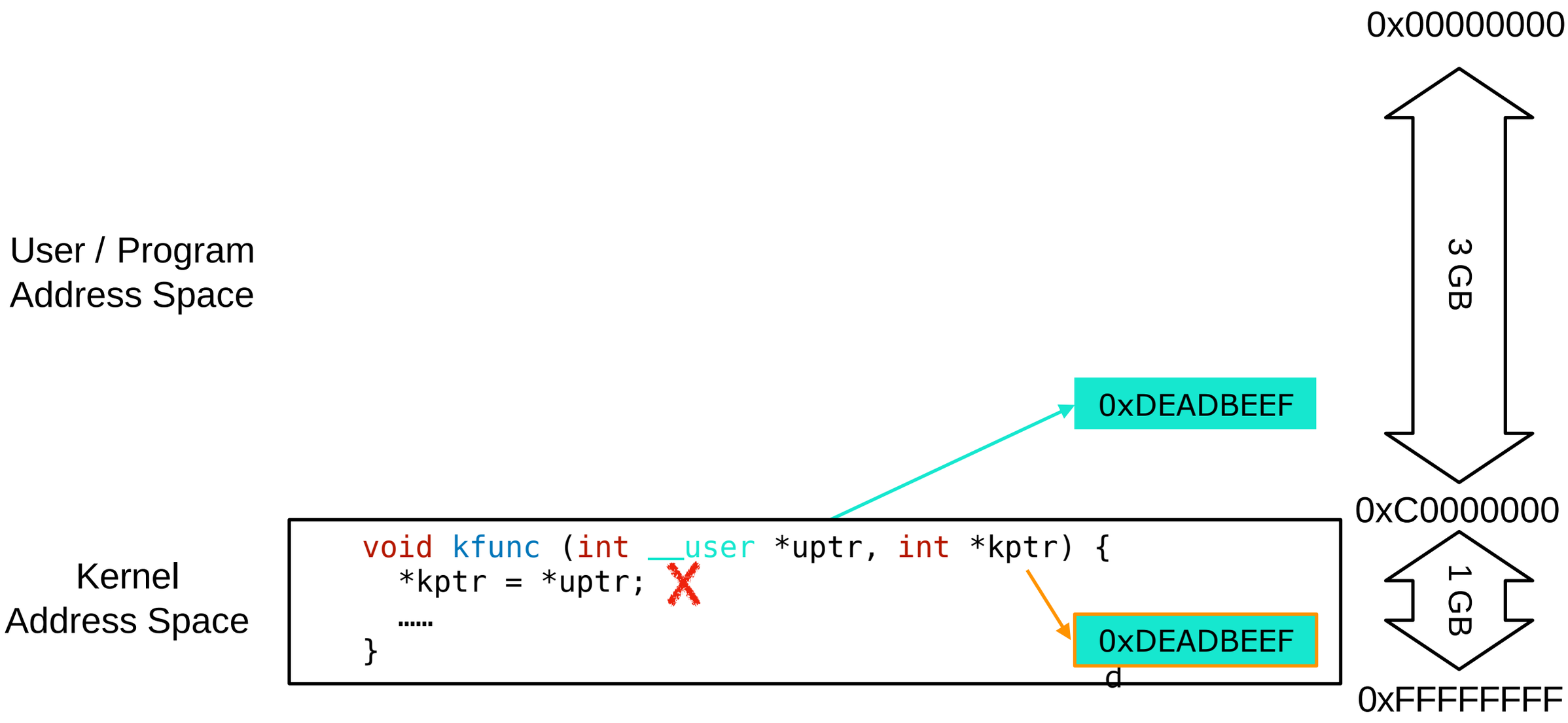
A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

No Dereference on Userspace Pointers



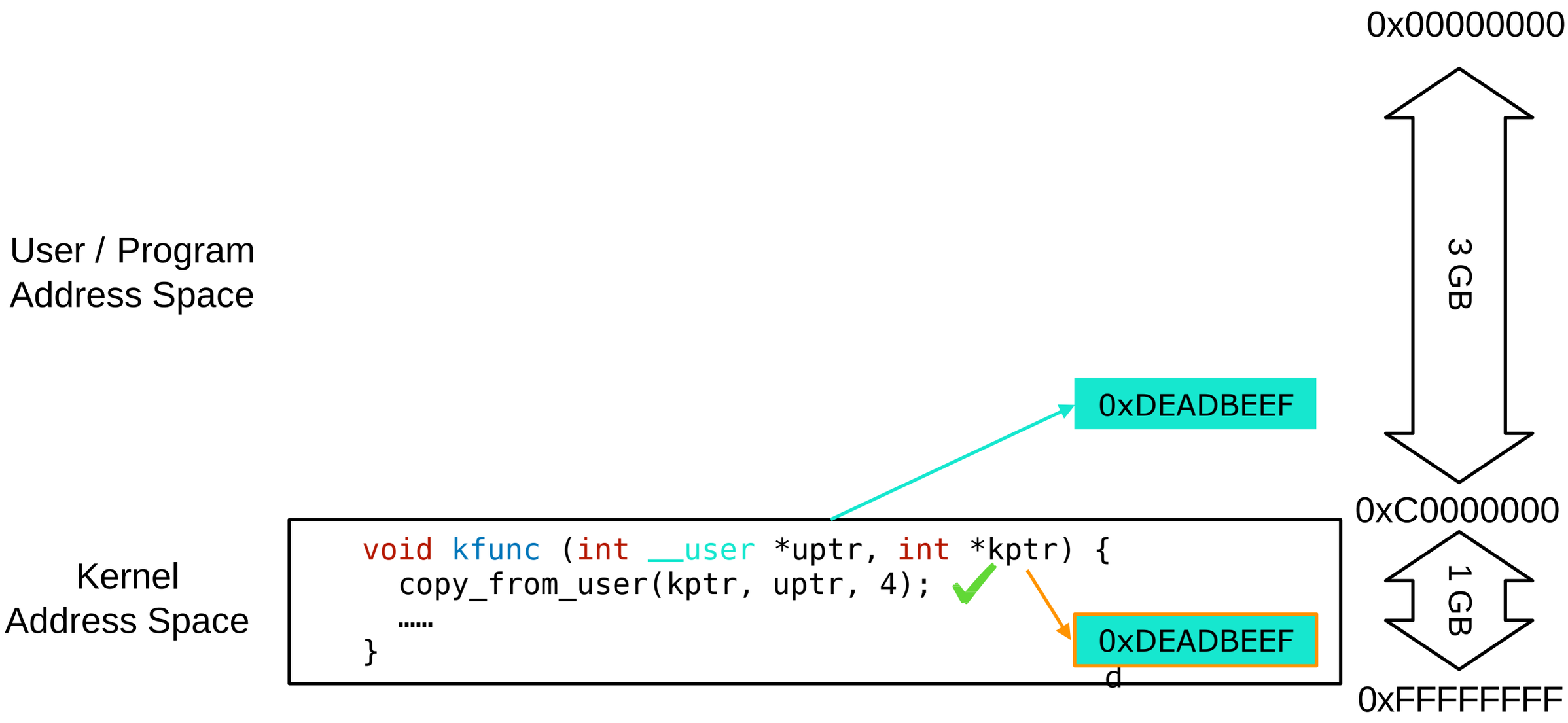
A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

No Dereference on Userspace Pointers



A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

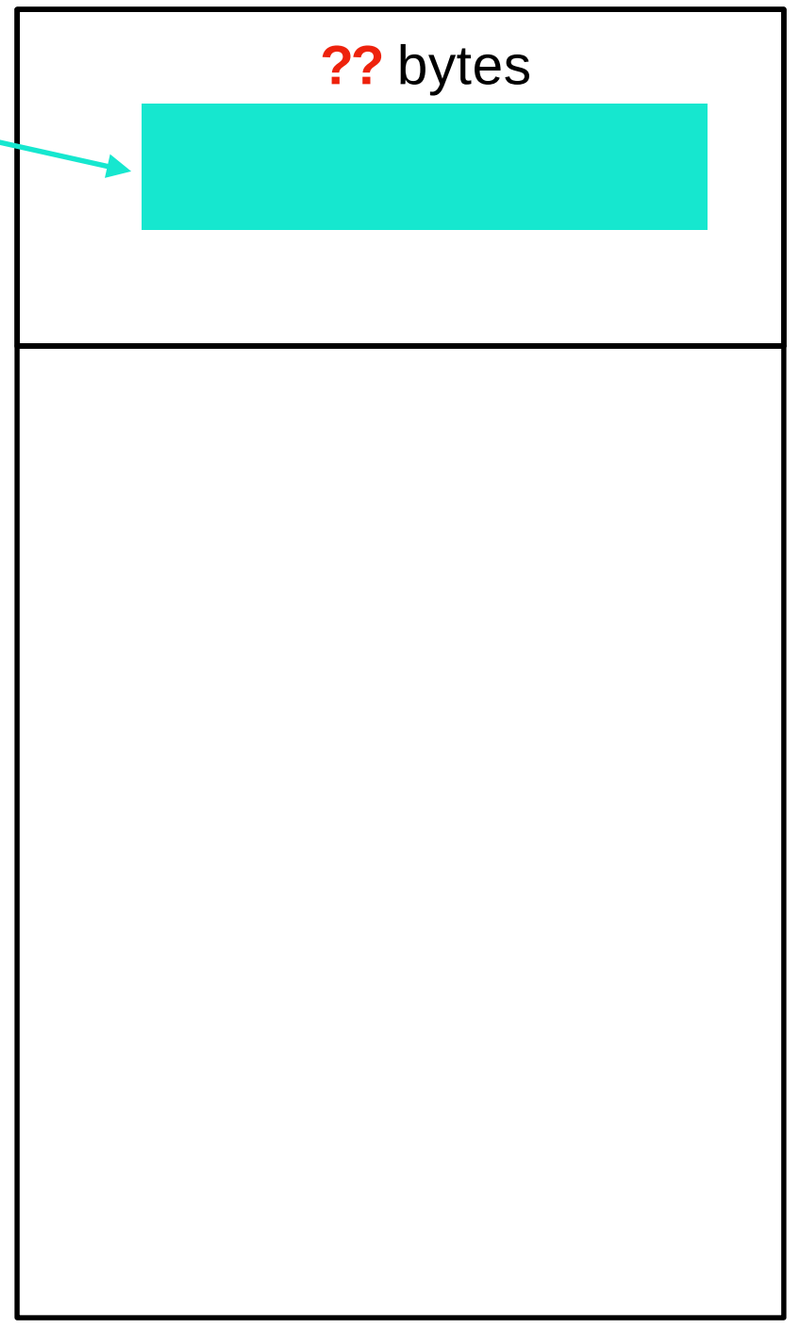
No Dereference on Userspace Pointers



A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

Why Double-Fetch?

```
1 static int perf_copy_attr_simplified
2    (struct perf_event_attr _user *uattr,
3     struct perf_event_attr *attr) {
```

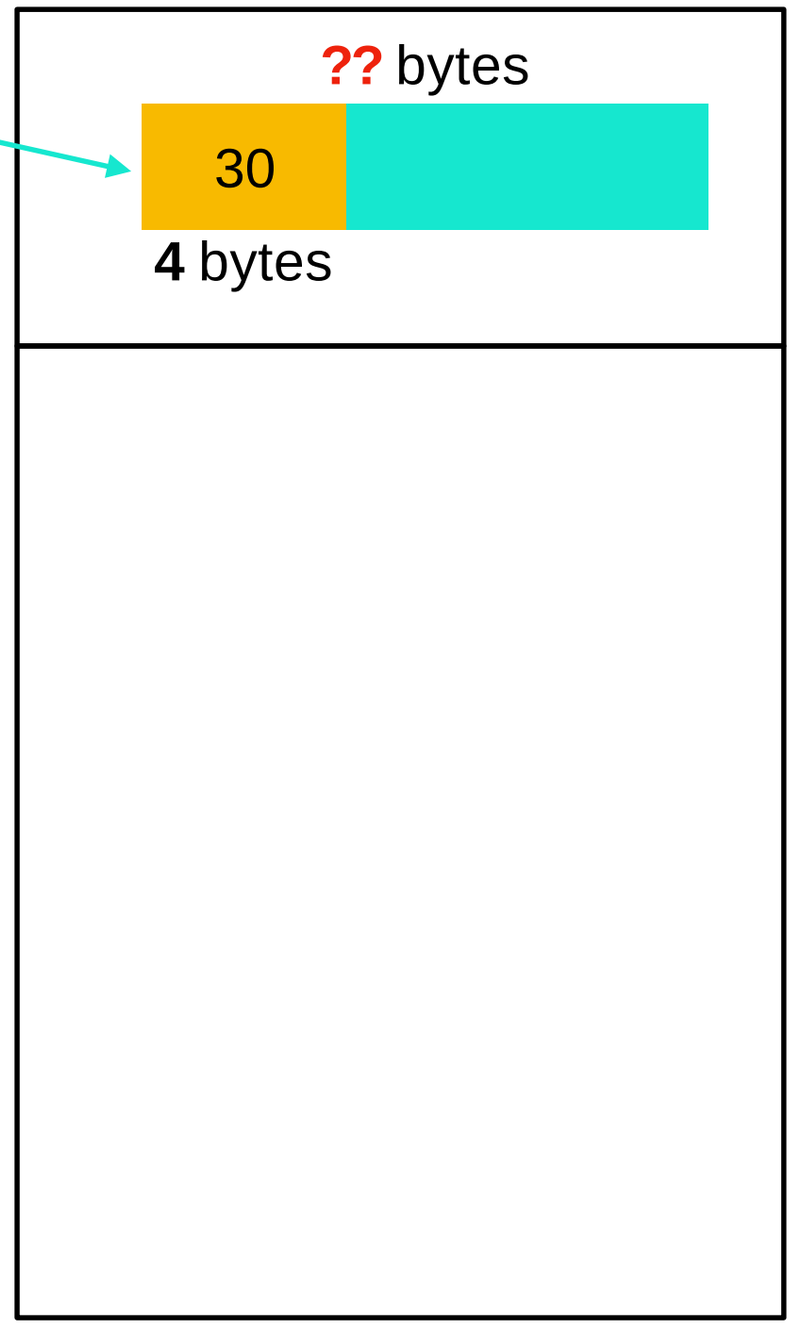



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Why Double-Fetch?

```
1 static int perf_copy_attr_simplified
2    (struct perf_event_attr _user *uattr,
3     struct perf_event_attr *attr) {
4
5    u32 size;
```

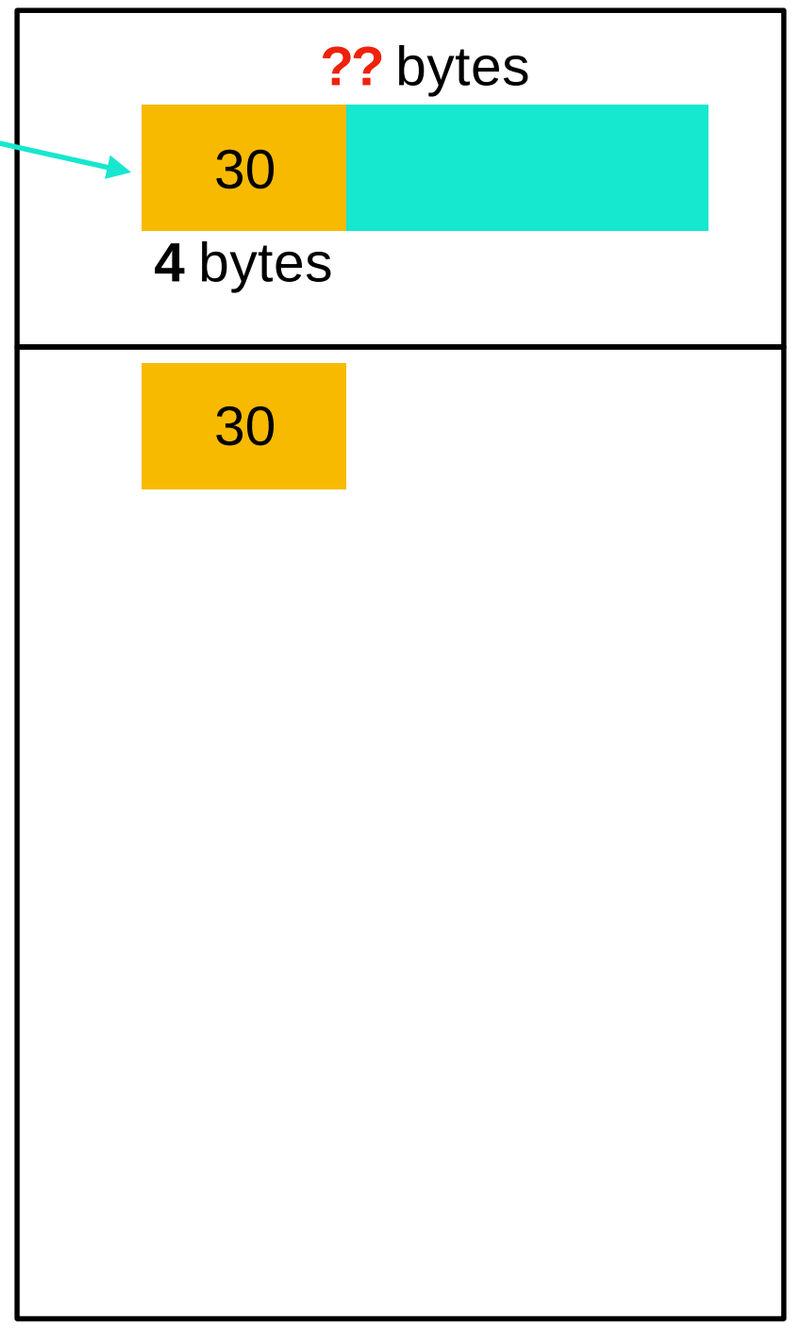



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Why Double-Fetch?

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
```

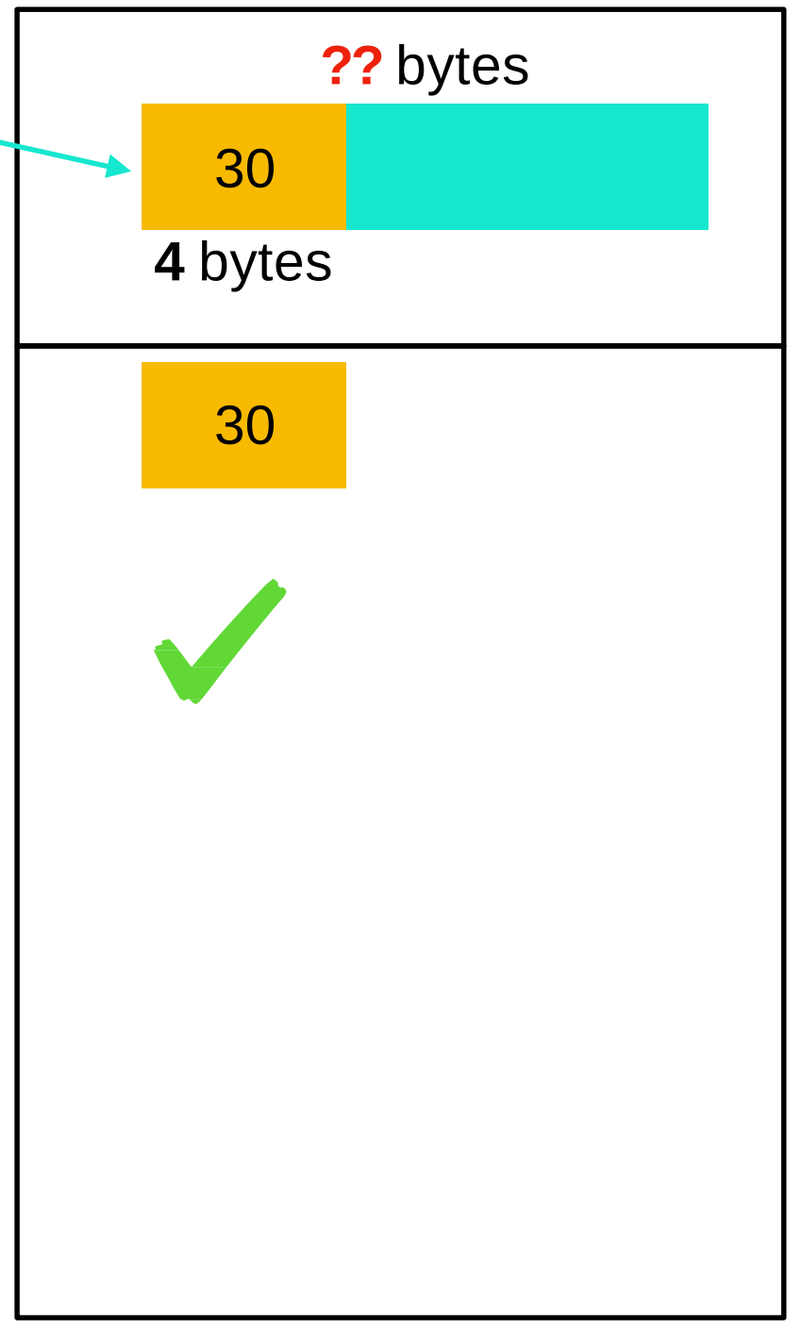



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Why Double-Fetch?

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
```

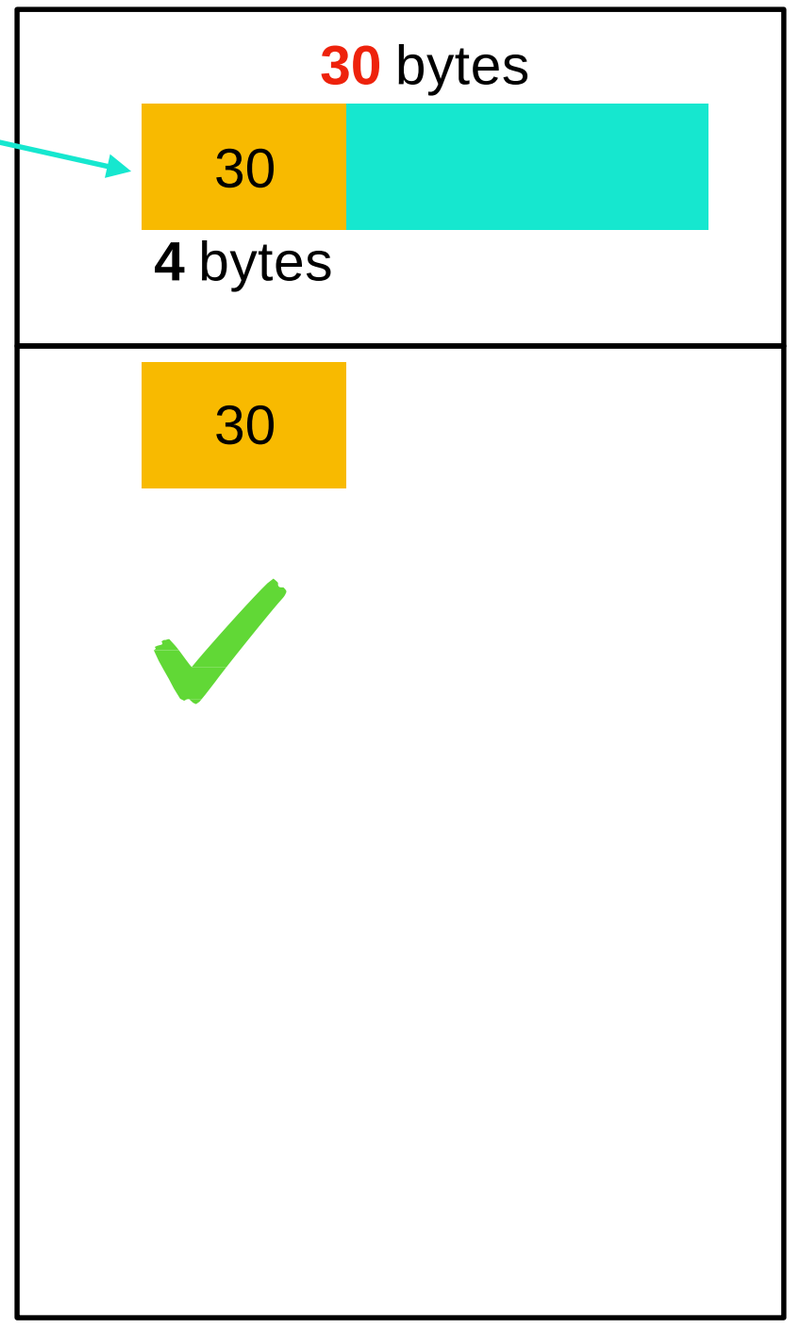



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Why Double-Fetch?

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
```

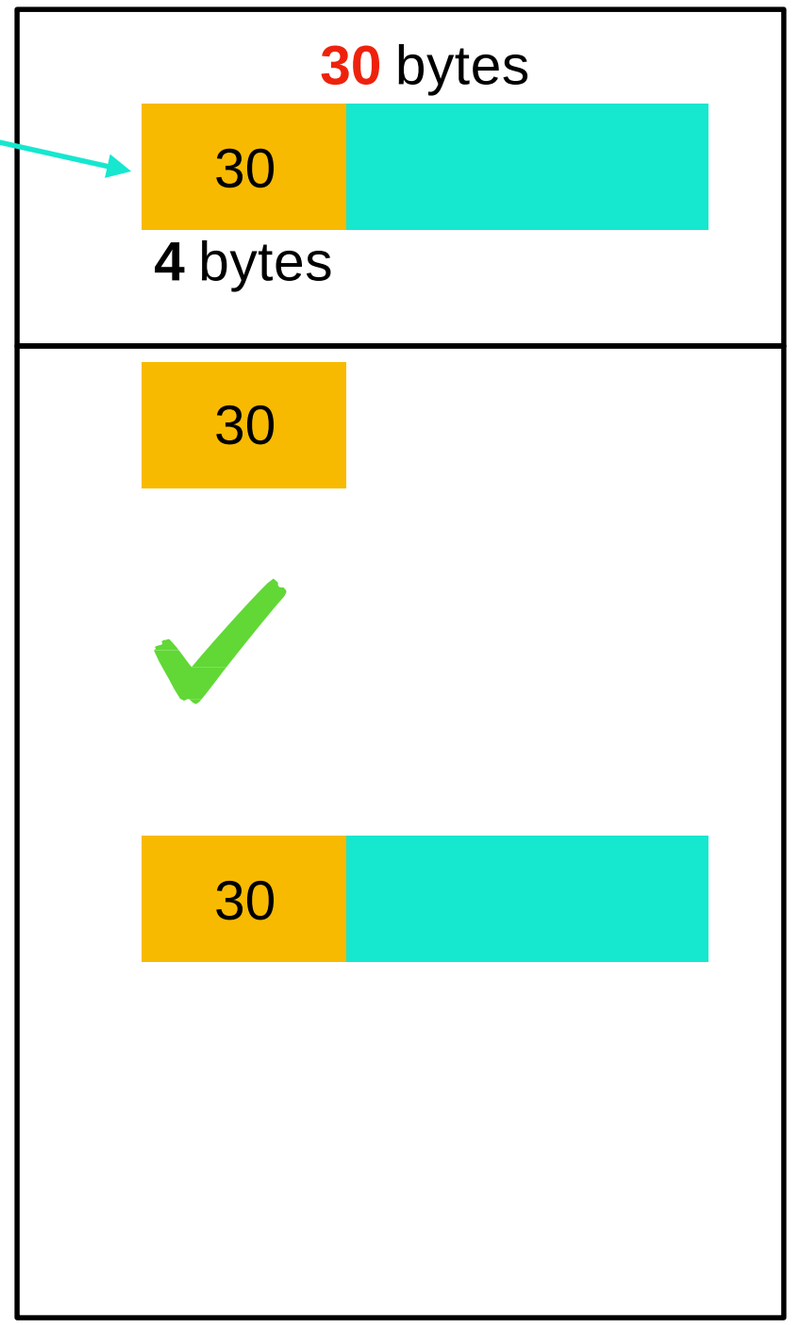



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Why Double-Fetch?

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr, size))
18      return -EFAULT;
```

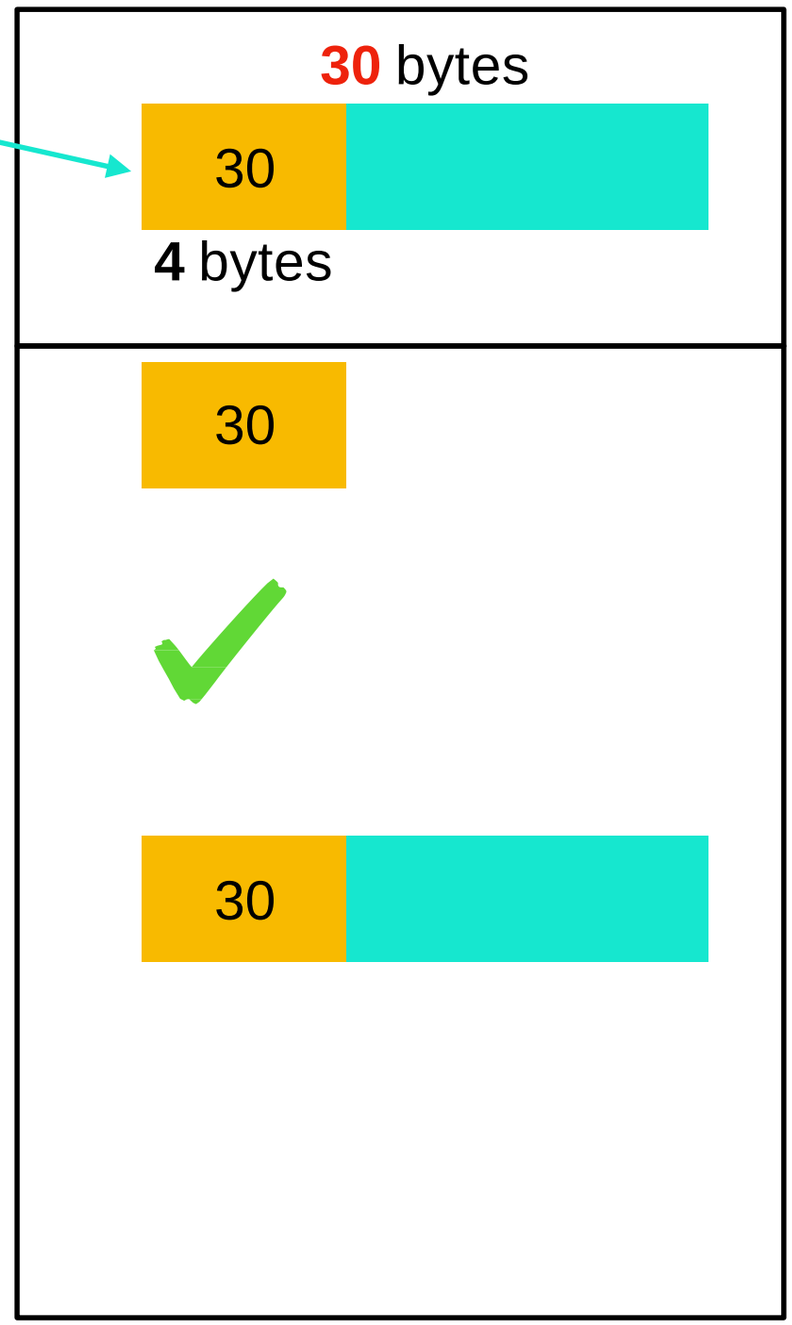



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Why Double-Fetch?

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr, size))
18      return -EFAULT;
19
20    .......
21 }
```



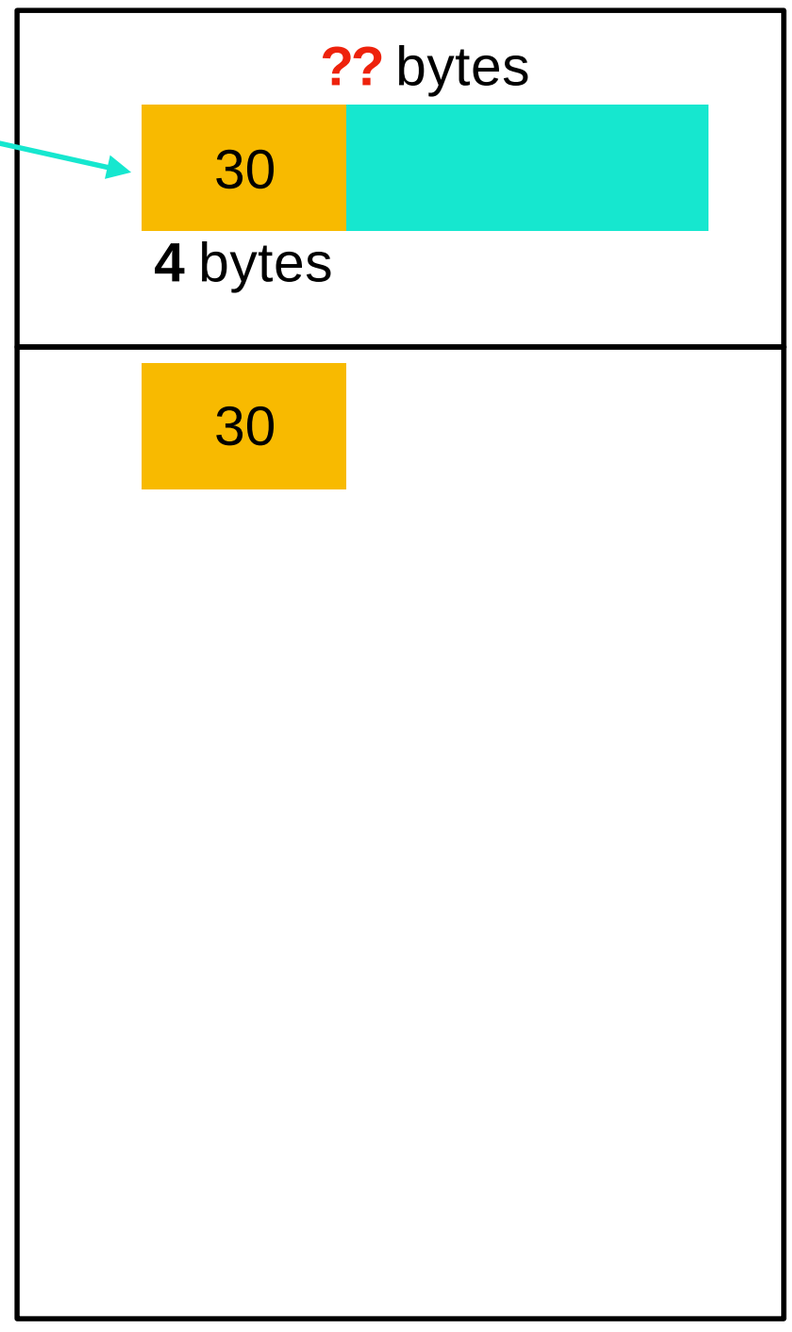

```
24 memcpy(buf,  attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

What Goes Wrong in This Process?

Up-until First-Fetch

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
```

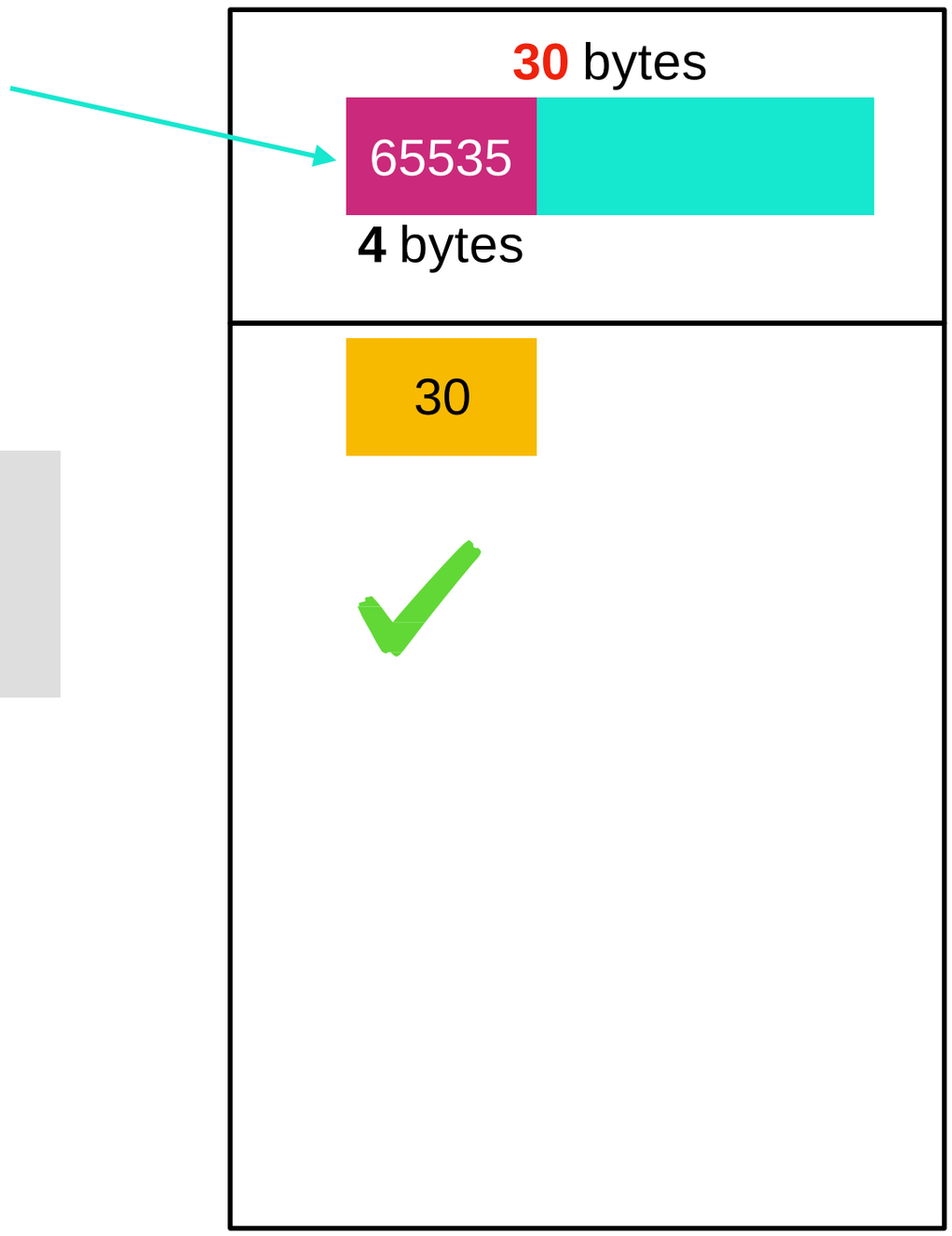



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Wrong Assumption: Atomicity in Syscall

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size <
          PERF_ATTR_SIZE_VER0)
```

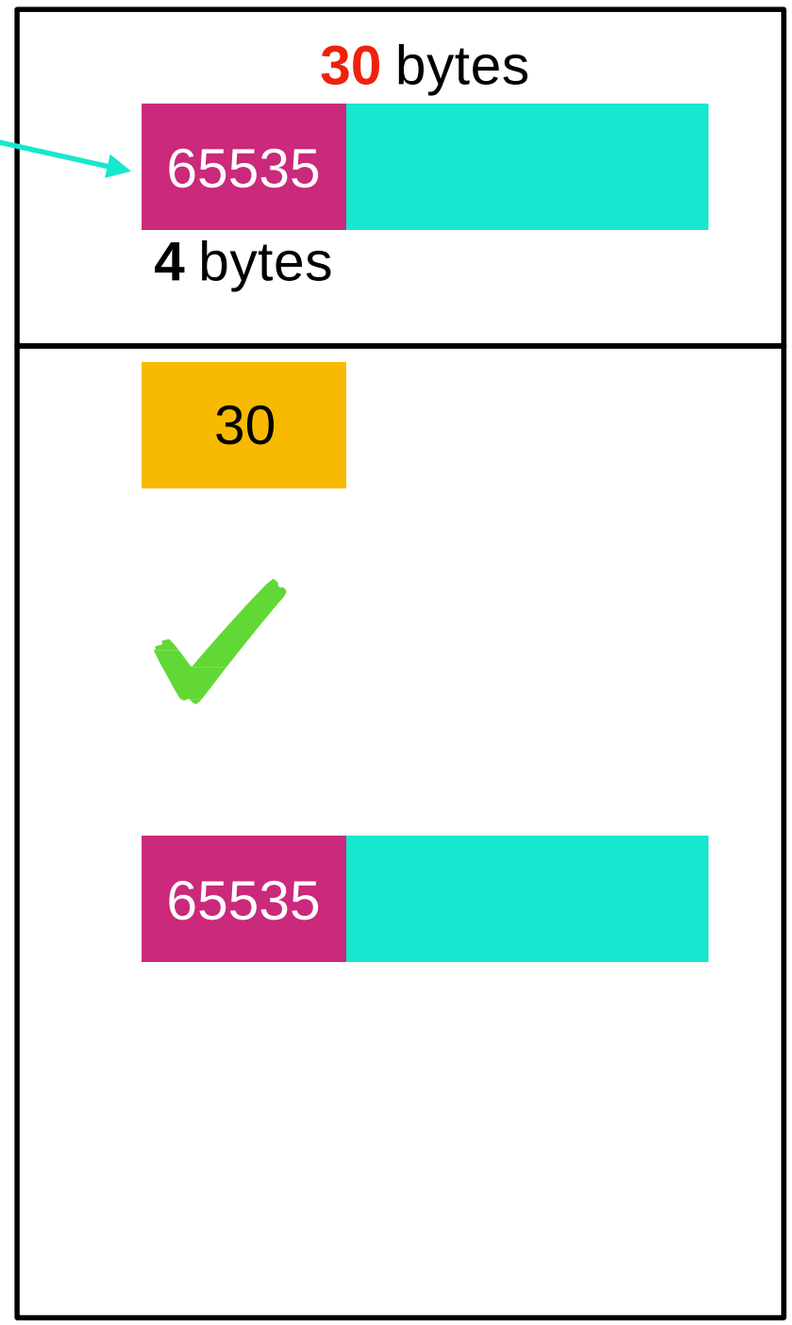



```
24 memcpy(buf, attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Wrong Assumption: Atomicity in Syscall

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr, size))
18      return -EFAULT;
19
20    ......
21 }
```

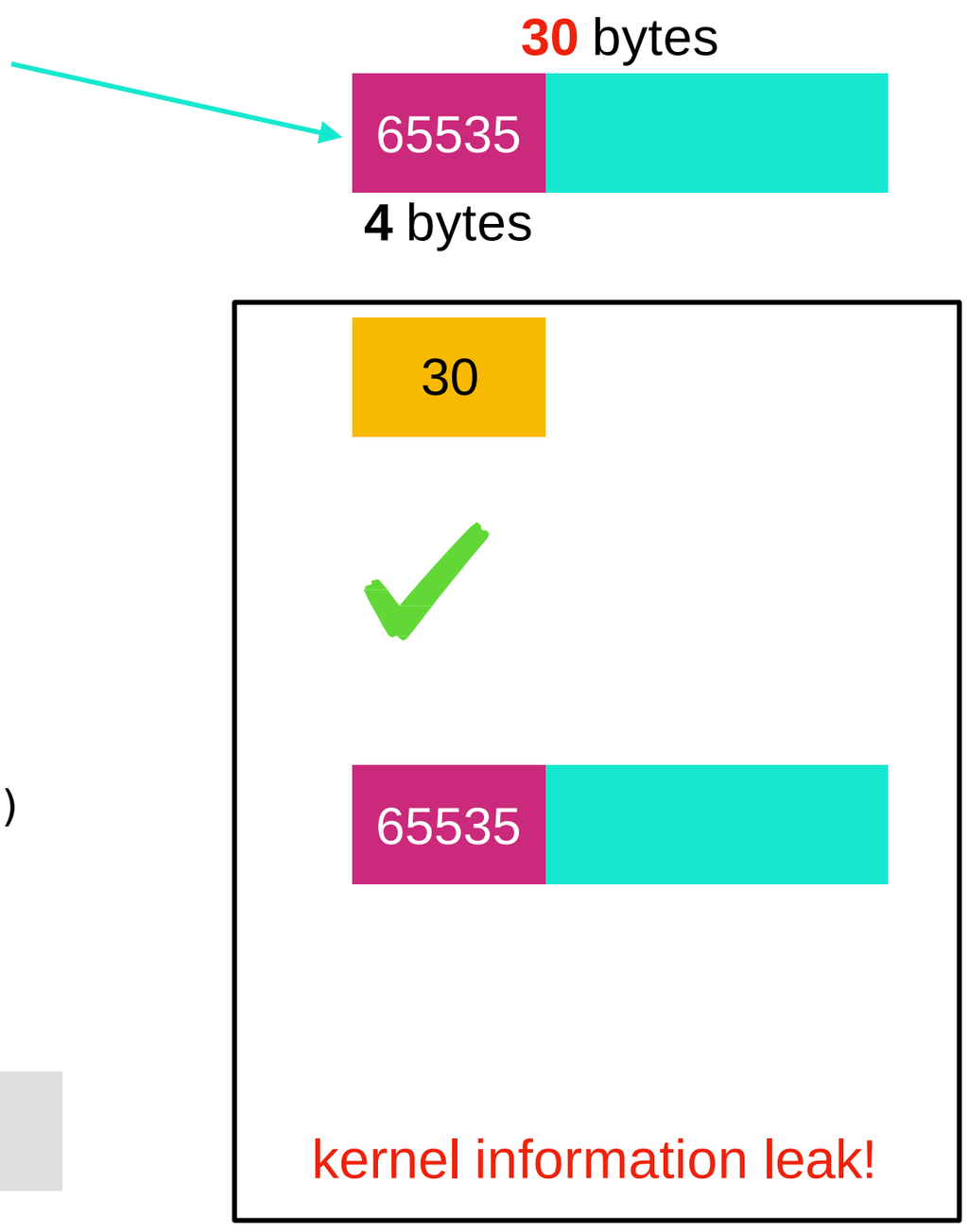



```
24 memcpy(buf,  attr, attr->size);
```

Adapted from `perf_copy_attr` in file `kernel/events/core.c`

When The Exploit Happens

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr, size))
18      return -EFAULT;
19
20    ......
21 }
22
23 //  BUG: when attr->size is used
       later
24 copy_to_user(ubuf, attr, attr-
>size);
```



Adapted from `perf_copy_attr` in file `kernel/events/core.c`

Double-Fetch Bugs: Towards A Formal Definition

Fetch: A pair *(A, S)*, where

A - the starting address of the fetch,

S - the size of memory copied into kernel.

Overlapped-fetch: Two fetches, (A_0, S_0) and (A_1, S_1), where

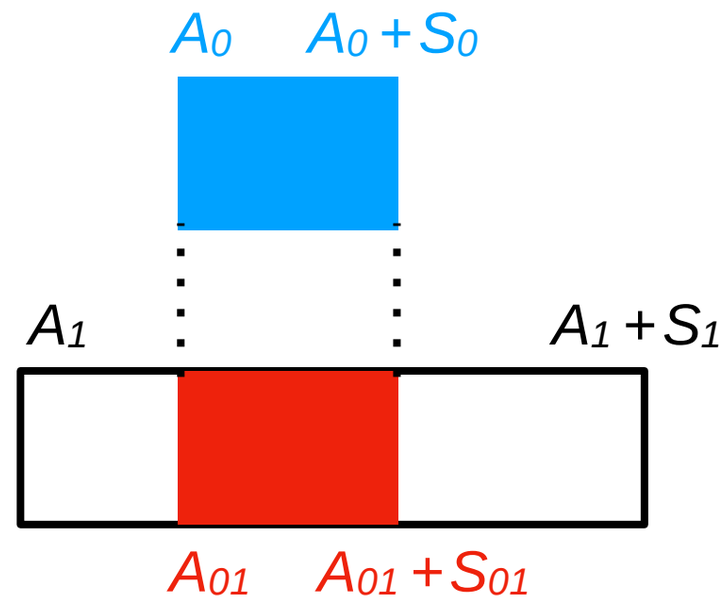
$$A_0 \leq A_1 < A_0 + S_0 \quad || \quad A_1 \leq A_0 < A_1 + S_1$$

- The overlapped memory region is marked as (A_{01}, S_{01}).
- The copied value during 1st fetch is $(A_{01}, S_{01}, 0)$
- The copied value during 2nd fetch is $(A_{01}, S_{01}, 1)$.

Overlapped-Fetch Case 1

```
get_user(attr, &uptr->attr)
```

```
copy_from_user(kptr, uptr, size)
```



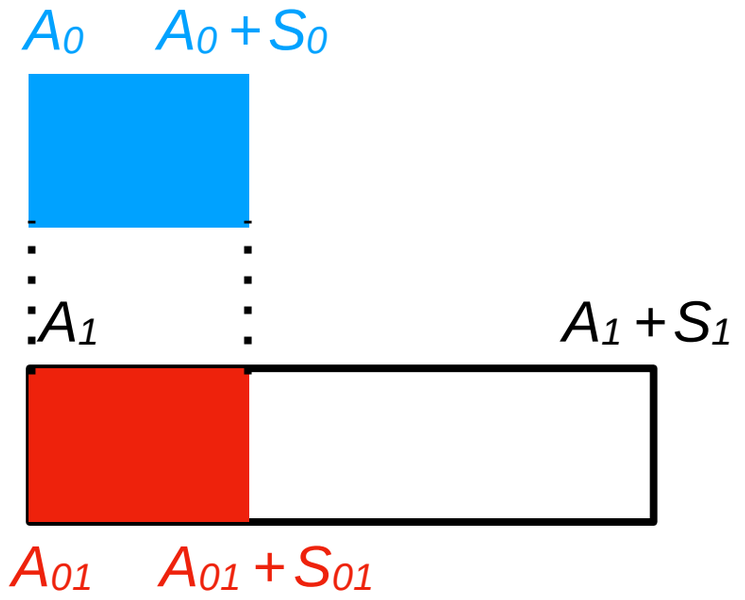
$(A_{01}, S_{01}, 0)$ `attr`

$(A_{01}, S_{01}, 1)$ `kptr->attr`

Overlapped-Fetch Case 2

```
copy_from_user(
  khdr, uptr, sizeof(struct hdr)
)
```

```
copy_from_user(
  kmsg, uptr, khdr->size
)
```



$(A_{01}, S_{01}, 0)$ `khdr->size, khdr->type, …`

$(A_{01}, S_{01}, 1)$ `kmsg->size, kmsg->type, …`

Double-Fetch Bugs: Towards A Formal Definition

Control dependence: A variable $V \in (A_{01}, S_{01})$ and V must satisfy a set of constraints before the second fetch can happen.

Double-Fetch Bugs: Towards A Formal Definition

Control dependence: A variable $V \in (A_{01}, S_{01})$ and V must satisfy a set of constraints before the second fetch can happen.

```
1  void tls_setsockopt_simplified(char __user *arg) {
2    struct tls_crypto_info header, *full = /* allocated before */;
3
4    // first fetch
5    if (copy_from_user(&header, arg, sizeof(struct tls_crypto_info)))
6      return -EFAULT;
7
8    // protocol check
9    if (header.version != TLS_1_2_VERSION)
10     return -ENOTSUPP;
11
12   // second fetch
13   if (copy_from_user(full, arg,
14         sizeof(struct tls12_crypto_info_aes_gcm_128)))
15     return -EFAULT;
16
17   // BUG: full->version might not be TLS_1_2_VERSION
18   do_sth_with(full);
19 }
```

Overlapped variable *V:*
`header.version`

The constraint it must satisfy:
`header.version == TLS_1_2_VERSION`

Expect:
`full->version == TLS_1_2_VERSION`

Double-Fetch Bugs: Towards A Formal Definition

Data dependence: A variable $V \in (A_{01}, S_{01})$ and V is consumed before or on the second fetch (e.g., involved in calculation, passed to function calls, etc).

Double-Fetch Bugs: Towards A Formal Definition

Data dependence: A variable $V \in (A_{01}, S_{01})$ and V is consumed before or on the second fetch.

```
1  void mptctl_simplified(unsigned long arg) {
2    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
3    MPT_ADAPTER *iocp = NULL;
4
5    // first fetch
6    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
7      return -EFAULT;
8
9    // dependency lookup
10   if (mpt_verify_adapter(khdr.iocnum, &iocp) <
11     return -EFAULT;
12
13   // dependency usage
14   mutex_lock(&iocp->ioctl_cmds.mutex);
15   struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;
16
17   // second fetch
18   if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
19     return -EFAULT;
20
21   // BUG: kfwdl.iocnum might not equal to khdr.
22   mptctl_do_fw_download(kfwdl.iocnum, ......);
23   mutex_unlock(&iocp->ioctl_cmds.mutex);
24 }
```

Overlapped variable *V*:
`khdr.iocnum`

Data dependence:
`mpt_verify_adapter(khdr.iocnum, &iocp)`

Expect:
`kfwdl.iocnum == khdr.iocnum`

Double-Fetch Bugs:
Towards A Formal Definition

1. Two fetches from userspace memory that cover an **overlapped** region.

2. A relation must exist on the overlapped region between the two fetches. The relation can be either **control-dependence** or **data-dependence**.

3. We cannot **prove** that the relation established after first fetch still holds after the second fetch.

If all conditions are satisfied: a user thread might race condition to change the content in the overlapped region, and thus, to destroy the relation.

How to Find Double-Fetch Bugs?

How to Find Double-Fetch Bugs?

1. Find as many double-fetch pairs as possible, construct the code paths associated with each pair.

2. Symbolically check each code path and determine whether the two fetches makes a double-fetch bug.

Fetch Pair Collection

Goal: Statically enumerate all pairs of fetches that could possibly occur.

Fetch Pairs Collection

Goal: Statically enumerate all pairs of fetches that could possibly occur.

Ideal solution (top-down):

1. ✓ Identify all fetches in the kernel
2. ✗ Construct a complete, inter-procedural CFG for the whole kernel
3. ✗ Perform pair-wise reachability tests for each pair of fetches

Fetch Pairs Collection

Goal: Statically enumerate all pairs of fetches that could possibly occur.

Ideal solution (top-down):

✓ 1. Identify all fetches in the kernel
✗ 2. Construct a complete, inter-procedural CFG for the whole kernel
✗ 3. Perform pair-wise reachability tests for each pair of fetches

Our solution (bottom-up):

✓ 1. Identify all fetches in the kernel
✓ 2. For each fetch, within the function it resides in, scan its reaching instructions for fetches or fetch-involved functions

Bottom-up Fetch Pairs Collection

```
static void enclosing_function(
   struct msg_hdr___user *uptr,
   struct msg_full *kptr
) {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   if (copy_from_user(kptr, uptr, size))
      return -EFAULT;
   ...
}
```

Start from a fetch →

Bottom-up Fetch Pairs Collection

Search through the reaching instructions →

```
static void enclosing_function(
   struct msg_hdr___user *uptr,
   struct msg_full *kptr
) {
   ...
   ...
   ...
   ...
   ...
   ...
   if (copy_from_user(kptr, uptr, size))
      return -EFAULT;
   ...
}
```

Bottom-up Fetch Pairs Collection

[Case 1]
Found another fetch
==>
found a fetch pair

```
static void enclosing_function(
   struct msg_hdr___user *uptr,
   struct msg_full *kptr
) {
   ...
   ...
   if (get_user(size, &uptr->size))
     return -EFAULT;
   ...
   ...
   ...
   if (copy_from_user(kptr, uptr, size))
      return -EFAULT;
   ...
}
```

Bottom-up Fetch Pairs Collection

[Case 2]

Found a fetch-involved function

==>

inline the function, found a fetch pair

```
static void enclosing_function(
   struct msg_hdr___user *uptr,
   struct msg_full *kptr
) {
   ...
   ...
   size = get_size_from_user(uptr);
   ...
   ...
   ...
   ...
   if (copy_from_user(kptr, uptr, size))
      return -EFAULT;
   ...
}
```

Bottom-up Fetch Pairs Collection

[Case 3]
No fetch-related instruction
==>
Not a double-fetch

```
static void enclosing_function(
   struct msg_hdr___user *uptr,
   struct msg_full *kptr
) {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   if (copy_from_user(kptr, uptr, size))
      return -EFAULT;
   ...
}
```

How to Find Double-Fetch Bugs?

✓ 1. Find as many double-fetch pairs as possible, construct the code paths associated with each pair.

2. Symbolically check each code path and determine whether the two fetches makes a double-fetch bug.

Symbolic Checking

Goal: Symbolically execute the code path that connects two fetches and determine whether the two fetches satisfy all the criteria set in formal definition of double-fetch bug, i.e.

- Overlapp
- Have a relation (control or data dependence)
- We cannot prove the relation still holds after second fetch

Symbolic Checking

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr, size))
18      return -EFAULT;
19
20    .......
21 }
22
23 //  BUG: when attr->size is used later
24 memcpy(buf,  attr, attr->size);
                                                    U1)
```

Symbolic Checking

```
static int perf_copy_attr_simplified
   (struct perf_event_attr __user *uattr,
    struct perf_event_attr *attr) {

   u32 size;

   // first fetch
   if (get_user(size, &uattr->size))
     return -EFAULT;

   // sanity checks
   if (size > PAGE_SIZE ||
       size < PERF_ATTR_SIZE_VER0)
     return -EINVAL;

   // second fetch
   if (copy_from_user(attr, uattr, size))
     return -EFAULT;

   ......
}

//  BUG: when attr->size is used later
memcpy(buf,  attr, attr->size);
```

```
// init root SR
$0  = PARM(0), @0 = UMEM(0)   // uattr
$1  = PARM(1), @1 = KMEM(1)   // attr
---
```

Symbolic Checking

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr-
>size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr, size))
18      return -EFAULT;
19
20    .......
21 }
22
23 //  BUG: when attr->size is used later
24 memcpy(buf,  attr, attr->size);
```

```
1 // init root SR
2 $0  = PARM(0), @0 = UMEM(0)   // uattr
3 $1  = PARM(1), @1 = KMEM(1)   // attr
4 ---
5 // first fetch
6 fetch(F1):  {A = $0 + 4, S = 4}
7 $2  = @0(4, 7, U0), @2 = nil   // size
8 ---
```

Symbolic Checking

```
static int perf_copy_attr_simplified
   (struct perf_event_attr _user *uattr,
    struct perf_event_attr *attr) {

   u32 size;

   // first fetch
   if (get_user(size, &uattr->size))
     return -EFAULT;

   // sanity checks
   if (size > PAGE_SIZE ||
       size <
       PERF_ATTR_SIZE_VER0)
     return -EINVAL;
   // second fetch
   if (copy_from_user(attr, uattr,
     return -EFAULT;
                                    size))

   ......
}

//  BUG: when attr->size is used later
memcpy(buf,  attr, attr->size);
```

```
// init root SR
$0 = PARM(0), @0 = UMEM(0)  // uattr
$1 = PARM(1), @1 = KMEM(1)  // attr
---
// first fetch
fetch(F1): {A = $0 + 4, S = 4}
$2 = @0(4, 7, U0), @2 = nil  // size
---
// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0
---
                                    U1)
```

Symbolic Checking

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr,       )
      size)
18      return -EFAULT;
20    ......
21 }
22
23 //  BUG: when attr->size is used later
24 memcpy(buf,  attr, attr->size);
```

```
1  // init root SR
2  $0  = PARM(0), @0 = UMEM(0)   // uattr
3  $1  = PARM(1), @1 = KMEM(1)   // attr
4  ---
5  // first fetch
6  fetch(F1):  {A = $0 + 4, S = 4}
7  $2  = @0(4, 7, U0), @2 = nil   // size
8  ---
9  // sanity checks
10 assert $2  <= PAGE_SIZE
11 assert $2  >= PERF_ATTR_SIZE_VER0
12 ---
13 // second fetch
14 fetch(F2):  {A = $0, S = $2}
15 @1(0,  $2 - 1, K) = @0(0, $2 - 1, U1)
16 ---
                                  U1)
```

Symbolic Checking

```
static int perf_copy_attr_simplified
   (struct perf_event_attr _user *uattr,
    struct perf_event_attr *attr) {

  u32 size;

  // first fetch
  if (get_user(size, &uattr->size))
    return -EFAULT;

  // sanity checks
  if (size > PAGE_SIZE ||
      size < PERF_ATTR_SIZE_VER0)
    return -EINVAL;

  // second fetch
  if (copy_from_user(attr, uattr, size))
    return -EFAULT;

  ......
}

// BUG: when attr->size is used later
memcpy(buf, attr, attr->size);
```

```
// init root SR
$0 = PARM(0), @0 = UMEM(0)  // uattr
$1 = PARM(1), @1 = KMEM(1)  // attr
---
// first fetch
fetch(F1): {A = $0 + 4, S = 4}
$2 = @0(4, 7, U0), @2 = nil  // size
---
// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0
---
// second fetch
fetch(F2): {A = $0, S = $2}
@1(0, $2 - 1, K) = @0(0, $2 - 1, U1)
---
// check fetch overlap
assert F2.A <= F1.A < F2.A + F2.S
    OR F1.A <= F2.A < F1.A + F1.S
[solve]
  --> satisfiable with @0(4, 7, U)
```

U1)

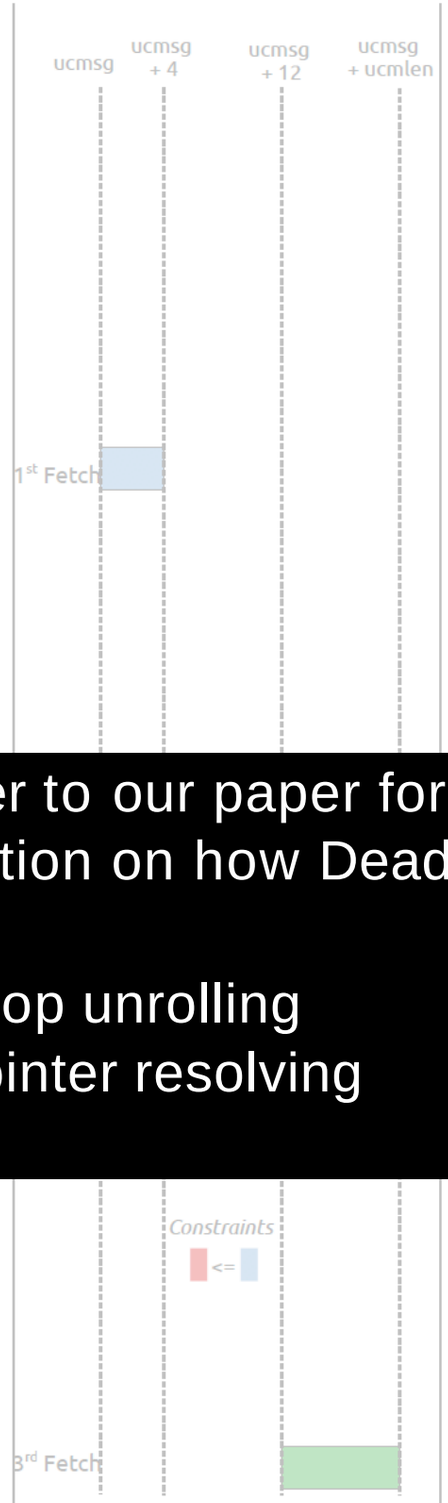
Symbolic Checking

```
1  static int perf_copy_attr_simplified
2     (struct perf_event_attr _user *uattr,
3      struct perf_event_attr *attr) {
4
5     u32 size;
6
7     // first fetch
8     if (get_user(size, &uattr->size))
9       return -EFAULT;
10
11    // sanity checks
12    if (size > PAGE_SIZE ||
13        size < PERF_ATTR_SIZE_VER0)
14      return -EINVAL;
15
16    // second fetch
17    if (copy_from_user(attr, uattr, size))
18      return -EFAULT;
19
20    .......
21 }
22
23 //  BUG: when attr->size is used later
24 memcpy(buf,  attr, attr->size);
```

```
1  // init root SR
2  $0 = PARM(0), @0 = UMEM(0)   // uattr
3  $1 = PARM(1), @1 = KMEM(1)   // attr
4  ---
5  // first fetch
6  fetch(F1): {A = $0 + 4, S = 4}
7  $2 = @0(4, 7, U0), @2 = nil   // size
8  ---
9  // sanity checks
10 assert $2 <= PAGE_SIZE
11 assert $2 >= PERF_ATTR_SIZE_VER0
12 ---
13 // second fetch
14 fetch(F2): {A = $0, S = $2}
15 @1(0, $2 - 1, K) = @0(0, $2 - 1, U1)
16 ---
17 // check fetch overlap
18 assert F2.A <= F1.A < F2.A + F2.S
19     OR F1.A <= F2.A < F1.A + F1.S
20 [solve]
21   --> satisfiable with @0(4, 7, U)
22 // check double-fetch bug
23 [prove] @0(4, 7, U0) == @0(4, 7, U1)
24   --> fail: no constraints on @0(4, 7, U1)
```

```c
int cmsghdr_from_user_compat_to_kern
  (struct msghdr *kmsg, char *kbuf) {

  struct compat_cmsghdr __user *ucmsg;
  compat_size_t ucmlen;
  struct cmsghdr *kcmsg;
  __kernel_size_t kcmlen, tmp;

  // 1st loop: calculate message length
  kcmlen = 0;
  ucmsg = kmsg->msg_control;
  while (ucmsg != NULL) {
    // first batch of fetches
    if (get_user(ucmlen, &ucmsg->cmsg_len))
      return -EFAULT;

    tmp = ucmlen + sizeof(struct cmsghdr)
          - sizeof(struct compat_cmsghdr);

    kcmlen += tmp;
    ucmsg = (char *)ucmsg + ucmlen;
  }

  // 2nd loop: copy the who
  kcmsg = kbuf;
  ucmsg = kmsg->msg_control
  while (ucmsg != NULL) {
    // secind batch of fetc
    if (get_user(ucmlen, &u
      return -EFAULT;

    tmp = ucmlen + sizeof(s
          - sizeof(struct c

    // sanity check, but insufficient
    if (kbuf + kcmlen - (char *)kcmsg < tmp)
      return -EINVAL;

    // irrelevant fetch
    if (copy_from_user(
        (char *)kcmsg + sizeof(*kcmsg),
        (char *)ucmsg + sizeof(*ucmsg),
        (ucmlen - sizeof(*ucmsg))))
      return -EFAULT;

    kcmsg = (char *)kcmsg + tmp;
    ucmsg = (char *)ucmsg + ucmlen;
  }

  // BUG: the actual message length != kcmlen
  kmsg->msg_controllen = kcmlen;
  return 0;
}
```

(a) C source code



(b) Memory access patterns

```
// init root SR
$0  = $PARM(0),              @0  = $KMEM(0)  // kmsg
$1  = $PARM(1),              @1  = $KMEM(1)  // kbuf
---
// prepare for the 1st batch of fetches
$2  = 0,                     @2  = nil       // kcmlen_0
$3  = @0(48, 55, K),         @3  = $UMEM(0)  // ucmsg_0
---
// unroll 1st loop
assert $2 != NULL
fetch(F1) is {A = $3 + 0, S = 4}
$4  = @3(0, 3, U0),          @4  = nil       // ucmlen_0
$5  = $4 - 12 + 16,          @5  = nil       // tmp_0
$6  = $2 + $5,               @6  = nil       // kcmlen_1
$7  = $3 + $4,               @7  = $UMEM(1)  // ucmsg_1
assert $7 == NULL (i.e., @7 = nil)           // exit loop
---
// prepare for the 2nd batch of fetches
$8  = $1                     @8  = $KMEM(1)  // kcmsg_0
$9  = @0(48, 55, K) == $3,   @9  = @3        // ucmsg_2
---
// unroll 2nd loop
                                                    4}
                                             0 = nil       // ucmlen_1
                                             1 = nil       // tmp_1
                                          > @3(0, 3, U0) >= @3(0, 3, U1)
                                              $10 - 12}
                                          10 - 13, U0)
                                          2 = $KMEM(2)  // kcmsg_1
                                          3 = $UMEM(3)  // ucmsg_3
assert $13 == NULL (i.e., @13 = nil)         // exit loop
---

// check fetch overlap
assert F2.A <= F1.A < F2.A + F2.S
   AND F1.A <= F2.A < F1.A + F1.S
// --> satisfiable with @3(0, 3, U)

assert F3.A <= F1.A < F3.A + F3.S
   AND F1.A <= F3.A < F1.A + F1.S
// --> unsatisfiable

assert F3.A <= F2.A < F3.A + F3.S
   AND F2.A <= F3.A < F2.A + F2.S
// --> unsatisfiable

// check double-fetch bug
prove @3(0, 3, U0) == @3(0, 3, U1)
// --> fail, as @3(0, 3, U0) >= @3(0, 3, U1)
```

(c) Symbolic representation and checking

Please refer to our paper for a comprehensive demonstration on how Deadline handles

1. Loop unrolling
2. Pointer resolving

Findings

- 24 bugs found in total
  - 23 bugs in Linux kernel and 1 in FreeBSD kernel
- 9 bugs have been patched with the fix we provide
- 4 bugs are acknowledged, we are still working on the fix
- 9 bugs are pending for review
- 2 bugs are marked as “won’t fix”

Double-Fetch Bug Mitigations

- The basic idea is to re-assure the control-dependence and data-dependence between the two fetches. In other words, the **automaticity** in user space memory fetches during the execution of the syscall.

Double-Fetch Bug Mitigations

- The basic idea is to re-assure the control-dependence and data-dependence between the two fetches. In other words, the **automaticity** in user space memory fetches during the execution of the syscall.

- Based on our experience and our communications with kernel developers, we found **four patterns** in patching double-fetch bugs.

Double-Fetch Bug Mitigations

1. Override after second fetch.

```
kernel/events/core.c | 2 ++
1 file changed, 2 insertions(+)

diff --git a/kernel/events/core.c b/kernel/events/core.c
index ee20d4c..c0d7946 100644
--- a/kernel/events/core.c
+++ b/kernel/events/core.c
@@ -9611,6 +9611,8 @@ static int perf_copy_attr(struct perf_event_attr __user *uattr,
     if (ret)
         return -EFAULT;

+    attr->size = size;
+
     if (attr->__reserved_1)
         return -EINVAL;
```

Override the overlapped memory (`attr->size`) with the value from the first fetch (`size`).

Double-Fetch Bug Mitigations

2. Abort on change detected.

```
 net/compat.c | 7 +++++++
 1 file changed, 7 insertions(+)

diff --git a/net/compat.c b/net/compat.c
index 6ded6c8..2238171 100644
--- a/net/compat.c
+++ b/net/compat.c
@@ -185,6 +185,13 @@ int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg, struct sock *sk,
         ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
     }

+   /*
+    * check the length of messages copied in is the same as the
+    * what we get from the first loop
+    */
+   if ((char *)kcmsg - (char *)kcmsg_base != kcmlen)
+       goto Einval;
+
     /* Ok, looks like we made it.  Hook it up and return success. */
     kmsg->msg_control = kcmsg_base;
     kmsg->msg_controllen = kcmlen;
```

Compare the new message length (`kcmsg - kcmsg_base`) with the value from the first fetch (`kcmlen`).

Double-Fetch Bug Mitigations

3. Refactor overlapped copies into incremental copies.

```
 block/scsi_ioctl.c | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

diff --git a/block/scsi_ioctl.c b/block/scsi_ioctl.c
index 7440de4..8fe1e05 100644
--- a/block/scsi_ioctl.c
+++ b/block/scsi_ioctl.c
@@ -463,7 +463,13 @@ int sg_scsi_ioctl(struct request_queue *q, struct gendisk *disk, fmode_t mode,
      */
     err = -EFAULT;
     req->cmd_len = cmdlen;
-    if (copy_from_user(req->cmd, sic->data, cmdlen))
+
+    /*
+     * avoid copying the opcode twice
+     */
+    memcpy(req->cmd, &opcode, sizeof(opcode));
+    if (copy_from_user(req->cmd + sizeof(opcode),
+                sic->data + sizeof(opcode), cmdlen - sizeof(opcode)))
         goto error;

     if (in_len && copy_from_user(buffer, sic->data + cmdlen, in_len))
```

When copying the whole message, skip the information copied in the first fetch (`+ sizeof(opcode)`).

Double-Fetch Bug Mitigations

4. Refactor overlapped copies into a single-fetch.

```
 drivers/isdn/i4l/isdn_ppp.c | 37 +++++++++++++++++++++++++------------
 1 file changed, 25 insertions(+), 12 deletions(-)

diff --git a/drivers/isdn/i4l/isdn_ppp.c b/drivers/isdn/i4l/isdn_ppp.c
index 6c44609..cd2b3c6 100644
--- a/drivers/isdn/i4l/isdn_ppp.c
+++ b/drivers/isdn/i4l/isdn_ppp.c
@@ -825,7 +825,6 @@ isdn_ppp_write(int min, struct file *file, const char __user *buf, int count)
    isdn_net_local *lp;
    struct ippp_struct *is;
    int proto;
-   unsigned char protobuf[4];

    is = file->private_data;

@@ -839,24 +838,28 @@ isdn_ppp_write(int min, struct file *file, const char __user *buf, int count)
    if (!lp)
        printk(KERN_DEBUG "isdn_ppp_write: lp == NULL\n");
    else {
-       /*
-        * Don't reset huptimer for
-        * LCP packets. (Echo requests).
-        */
-       if (copy_from_user(protobuf, buf, 4))
-           return -EFAULT;
-       proto = PPP_PROTOCOL(protobuf);
-       if (proto != PPP_LCP)
-           lp->huptimer = 0;
+       if (lp->isdn_device < 0 || lp->isdn_channel < 0) {
+           unsigned char protobuf[4];
+           /*
+            * Don't reset huptimer for
+            * LCP packets. (Echo requests).
+            */
+           if (copy_from_user(protobuf, buf, 4))
+               return -EFAULT;
+
+           proto = PPP_PROTOCOL(protobuf);
+           if (proto != PPP_LCP)
+               lp->huptimer = 0;

-       if (lp->isdn_device < 0 || lp->isdn_channel < 0)
            return 0;
+       }

        if ((dev->drv[lp->isdn_device]->flags & DRV_FLAG_RUNNING) &&
            lp->dialstate == 0 &&
            (lp->flags & ISDN_NET_CONNECTED)) {
            unsigned short hl;
            struct sk_buff *skb;
+           unsigned char *cpy_buf;
            /*
             * we need to reserve enough space in front of
             * sk_buff. old call to dev_alloc_skb only reserved
@@ -869,11 +872,21 @@ isdn_ppp_write(int min, struct file *file, const char __user *buf, int count)
                return count;
            }
            skb_reserve(skb, hl);
-           if (copy_from_user(skb_put(skb, count), buf, count))
+           cpy_buf = skb_put(skb, count);
+           if (copy_from_user(cpy_buf, buf, count))
            {
                kfree_skb(skb);
                return -EFAULT;
            }
+
+           /*
+            * Don't reset huptimer for
+            * LCP packets. (Echo requests).
+            */
+           proto = PPP_PROTOCOL(cpy_buf);
+           if (proto != PPP_LCP)
+               lp->huptimer = 0;
+
            if (is->debug & 0x40) {
                printk(KERN_DEBUG "ppp xmit: len %d\n", (int) skb->len);
                isdn_ppp_frame_log("xmit", skb->data, skb->len, 32, is->unit, lp->ppp_slot);
```

Such a strategy is usually very complex and requires careful refactoring.

Limitations of Deadline

- **Source code coverage**
  - Files not compilable under LLVM.
  - Special combination of kernel configs (e.g., `CONFIG_*`).

- **Execution path construction**
  - Limit on total number of paths explored per fetch pair (4096).
  - Loop unrolling (limited to unroll once only).

- **Symbolic checking**
  - Ignores inline assemblies.
  - Imprecise pointer to memory object mapping.
  - Assumption on enclosing function.

Conclusion

- Detecting double-fetch bugs without a precise and formal definition has led to many false alerts and tremendous manual effort.
- Deadline is based on a precise modeling of double-fetch bugs and achieves both high accuracy and high scalability.
- Application beyond kernels: hypervisors, browsers, TEE, etc.
- Logic bugs are on the rise! We hope that more logic bugs can be modeled and checked systematically.

https://github.com/sslab-gatech/deadline