# 对内核中“二次获取”漏洞的精确以及 大范围检测

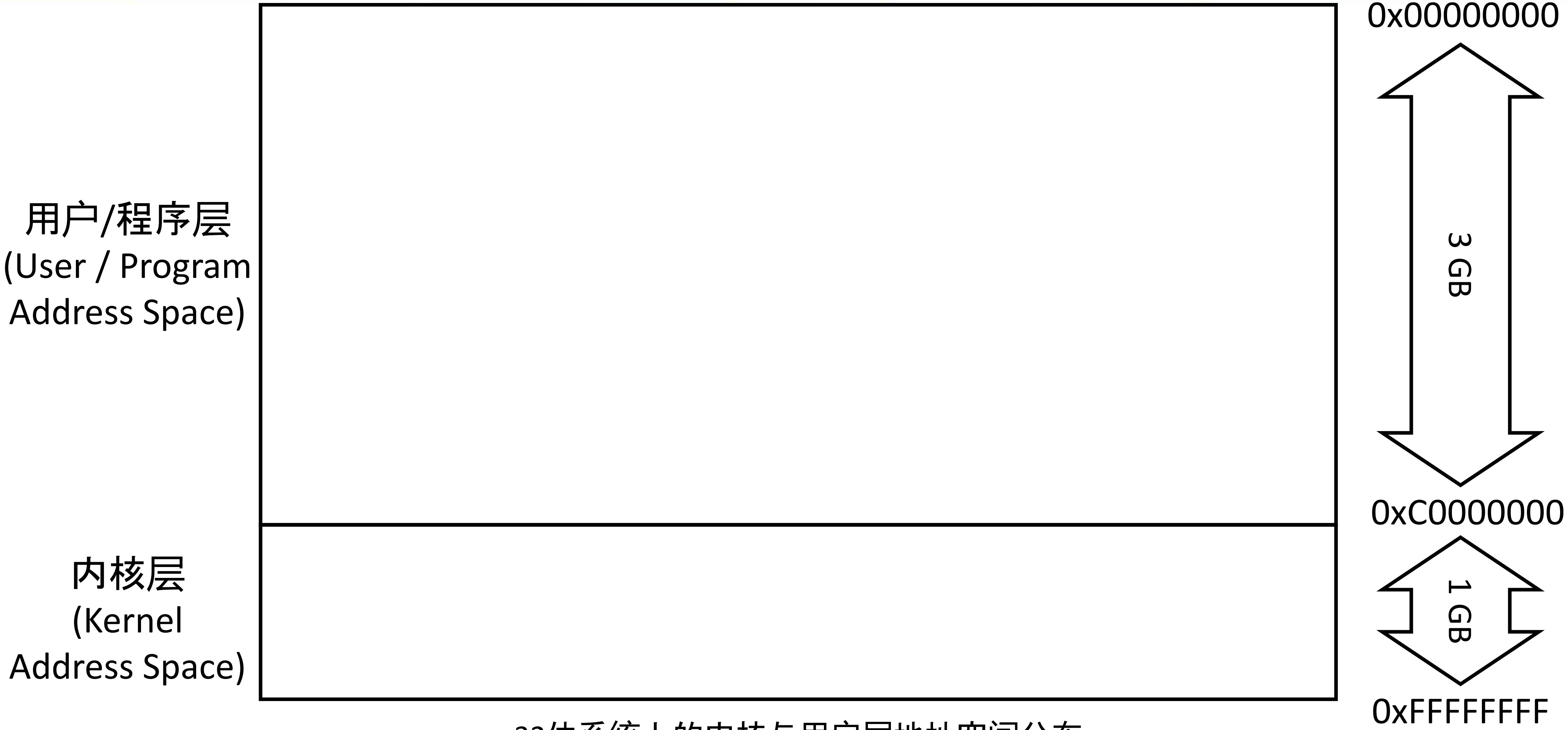
徐萌

佐治亚理工学院计算机系博士
在读 SSLab以及IISP成员

2018 ISC 互联网安全大会 中国 · 北京
Internet Security Conference 2018 Beijing · China
（原中国互联网安全大会）

# 什么是“二次获取”(What is Double-Fetch?)

# 地址空间分离 (Address Space Separation)



用户/程序层
(User / Program
Address Space)

内核层
(Kernel
Address Space)

0x00000000

3 GB

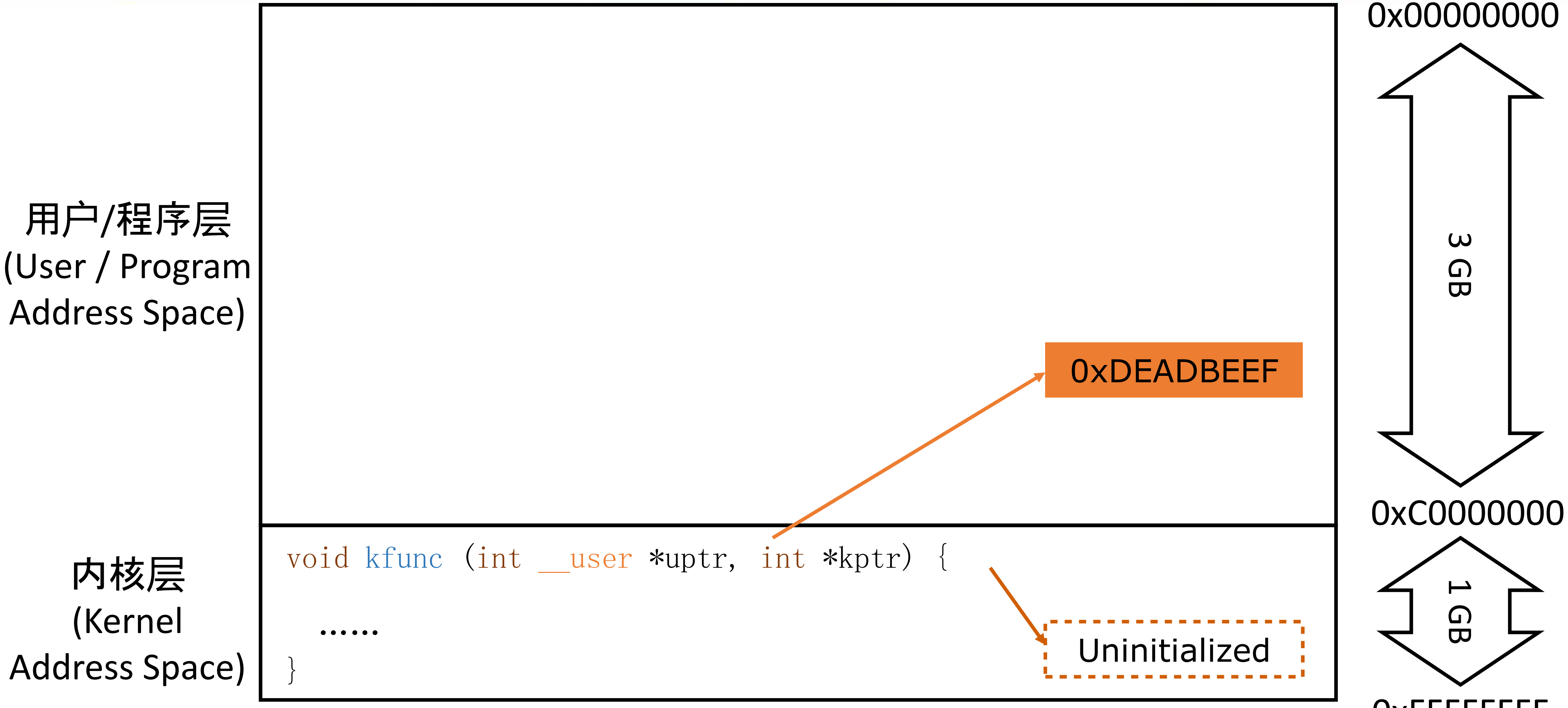
0xC0000000

1 GB

0xFFFFFFFF

32位系统上的内核与用户层地址空间分布
A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

# 单次获取 (How To Do A Single Fetch?)

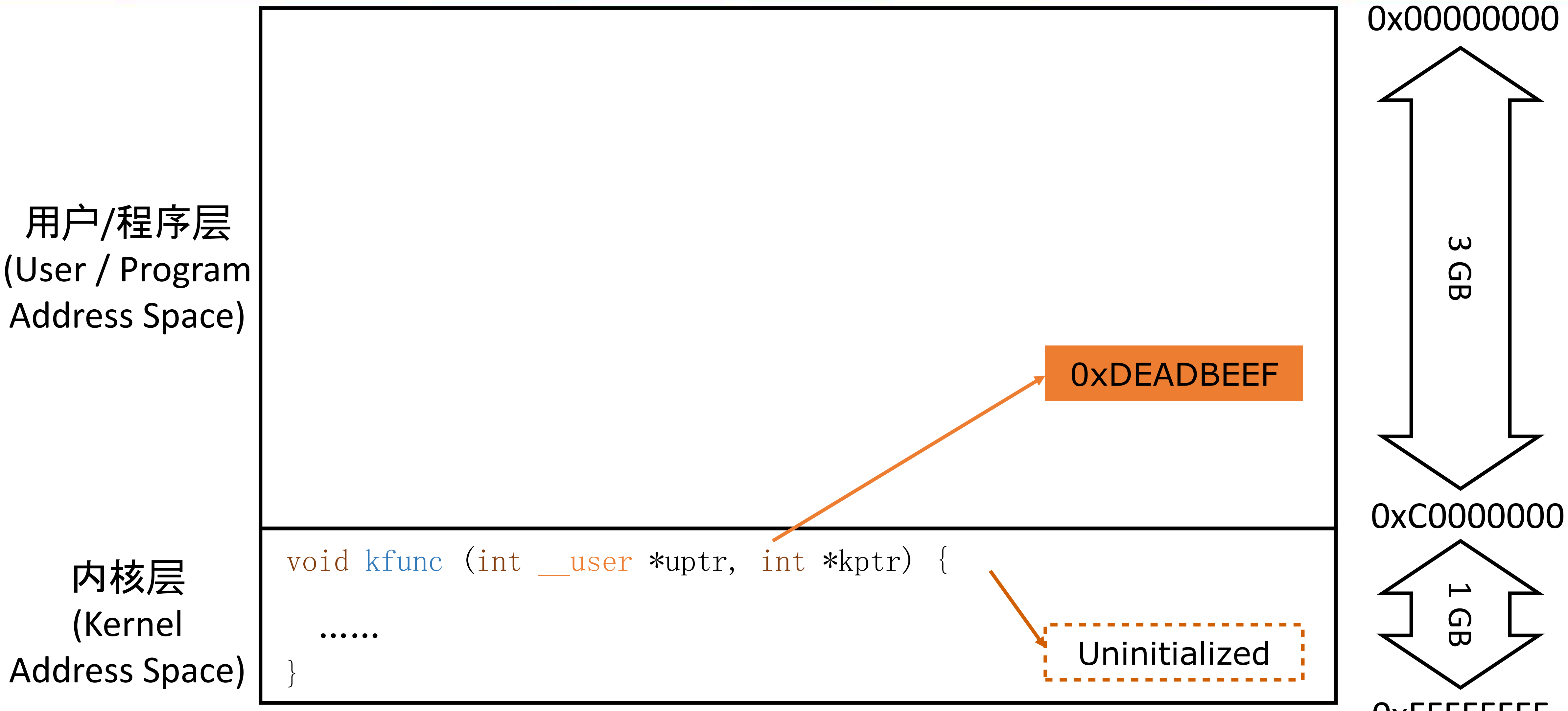


```
void kfunc (int __user *uptr, int *kptr) {
  ......
}
```

32位系统上的内核与用户层地址空间分布

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

# 单次获取 (How To Do A Single Fetch?)

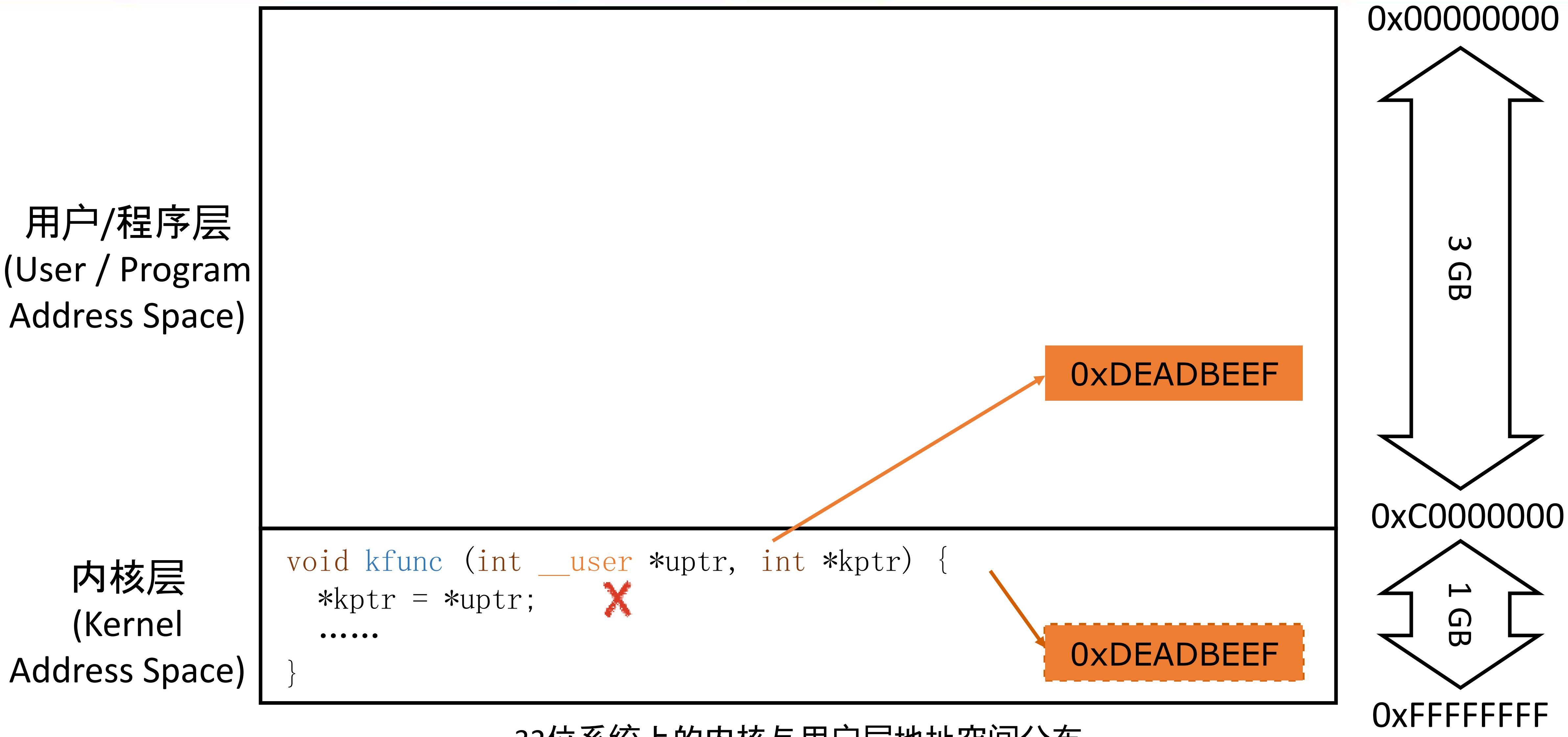


```
void kfunc (int __user *uptr, int *kptr) {
  ......
}
```

32位系统上的内核与用户层地址空间分布

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

# XXXXXX (No Dereference on Userspace Pointers)

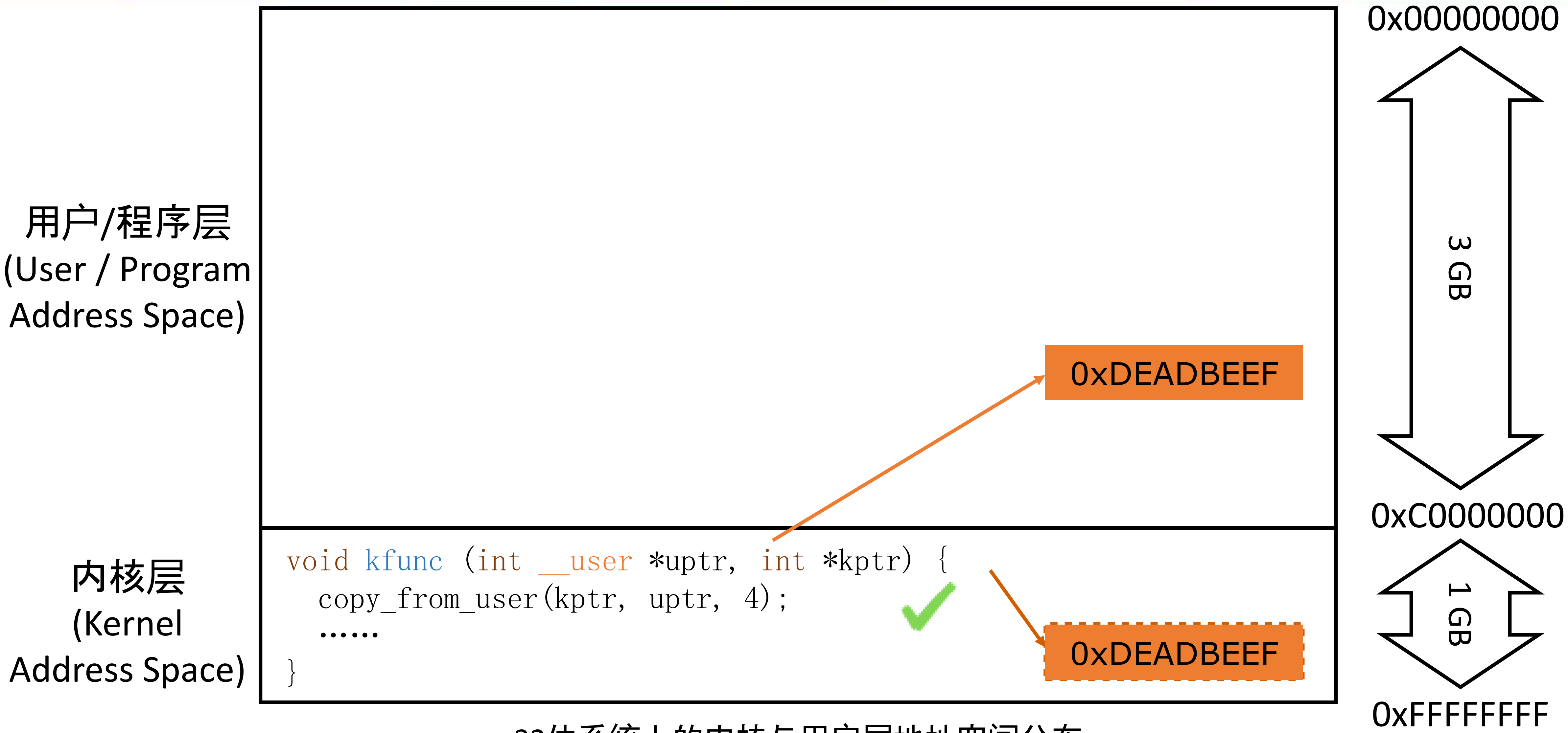


```
void kfunc (int __user *uptr, int *kptr) {
  *kptr = *uptr;     ✗
  ......
}
```

32位系统上的内核与用户层地址空间分布

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

# 指定的用户层内存访问函数 (Transfer Functions)

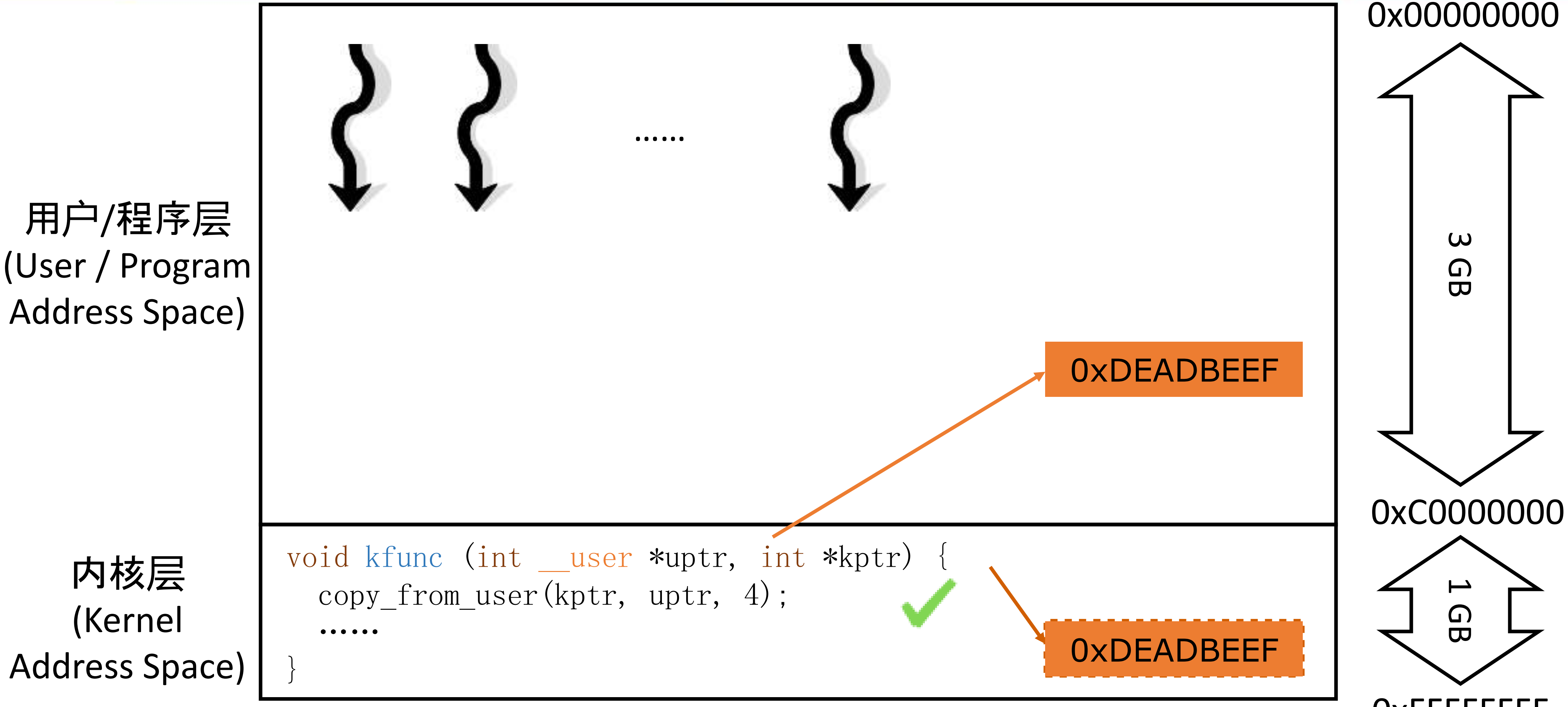


```
void kfunc (int __user *uptr, int *kptr) {
  copy_from_user(kptr, uptr, 4);
  ……
}
```

32位系统上的内核与用户层地址空间分布

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

# 用户层指针多线程共享 (Shared Userspace Pointer Across Threads)



```
void kfunc (int __user *uptr, int *kptr) {
  copy_from_user(kptr, uptr, 4);
  ......
}
```

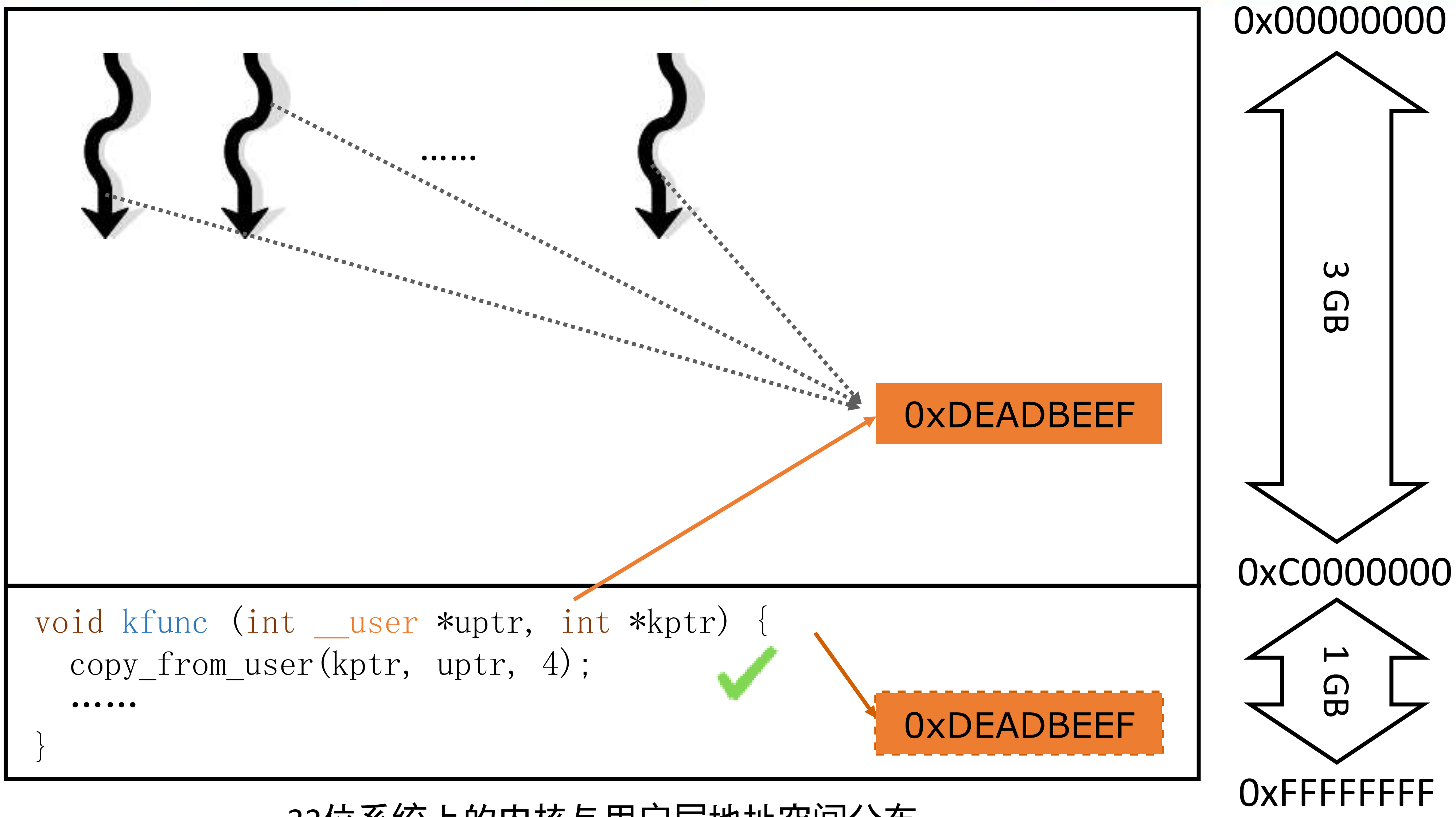
32位系统上的内核与用户层地址空间分布

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

# 用户层指针多线程共享 (Shared Userspace Pointer Across Threads)

用户/程序层
(User / Program
Address Space)

内核层
(Kernel
Address Space)

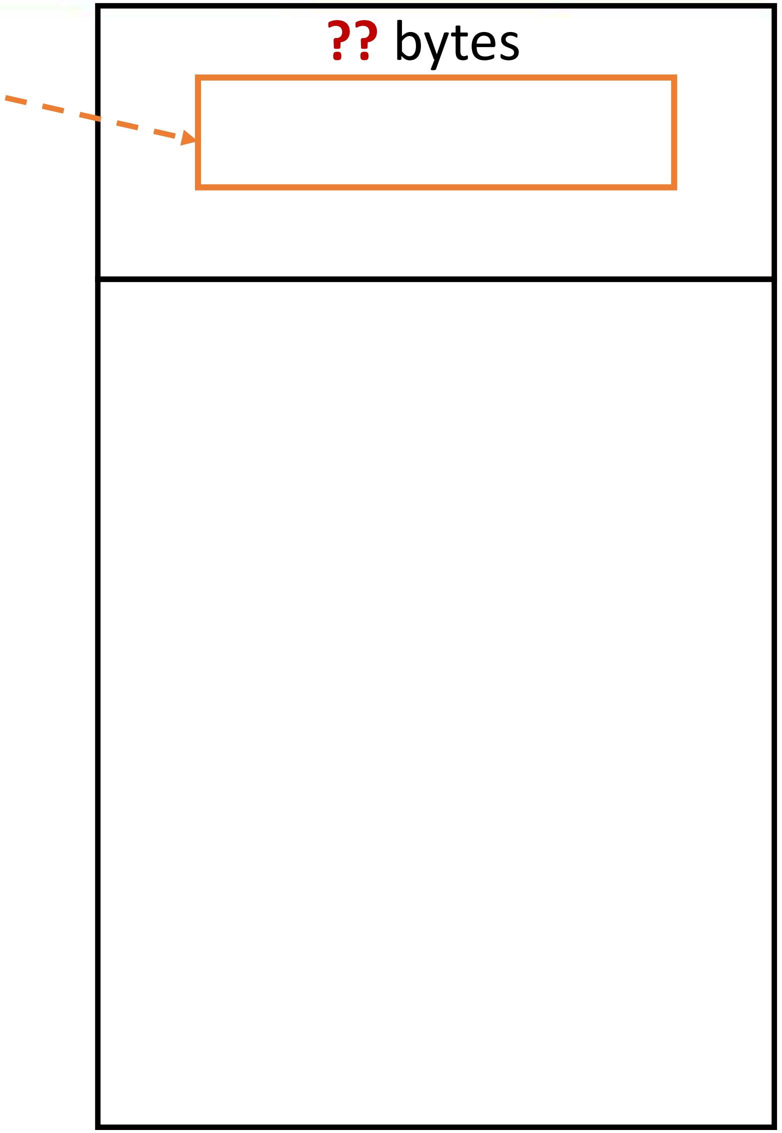


```
void kfunc (int __user *uptr, int *kptr) {
  copy_from_user(kptr, uptr, 4);
  ......
}
```

32位系统上的内核与用户层地址空间分布
A Typical Address Space Separation Scheme with a 32-bit Virtual Address Space

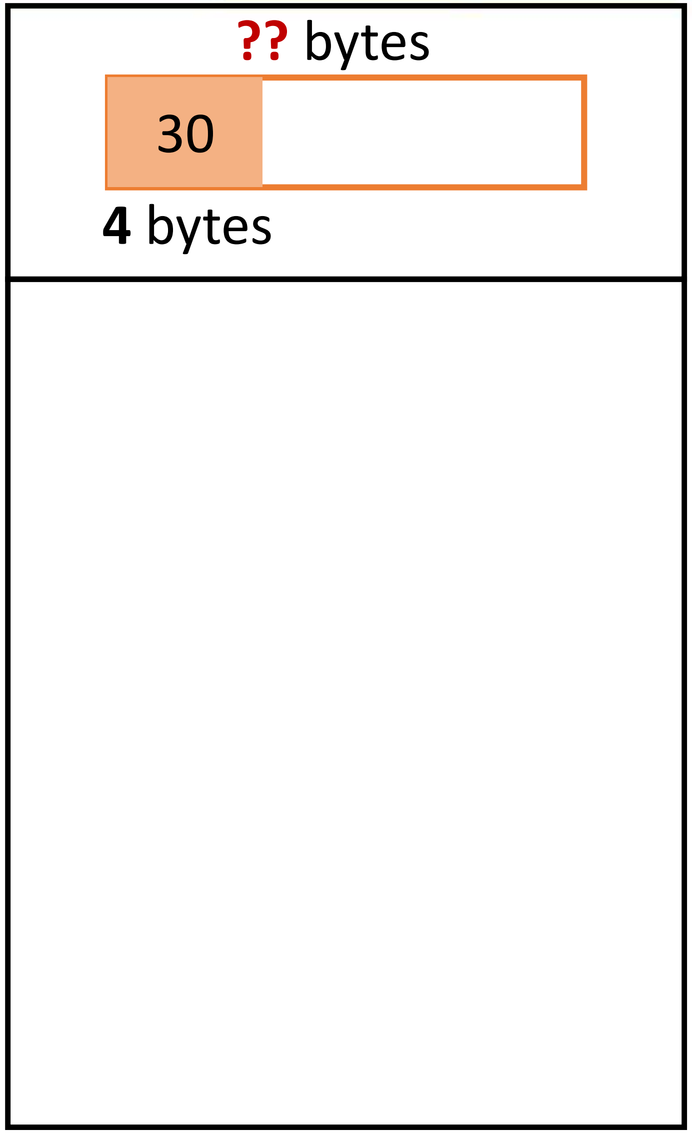
# 为什么要“二次获取”(Why Double-Fetch?)

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



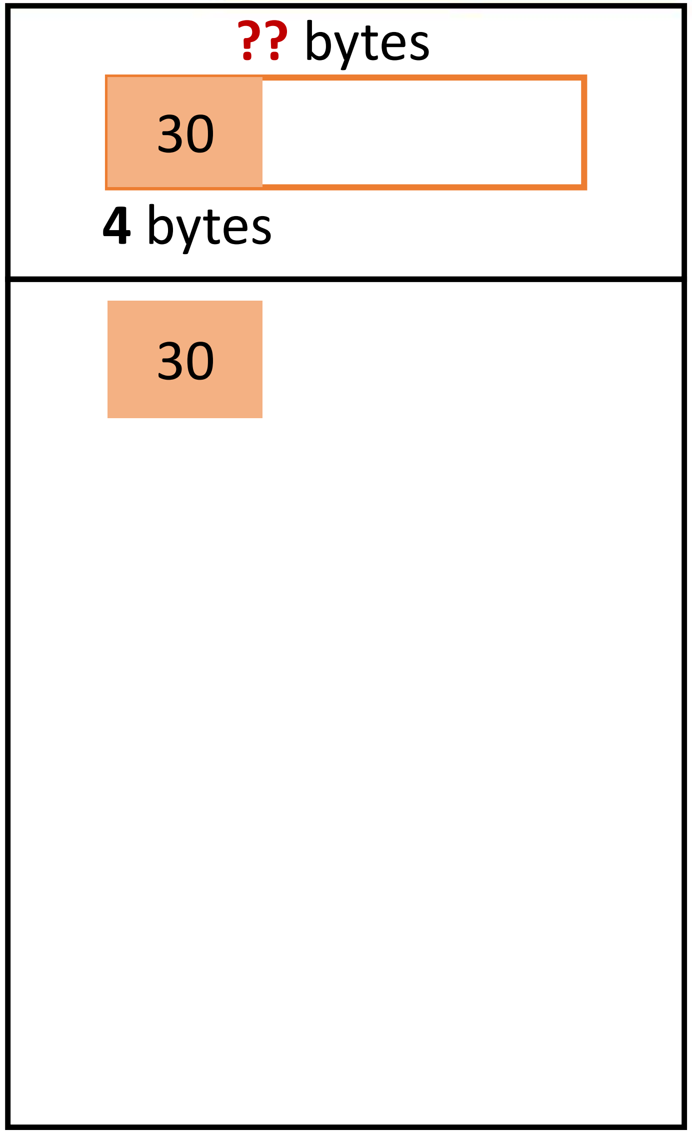
# 为什么要“二次获取”(Why Double-Fetch?)

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

  u32 size;
```



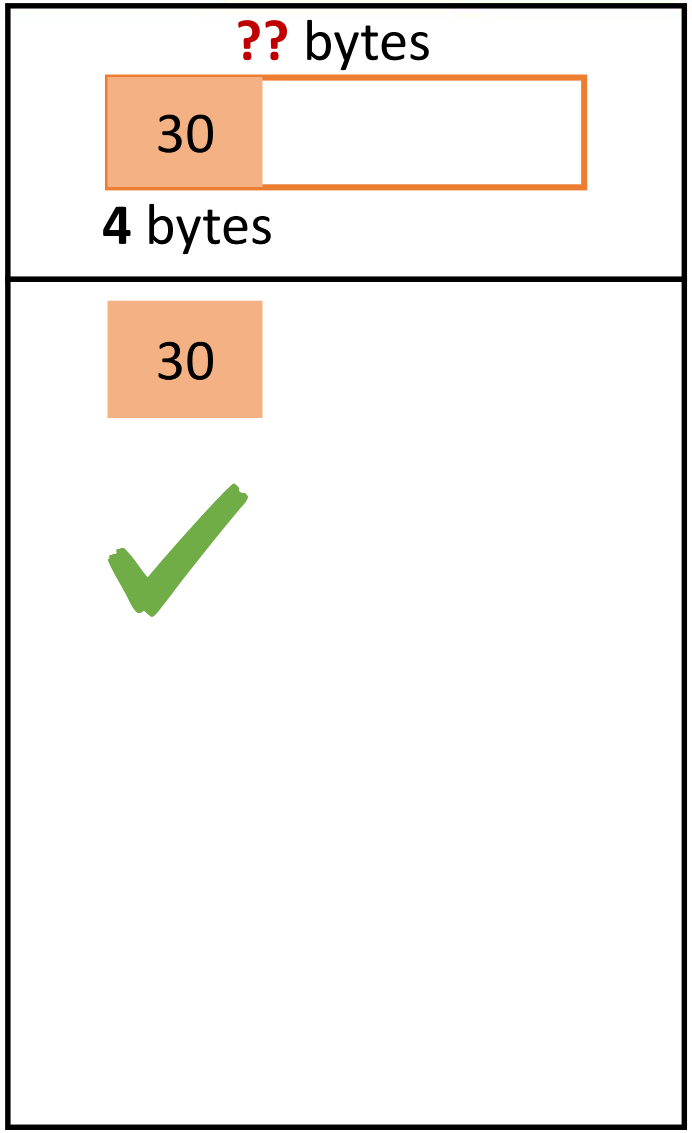
# 为什么要“二次获取”(Why Double-Fetch?)

```
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;
```



# 为什么要“二次获取”(Why Double-Fetch?)

```
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;
```



?? bytes

30

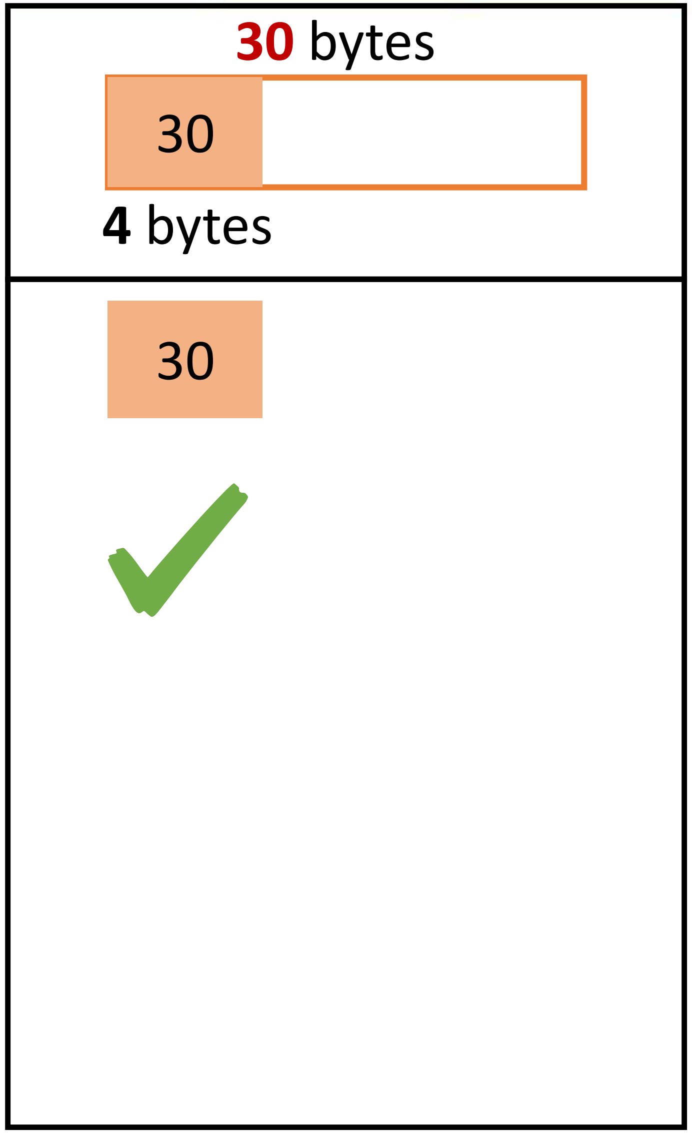
4 bytes

30

✓

# 为什么要“二次获取”(Why Double-Fetch?)

```
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;
```



**30** bytes

30

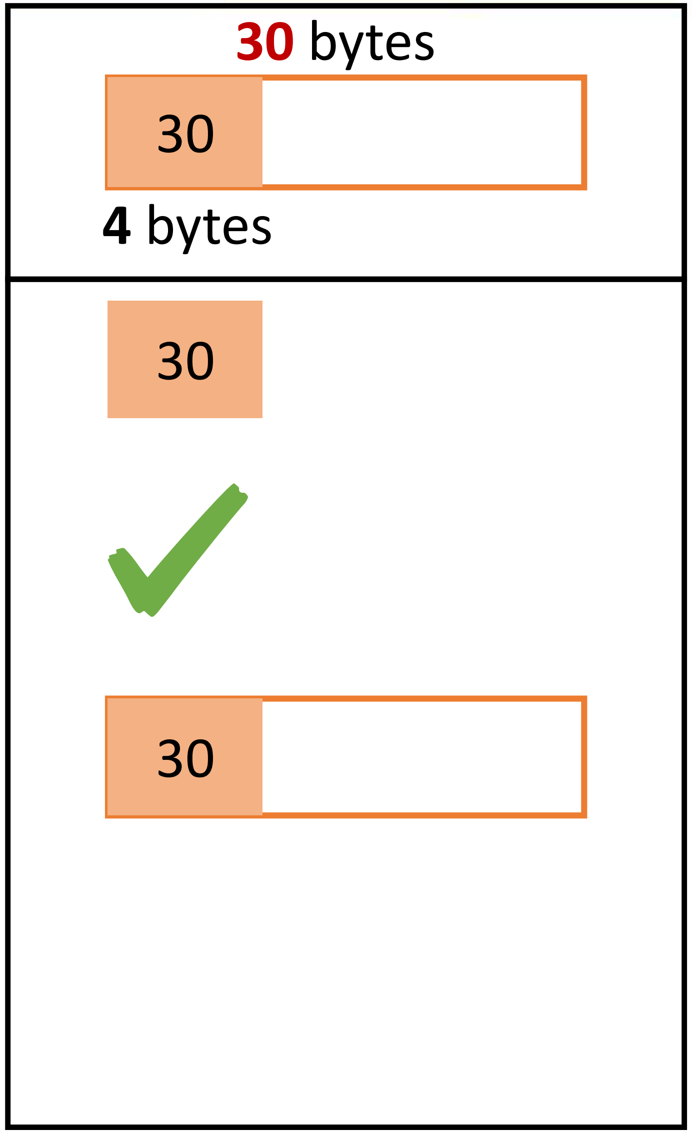
**4** bytes

30

✓

# 为什么要“二次获取”(Why Double-Fetch?)

```
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;
```



**30** bytes

30

**4** bytes

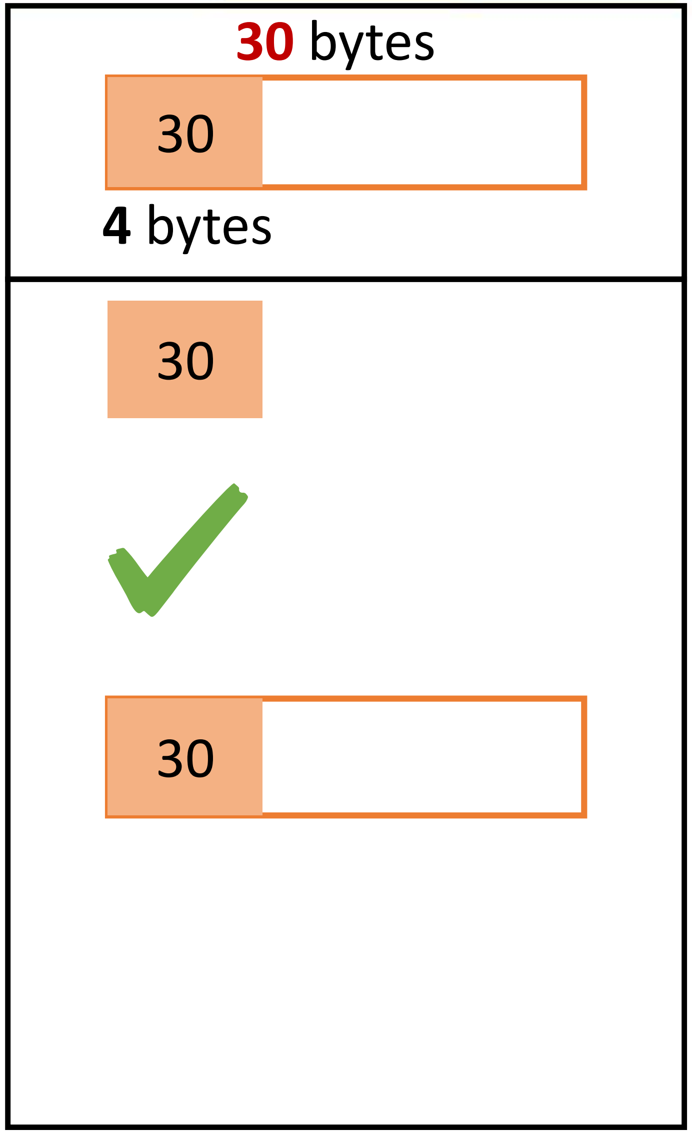
30

✓

30

# 为什么要“二次获取”(Why Double-Fetch?)

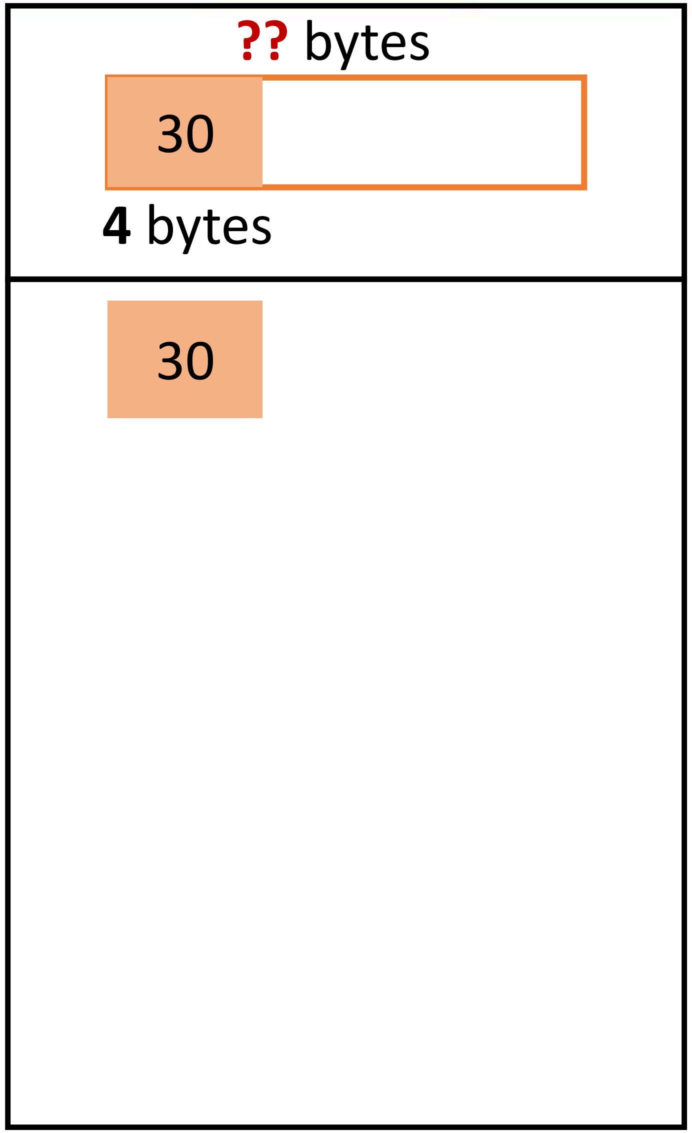
```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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;

  ......
}
```



# “二次获取”过程中潜藏的问题 (What Can Go Wrong in This Process?)

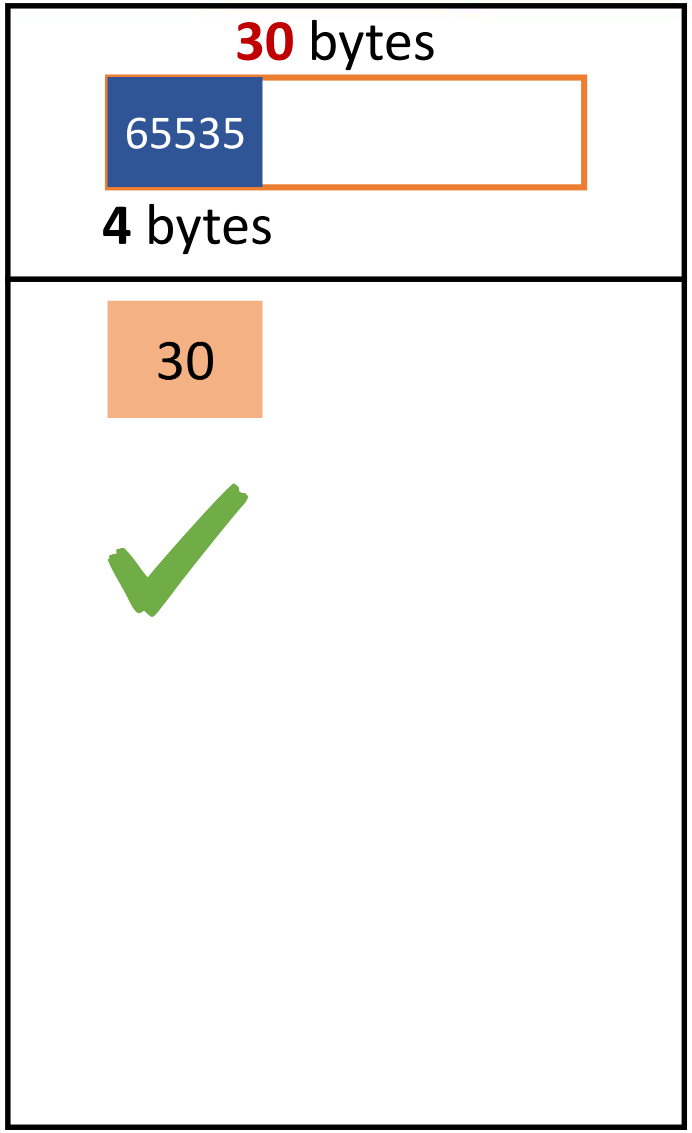
# 第一次获取之后的状态 (Right After the First Fetch)

```
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;
```



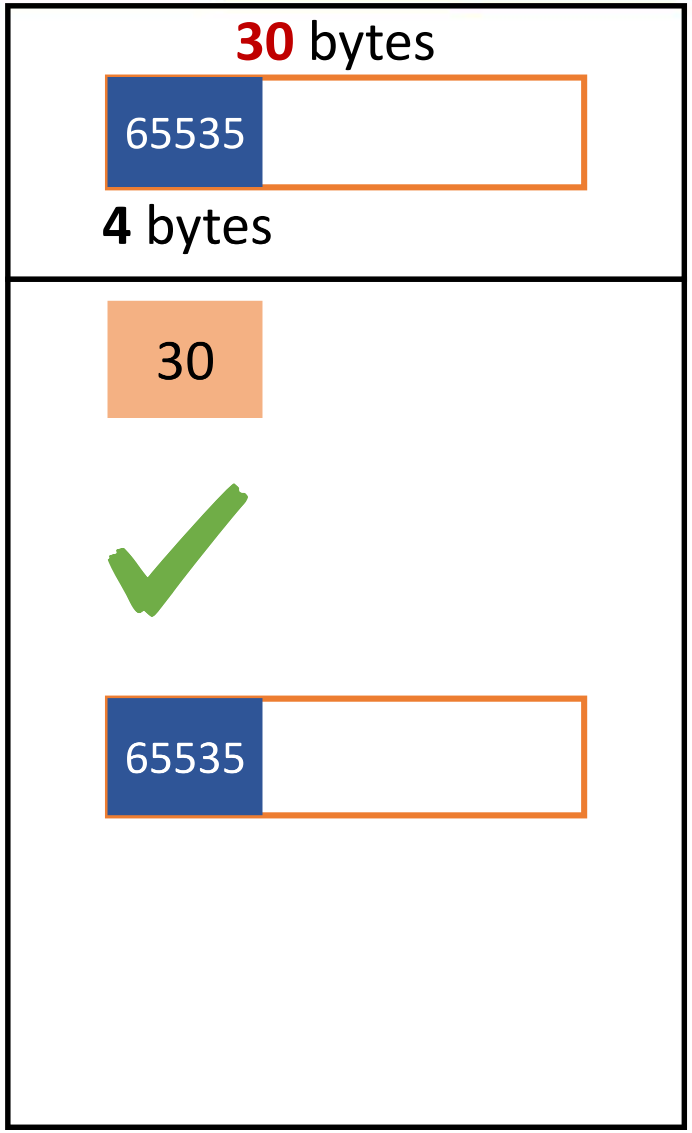
# 用户层内存访问冲突 (Race Condition in The Userspace)

```
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;
```



# 第二次获取之后的状态 (Right After the Second Fetch)

```
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;
```



**30** bytes

65535

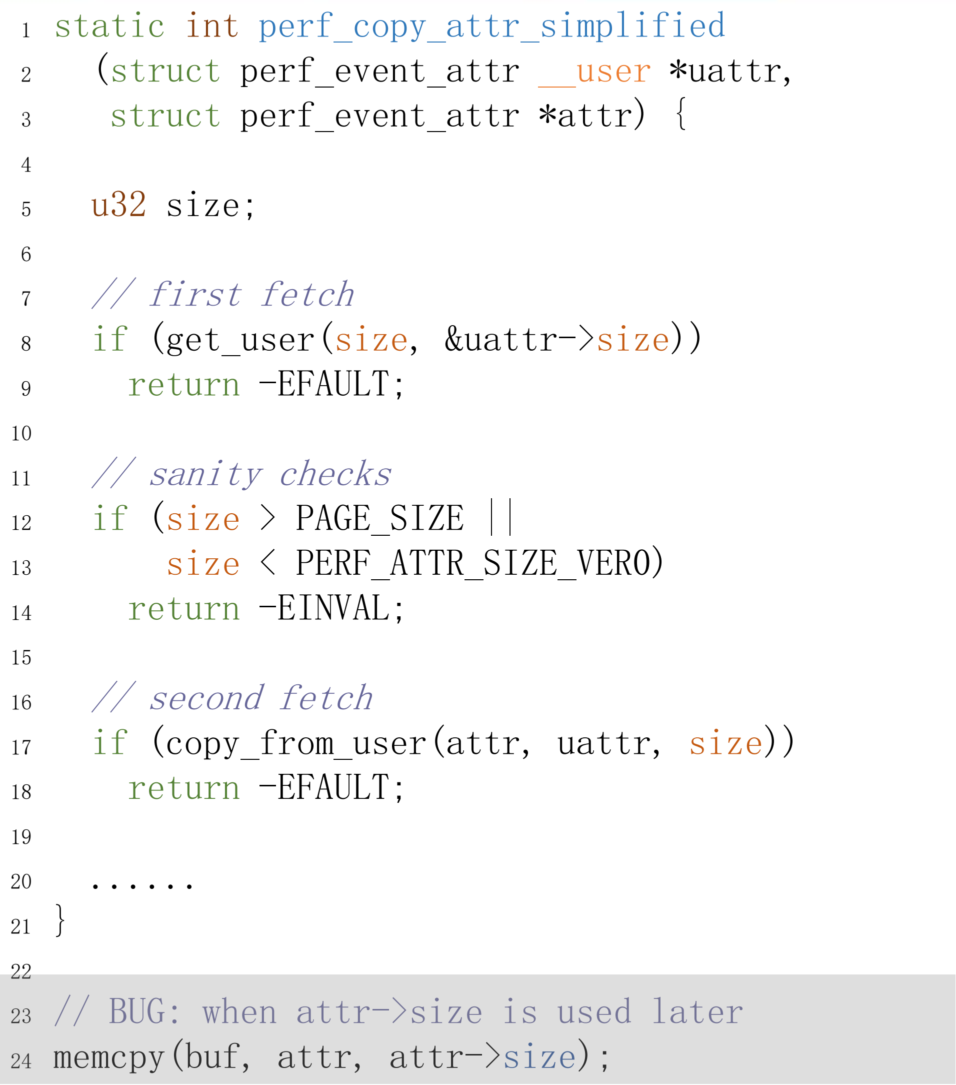
**4** bytes

30

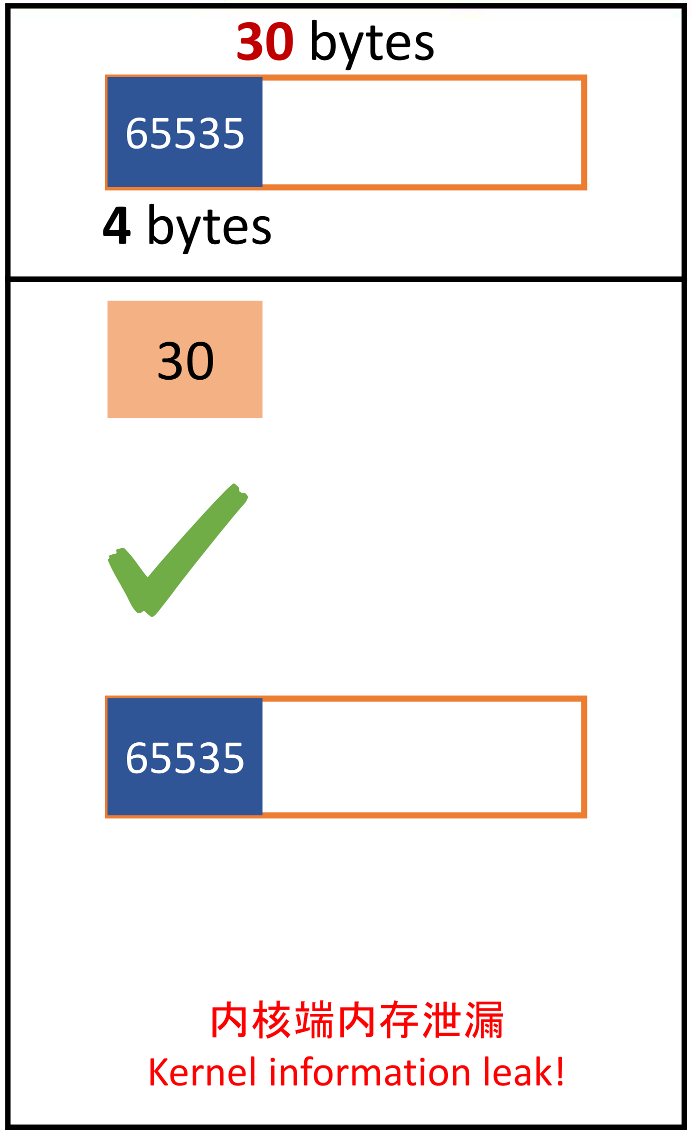
✓

65535

# 之后对‘size’的调用会导致内存泄漏 (When Exploits Happen)



```
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);
```



内核端内存泄漏
Kernel information leak!

# “二次获取”漏洞的根本原因 (Root Cause of Double-Fetch Bugs)

- 错误的认为在一个系统调用中对相同的用户层地址的访问会得到同样的结果
- (FALSE ASSUMED ATOMICITY IN SYSCALL EXECUTION)

- “二次获取”漏洞本质上是一个检查时与使用时不匹配的漏洞
- (IN ESSENCE, A TIME-OF-CHECK-TO-TIME-OF-USE (TOCTOU) BUG)

# 常见的“二次获取”情境 (Double-Fetch is Prevalent in Kernels)

- 检查整个信息的大小
- (SIZE CHECKING)

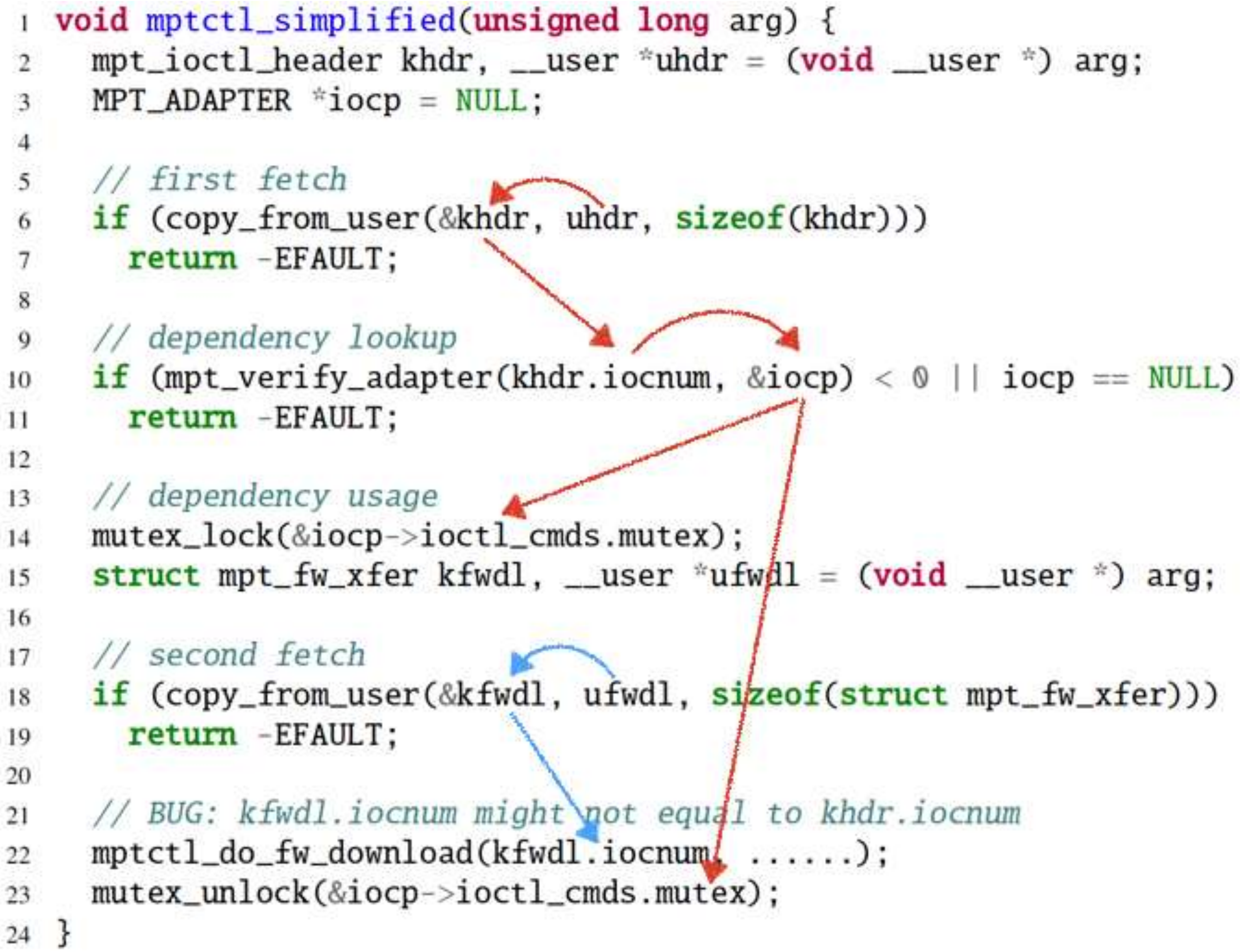
- 查找处理这个信息所依赖的对象
- (DEPENDENCY LOOKUP)

- 检查协议/签名
- (PROTOCOL/SIGNATURE CHECKING)

- 补全信息
- (INFORMATION GUESSING)

- ……

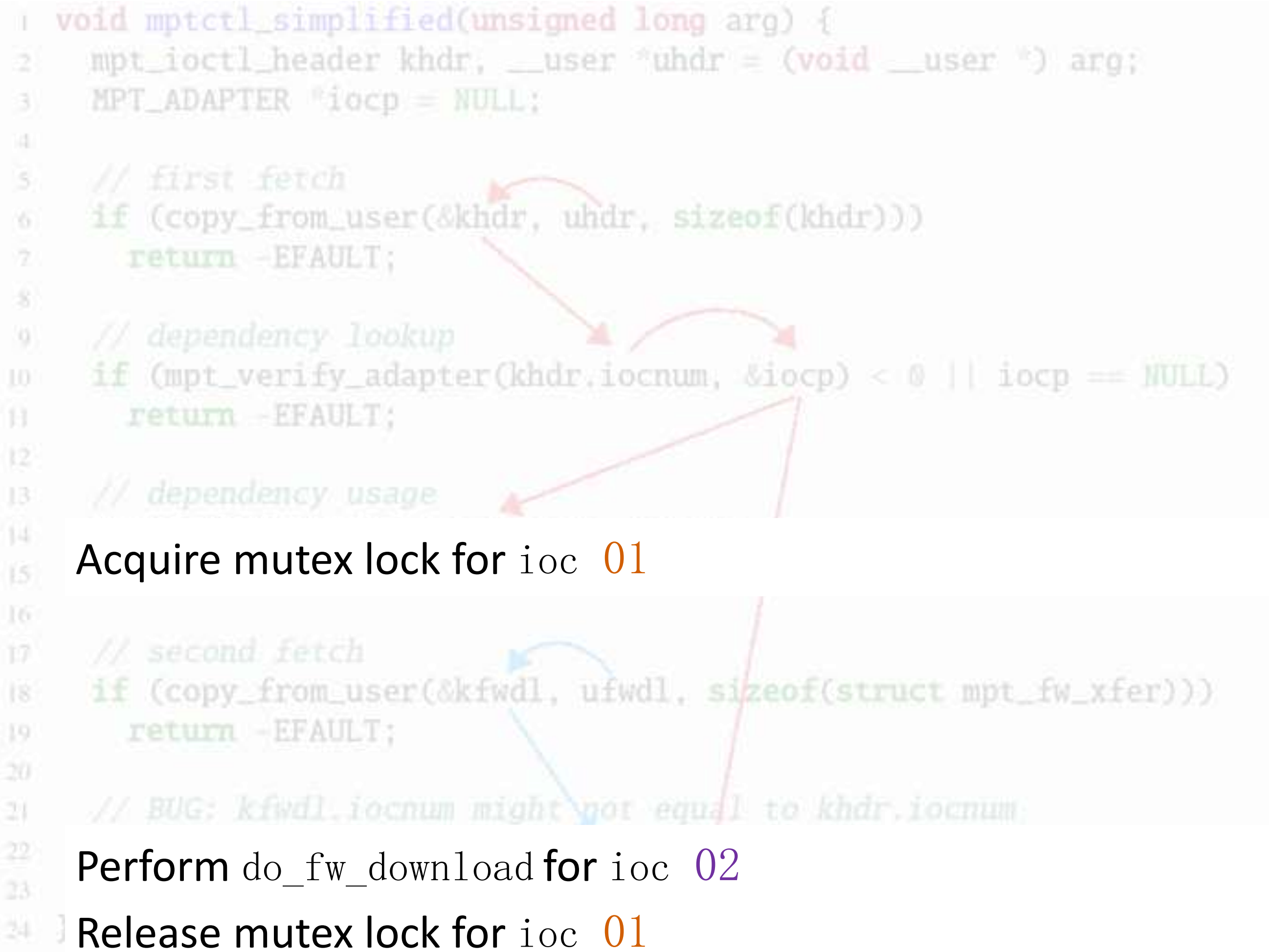
# “二次获取”案例2: 依赖查找 (Case 2: Dependency Lookup)



```
void mptctl_simplified(unsigned long arg) {
  mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
  MPT_ADAPTER *iocp = NULL;

  // first fetch
  if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
    return -EFAULT;

  // dependency lookup
  if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
    return -EFAULT;

  // dependency usage
  mutex_lock(&iocp->ioctl_cmds.mutex);
  struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;

  // second fetch
  if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
    return -EFAULT;

  // BUG: kfwdl.iocnum might not equal to khdr.iocnum
  mptctl_do_fw_download(kfwdl.iocnum, ......);
  mutex_unlock(&iocp->ioctl_cmds.mutex);
}
```

Adapted from __*mptctl_ioctl* in file *drivers/message/fusion/mptctl.c*

# “二次获取”案例2: 依赖查找 (Case 2: Dependency Lookup)



```
void mptctl_simplified(unsigned long arg) {
  mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
  MPT_ADAPTER *iocp = NULL;

  // first fetch
  if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
    return -EFAULT;

  // dependency lookup
  if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
    return -EFAULT;

  // dependency usage
```

Acquire mutex lock for `ioc` 01

```
  // second fetch
  if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
    return -EFAULT;

  // BUG: kfwdl.iocnum might not equal to khdr.iocnum
```

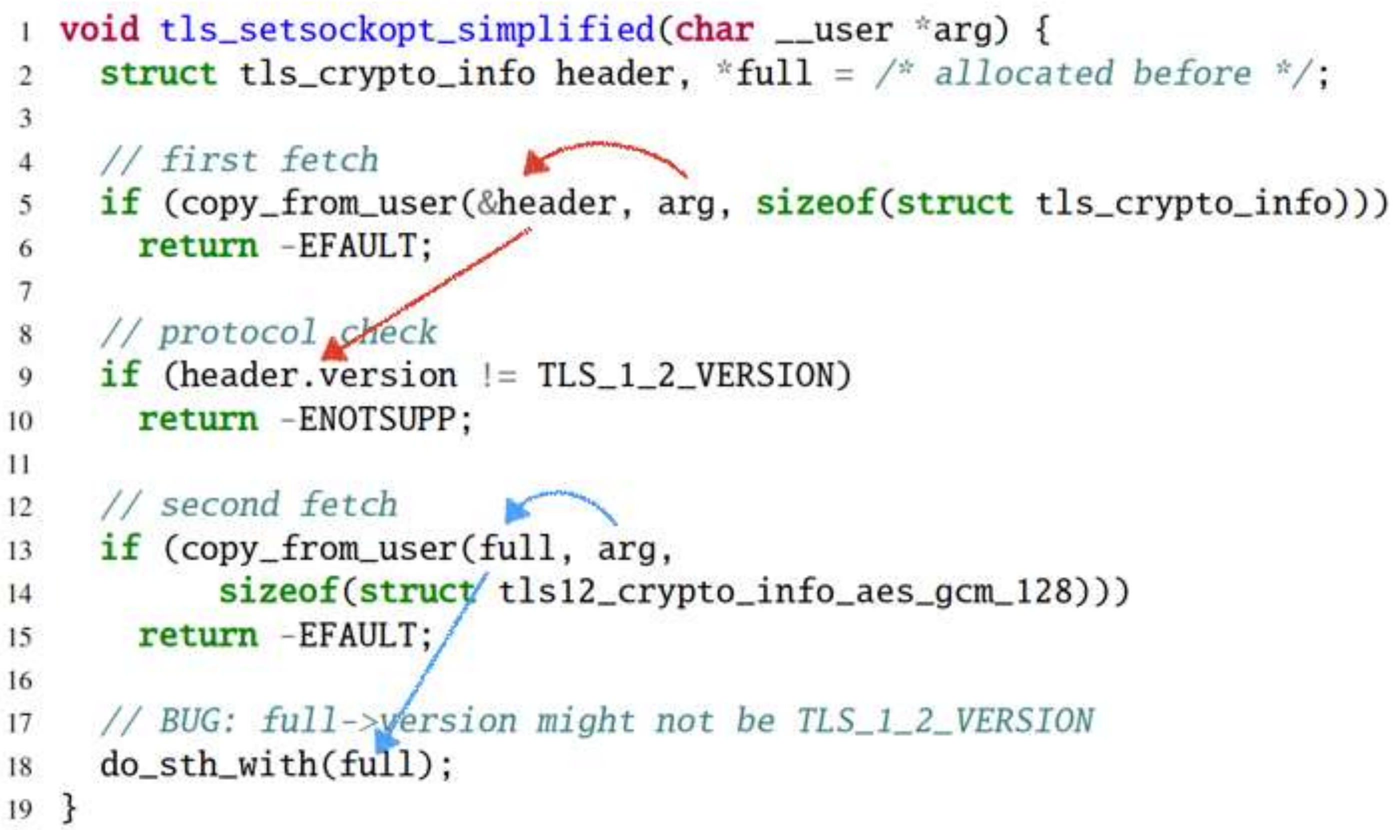
Perform `do_fw_download` for `ioc` 02

Release mutex lock for `ioc` 01

```
}
```

Adapted from *__mptctl_ioctl* in file *drivers/message/fusion/mptctl.c*

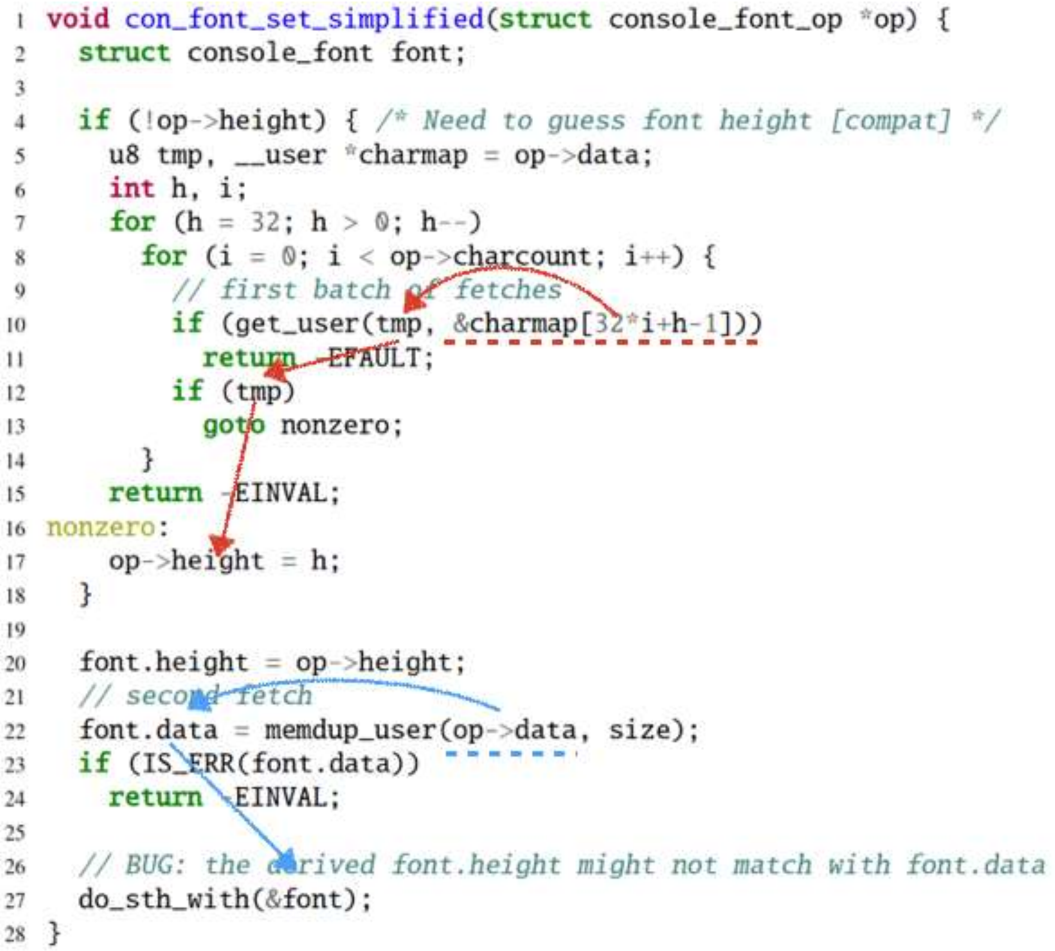
# “二次获取”案例3: 协议检查 (Case 3: Protocol/Signature Check)



```
void tls_setsockopt_simplified(char __user *arg) {
  struct tls_crypto_info header, *full = /* allocated before */;

  // first fetch
  if (copy_from_user(&header, arg, sizeof(struct tls_crypto_info)))
    return -EFAULT;

  // protocol check
  if (header.version != TLS_1_2_VERSION)
    return -ENOTSUPP;

  // second fetch
  if (copy_from_user(full, arg,
        sizeof(struct tls12_crypto_info_aes_gcm_128)))
    return -EFAULT;

  // BUG: full->version might not be TLS_1_2_VERSION
  do_sth_with(full);
}
```

Adapted from *do_tls_setsockopt_tx* in file *net/tls/tls_main.c*

# “二次获取”案例4: 信息补全 (Case 4: Information Guessing)



```c
void con_font_set_simplified(struct console_font_op *op) {
  struct console_font font;

  if (!op->height) { /* Need to guess font height [compat] */
    u8 tmp, __user *charmap = op->data;
    int h, i;
    for (h = 32; h > 0; h--)
      for (i = 0; i < op->charcount; i++) {
        // first batch of fetches
        if (get_user(tmp, &charmap[32*i+h-1]))
          return -EFAULT;
        if (tmp)
          goto nonzero;
      }
    return -EINVAL;
nonzero:
    op->height = h;
  }

  font.height = op->height;
  // second fetch
  font.data = memdup_user(op->data, size);
  if (IS_ERR(font.data))
    return -EINVAL;

  // BUG: the derived font.height might not match with font.data
  do_sth_with(&font);
}
```

Adapted from *con_font_set* in file *drivers/tty/vt/vt.c*

# “二次获取” 相关的研究 (Prior Works)

| | Bochspwn<br>(BlackHat’13) | DECAF<br>(arXiv’17) | Pengfei et. al.,<br>(Security’17) | |
|---|---|---|---|---|
| 内核<br>(Kernel) | Windows | Linux | Linux, FreeBSD | |
| 分析模式<br>(Analysis) | 动态分析<br>(Dynamic) | 动态分析<br>(Dynamic) | 静态分析<br>(Static) | |
| 主要方法<br>(Method) | 虚拟机检查<br>(VMI) | 内核模糊测试<br>(Kernel fuzzing) | 源代码匹配<br>(Lexical Code Matching) | |
| “二次引用”模版<br>(Patten) | 短时间内两次访问相同的内存地址<br>(Memory access timing) | 基于缓存的侧信道<br>(Cache side channel) | 基于信息大小检查的源代码模式<br>(Size checking) | |
| 代码覆盖率<br>(Code Coverage) | 低<br>(Low) | 低<br>(Low) | 高<br>(High) | |
| 手动辨识<br>(Manual Effort) | 需要手动区分正常的”二次引用”与”二次引用”漏洞<br>(Manual checking required to differentiate double-fetch cases and bugs) | | | |

# “二次获取”相关的研究 (Prior Works)

| | Bochspwn<br>(BlackHat’13) | DECAF<br>(arXiv’17) | Pengfei et. al.,<br>(Security’17) | **Deadline**<br>**(IEEE SP’18)** |
|---|---|---|---|---|
| 内核<br>(Kernel) | Windows | Linux | Linux, FreeBSD | Linux, FreeBSD |
| 分析模式<br>(Analysis) | 动态分析<br>(Dynamic) | 动态分析<br>(Dynamic) | 静态分析<br>(Static) | 静态分析<br>(Static) |
| 主要方法<br>(Method) | 虚拟机检查<br>(VMI) | 内核模糊测试<br>(Kernel fuzzing) | 源代码匹配<br>(Lexical Code Matching) | 符号执行<br>(Symbolic Execution) |
| “二次引用”模版<br>(Patten) | 短时间内两次访问相同的内存地址<br>(Memory access timing) | 基于缓存的侧信道<br>(Cache side channel) | 基于信息大小检查的源代码模式<br>(Size checking) | 基于”二次获取”漏洞的严谨定义<br>(Formal definitions) |
| 代码覆盖率<br>(Code Coverage) | 低<br>(Low) | 低<br>(Low) | 高<br>(High) | 高<br>(High) |
| 手动辨识<br>(Manual Effort) | 需要手动区分正常的”二次引用”与”二次引用”漏洞<br>(Manual checking required to differentiate double-fetch cases and bugs) | | | 无需手动区分<br>(No manual checking) |

# “二次获取”漏洞的定义 (Double-Fetch Bugs: A Formal Definition)

**获取**：每一次获取可以表示为一个 $(A, S)$ 对.

$A$ – 获取的起始地址

$S$ – 复制至内核层的信息的大小

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

$A$ - the starting address of the fetch,

$S$ - the size of memory copied into kernel.

**有重叠的获取**：即两次获取， $(A_0, S_0)$ 与 $(A_1, S_1)$，

- 满足条件 $A_0 \leqslant A_1 < A_0 + S_0 \ \| \ A_1 \leqslant A_0 < A_1 + S_1$
- 重叠的内存地址会被标记为 $(A_{01}, S_{01})$
- 第一次复制进来的内容会被标记为 $(A_{01}, S_{01}, 0)$
- 第二次复制进来的内容会被标记为 $(A_{01}, S_{01}, 1)$

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

- $A_0 \leq A_1 < A_0 + S_0 \ \| \ A_1 \leq A_0 < A_1 + S_1$
- 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 $(A_{01}, S_{01}, 1)$

“二次获取”漏洞的直观（但不严谨）定义： $(A_{01}, S_{01}, 0) \mathrel{!=} (A_{01}, S_{01}, 1)$

An intuitive (but imprecise) definition of “double-fetch”: $(A_{01}, S_{01}, 0) \mathrel{!=} (A_{01}, S_{01}, 1)$

# “二次获取”漏洞的定义 (Double-Fetch Bugs: A Formal Definition)

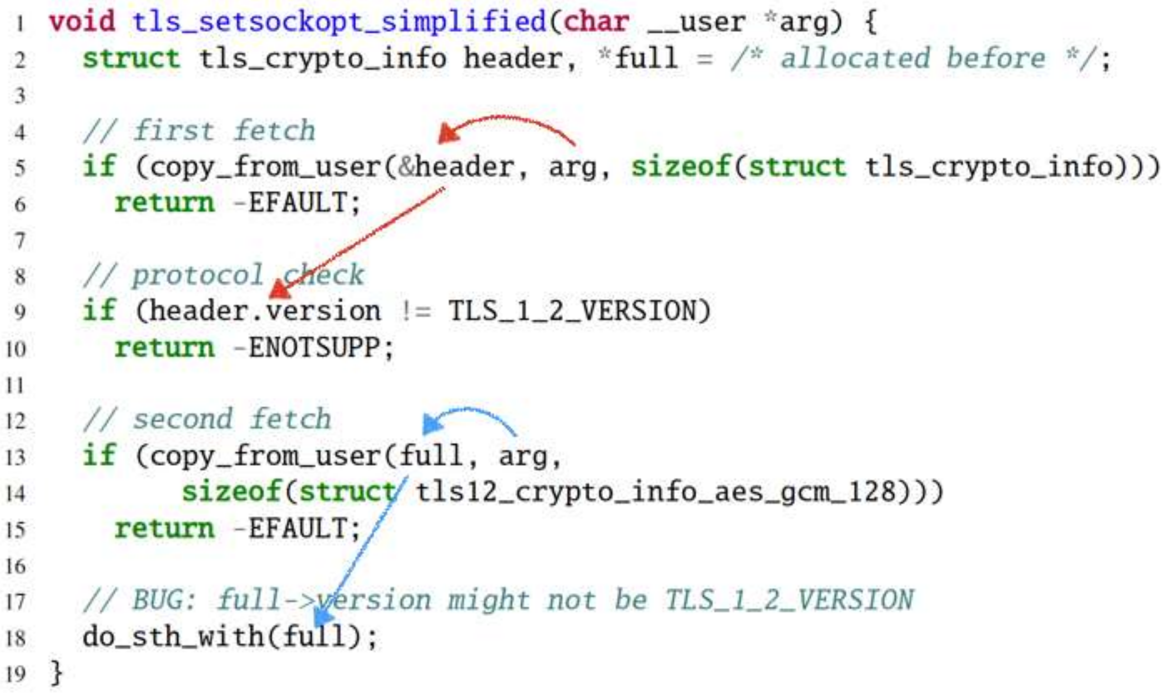
**控制流依赖**：变量 $V \in (A_{01}, S_{01})$ 且 $V$ 必须满足某些条件才能使得第二次获取发生

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

# “二次获取”漏洞的定义 (Double-Fetch Bugs: A Formal Definition)

**控制流依赖**：变量 $V \in (A_{01}, S_{01})$ 且 $V$ 必须满足某些条件才能使得第二次获取发生

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



```
void tls_setsockopt_simplified(char __user *arg) {
  struct tls_crypto_info header, *full = /* allocated before */;

  // first fetch
  if (copy_from_user(&header, arg, sizeof(struct tls_crypto_info)))
    return -EFAULT;

  // protocol check
  if (header.version != TLS_1_2_VERSION)
    return -ENOTSUPP;

  // second fetch
  if (copy_from_user(full, arg,
        sizeof(struct tls12_crypto_info_aes_gcm_128)))
    return -EFAULT;

  // BUG: full->version might not be TLS_1_2_VERSION
  do_sth_with(full);
}
```

重叠的变量 $V$

（Overlapped variable $V$）:

header.version

$V$必须满足的条件

(The constraint $V$ must satisfy):

header.version == TLS_1_2_VERSION

第二次获取后 $V$期待的值

(The expectation for $V$ after second fetch)

full->version == TLS_1_2_VERSION

# “二次获取”漏洞的定义 (Double-Fetch Bugs: A Formal Definition)

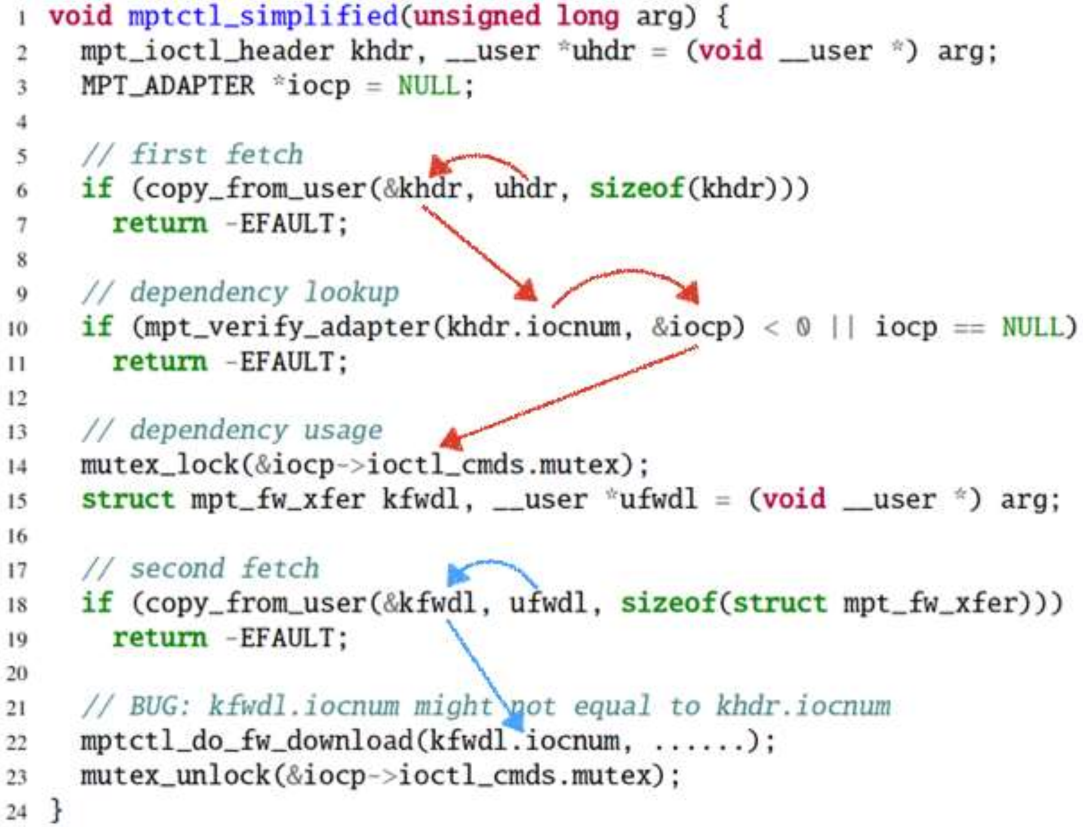
**数据流依赖**：变量 $V \in (A_{01}, S_{01})$ 且 $V$ 在第二次获取之前（或第二次获取中）被用于其他执行语句中，例如函数调用，变量的推倒等

**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).

# “二次获取”漏洞的定义 (Double-Fetch Bugs: A Formal Definition)

**数据流依赖**：变量 $V \in (A_{01}, S_{01})$ 且 $V$ 在第二次获取之前被用于其他语句中，例如函数调用等

**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).



```
void mptctl_simplified(unsigned long arg) {
  mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
  MPT_ADAPTER *iocp = NULL;

  // first fetch
  if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
    return -EFAULT;

  // dependency lookup
  if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
    return -EFAULT;

  // dependency usage
  mutex_lock(&iocp->ioctl_cmds.mutex);
  struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;

  // second fetch
  if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
    return -EFAULT;

  // BUG: kfwdl.iocnum might not equal to khdr.iocnum
  mptctl_do_fw_download(kfwdl.iocnum, ......);
  mutex_unlock(&iocp->ioctl_cmds.mutex);
}
```

重叠的变量 $V$

（Overlapped variable $V$）:

khdr.iocnum

$V$被用到的语句

(The statement where $V$ is consumed):

mpt_verify_adapter(khdr.iocnum, &iocp)

第二次获取后$V$期待的值

(The expectation for $V$ after second fetch)

kfwdl.iocnum == khdr.iocnum

# “二次获取”漏洞的定义 (Double-Fetch Bugs: A Formal Definition)

1. 两次从用户层内存空间的获取有**重叠的区域。**

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

2. 在重叠的区域里面有一个变量使得这两次获取之间可以建立某种联系。这种联系即可以是**控制流依赖**也可以是**数据流依赖**，还可能**两者都有。**

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

3. 在第二次获取之后无法证明这个变量没有变化。

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

# 如何根据定义来寻找漏洞？ (How to Find Double-Fetch Bugs?)

1. 寻找尽可能多的“获取”对，并对每一对建立程序路径

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

2. 符号性的执行每一个程序路径并且由此来决定这两次“获取”是不是一个真正的漏洞

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

# 寻找“获取”对 (Fetch Pair Collection)

**目标：**静态的枚举所有在执行一个系统调用时可能的获取对

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

# 寻找“获取”对 (Fetch Pair Collection)

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

从某一个获取开始
(Start from a fetch)

# 寻找“获取”对 (Fetch Pair Collection)

依次遍历之前的语句
（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;
   …
}
```

# 寻找“获取”对 (Fetch Pair Collection)

[第一种情况]
找到另一个“获取”
[Case 1]
Found another fetch

```
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;
  …
}
```

# 寻找“获取”对 (Fetch Pair Collection)

[第二种情况]
找到另一个包含“获取”的函数
[Case 2]
ound a fetch-involved function

```
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;
   …
}
```

# 寻找“获取”对 (Fetch Pair Collection)

[第三种情况]
没有找到跟“获取”相关的语句
[Case 3]
o fetch-related instruction found

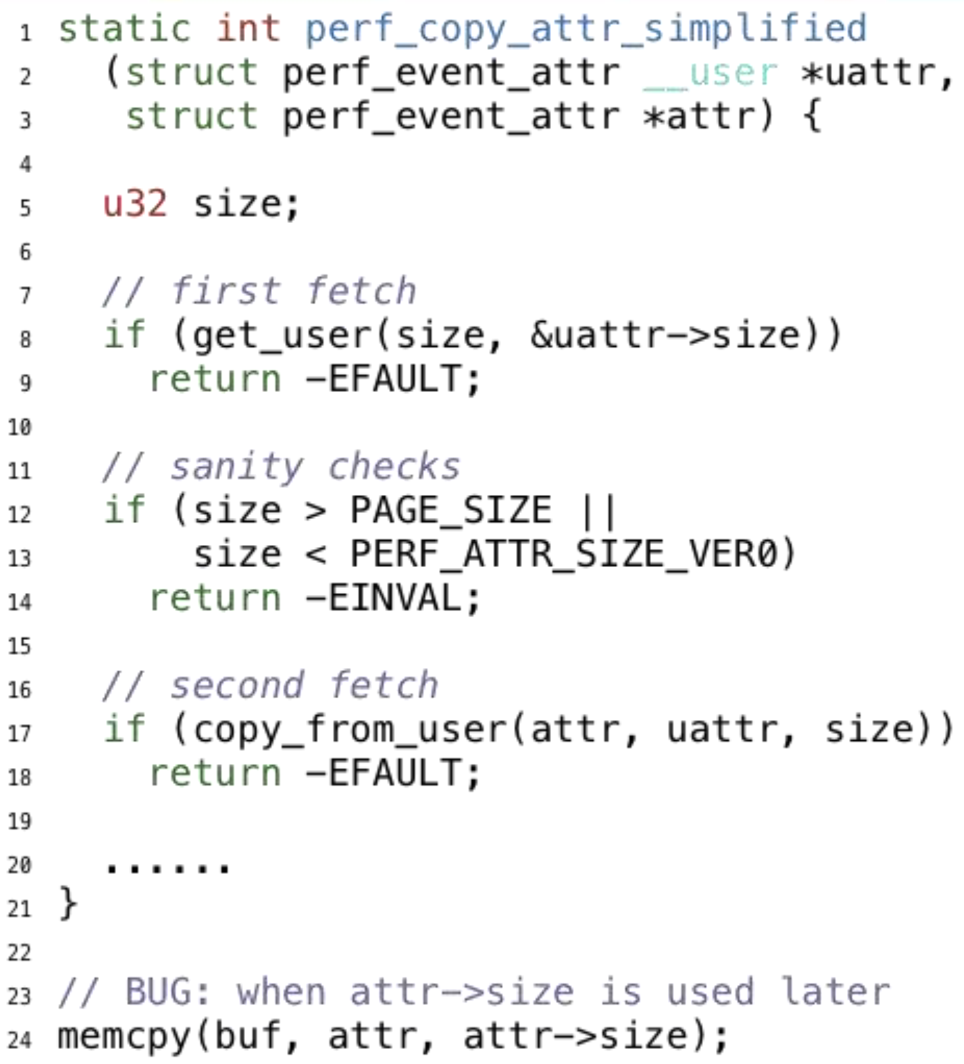
```
static void enclosing_function(
   struct msg_hdr __user *uptr,
   struct msg_full *kptr
) {
   …
   …
   …
   …
   …
   …
   …
   if (copy_from_user(kptr, uptr, size))
      return -EFAULT;
   …
}
```

# 符号执行 (Symbolic Checking)

**目标：**符号性的执行所有找到的链接两次获取的程序路径并根据定义来判断这两次获取是否构成二次获取漏洞

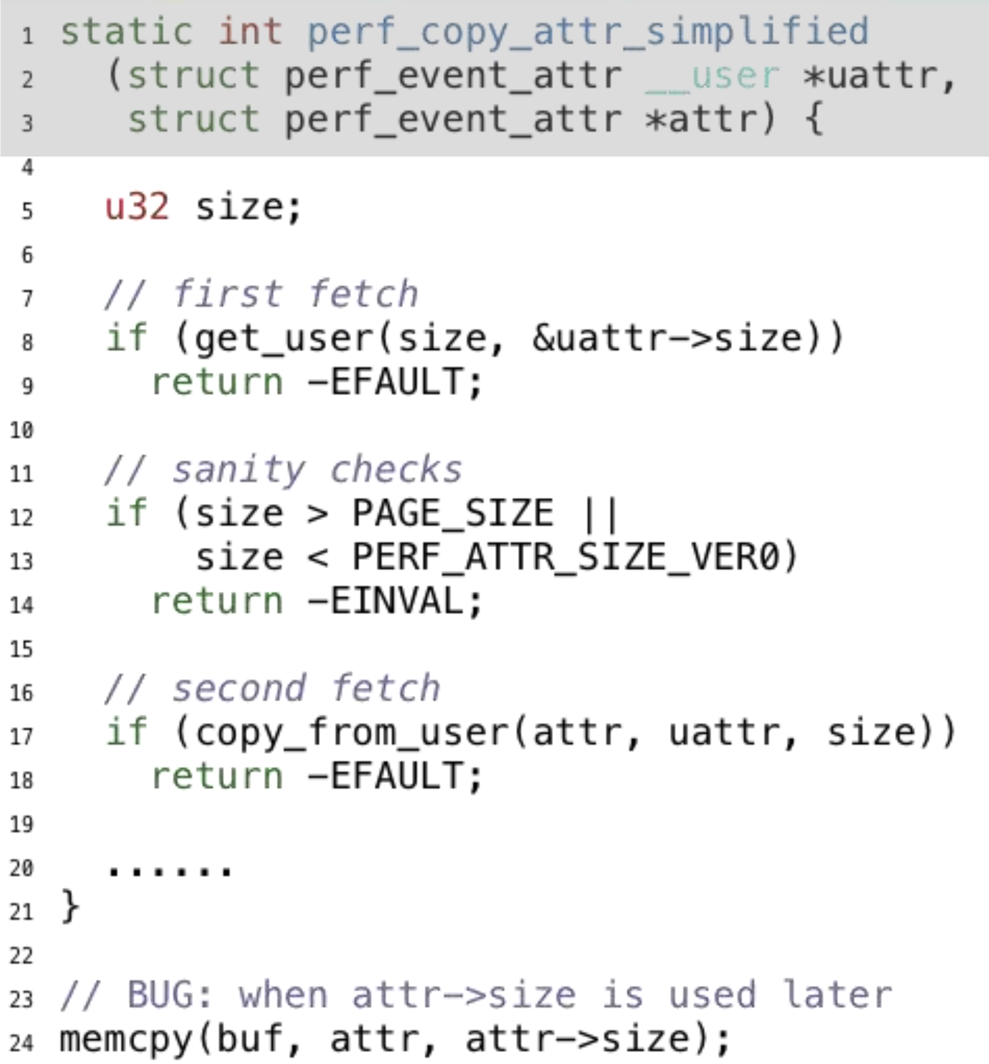
**Goal**: Symbolically execute the code path that connects two fetches and determine whether the two fetches satisfy all the criteria set in formal definition of double-fetch bug

# 符号执行 (Symbolic Checking)

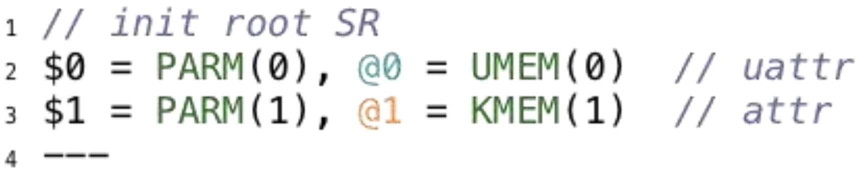


```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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);
```

# 符号执行 (Symbolic Checking)

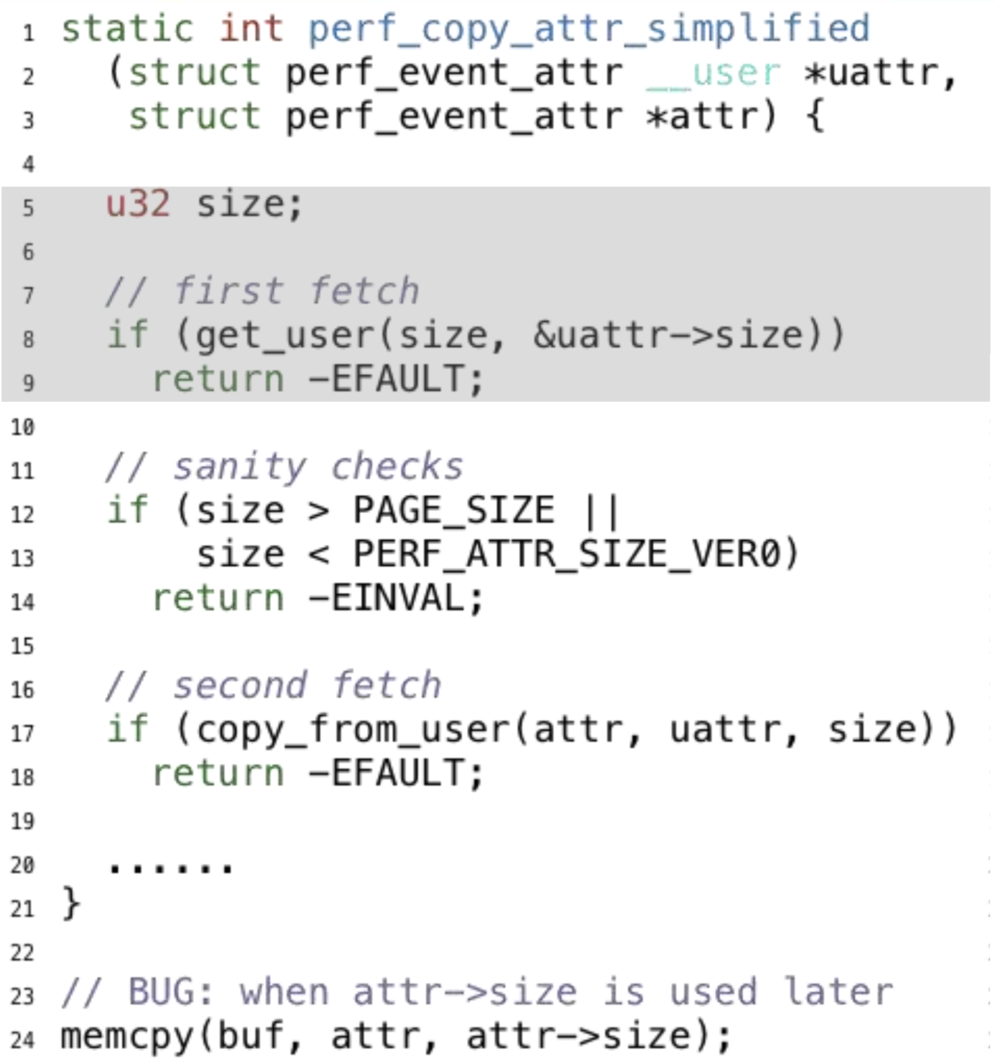


```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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);
```

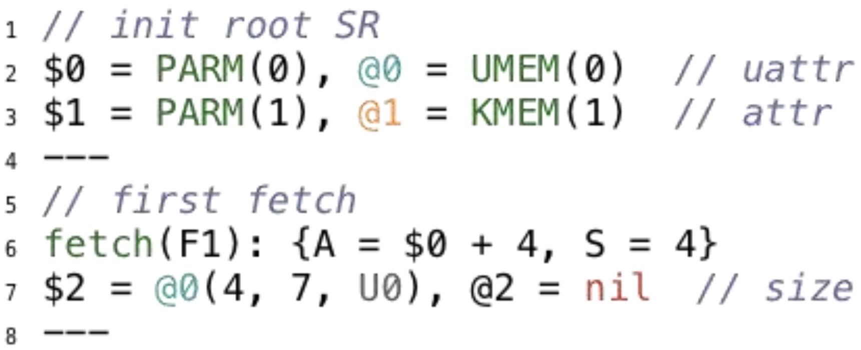


```
// init root SR
$0 = PARM(0), @0 = UMEM(0)  // uattr
$1 = PARM(1), @1 = KMEM(1)  // attr
---
```

# 符号执行 (Symbolic Checking)

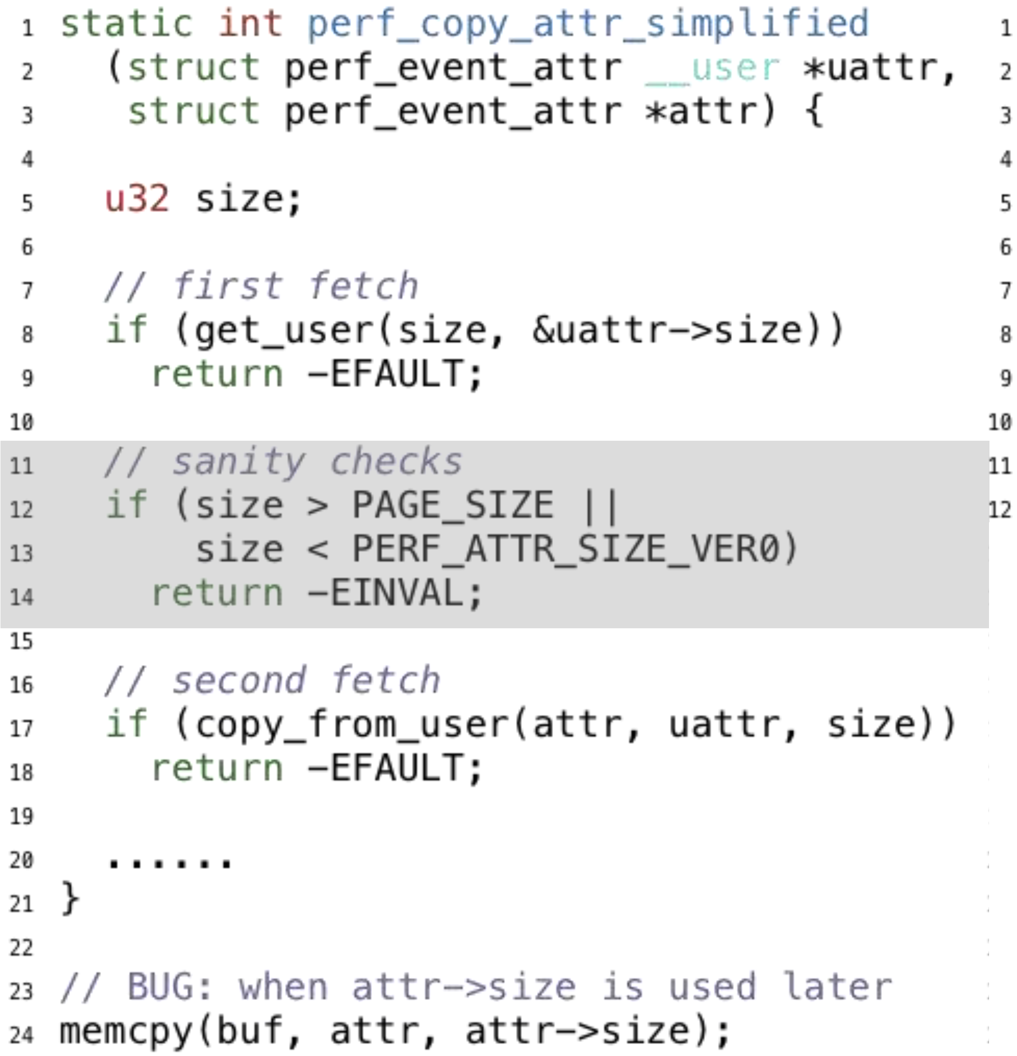


```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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);
```

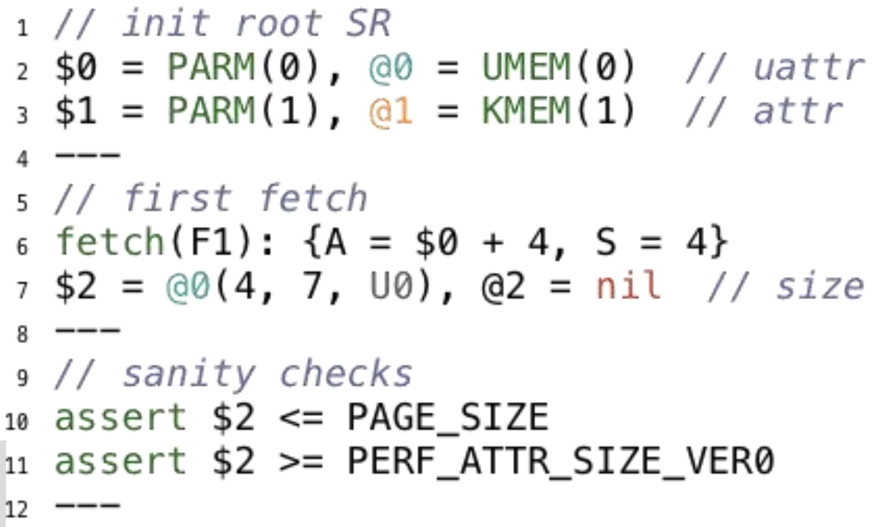


```
// init root SR
$0 = PARM(0), @0 = UMEM(0)   // uattr
$1 = PARM(1), @1 = KMEM(1)   // attr
---
// first fetch
fetch(F1): {A = $0 + 4, S = 4}
$2 = @0(4, 7, U0), @2 = nil  // size
---
```

# 符号执行 (Symbolic Checking)

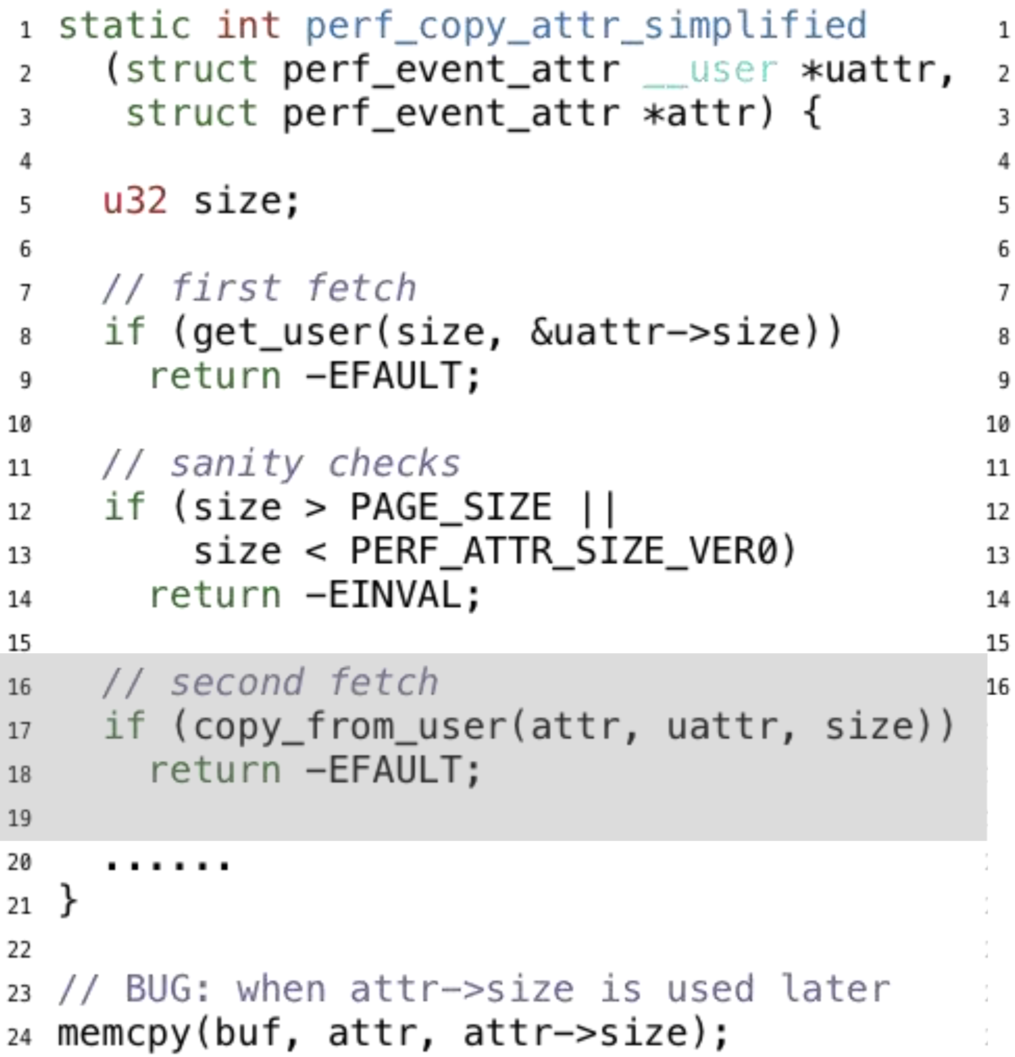


```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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);
```

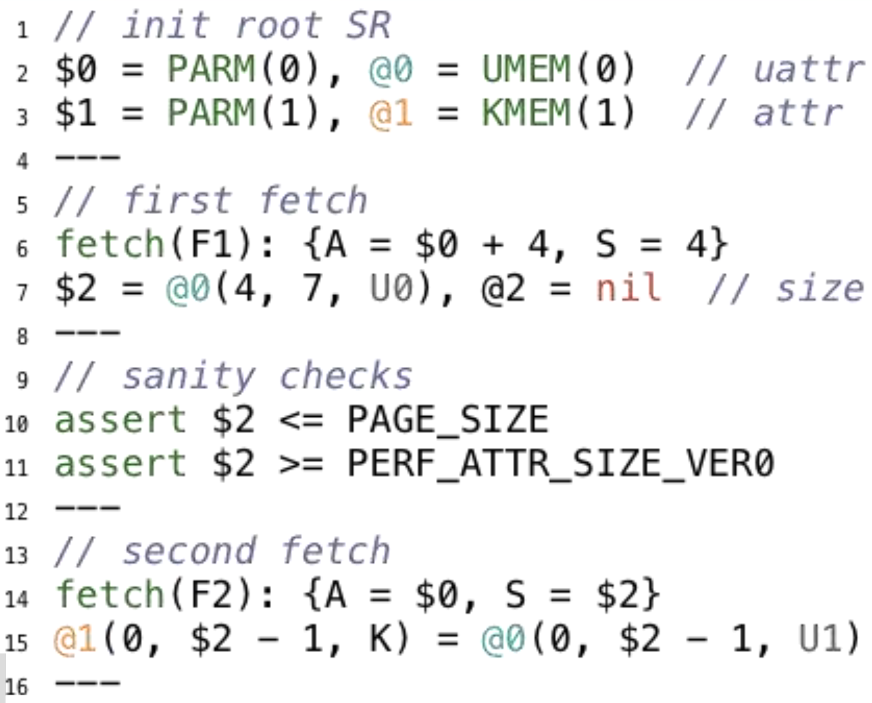


```
// init root SR
$0 = PARM(0), @0 = UMEM(0)  // uattr
$1 = PARM(1), @1 = KMEM(1)  // attr
---
// first fetch
fetch(F1): {A = $0 + 4, S = 4}
$2 = @0(4, 7, U0), @2 = nil  // size
---
// sanity checks
assert $2 <= PAGE_SIZE
assert $2 >= PERF_ATTR_SIZE_VER0
---
```

# 符号执行 (Symbolic Checking)

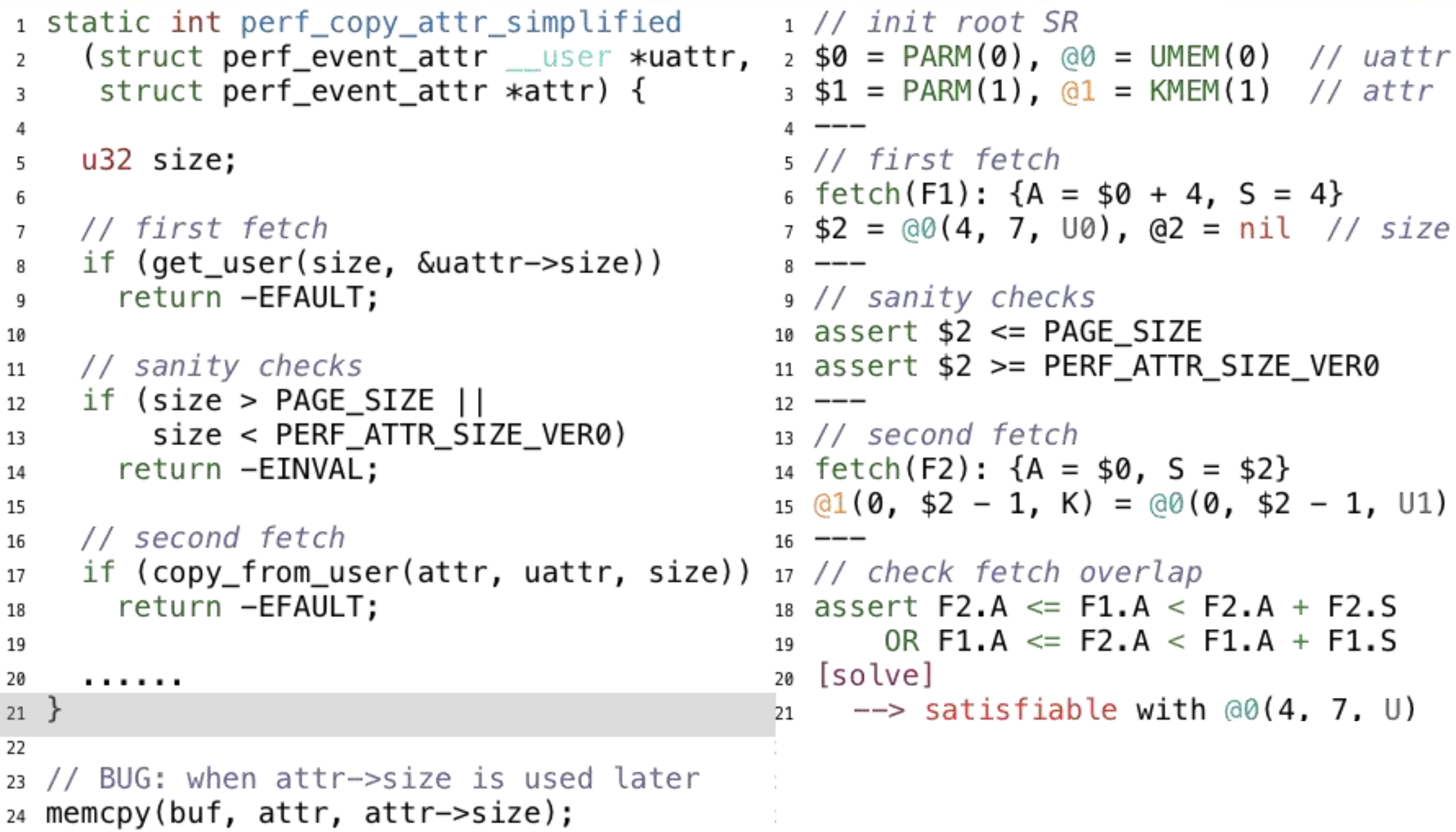


```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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)
---
```

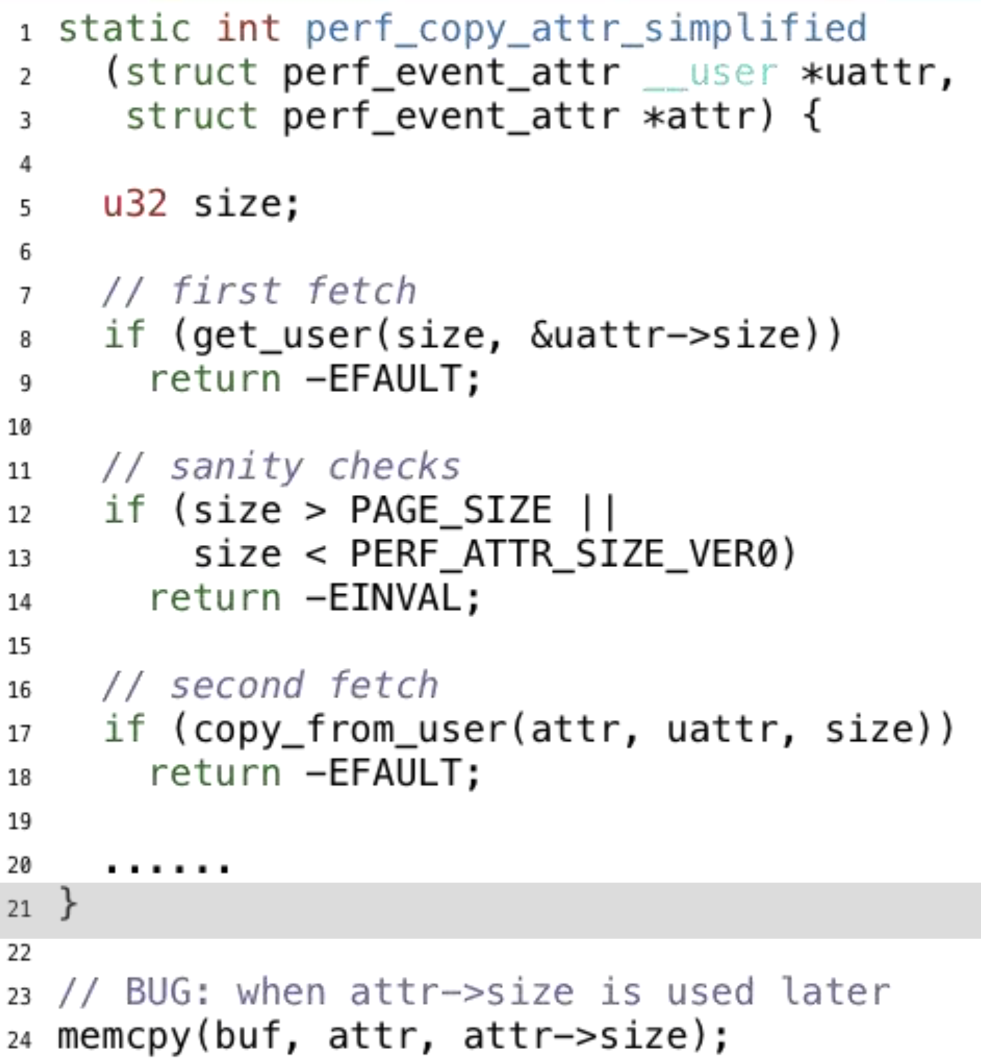
# 符号执行 (Symbolic Checking)



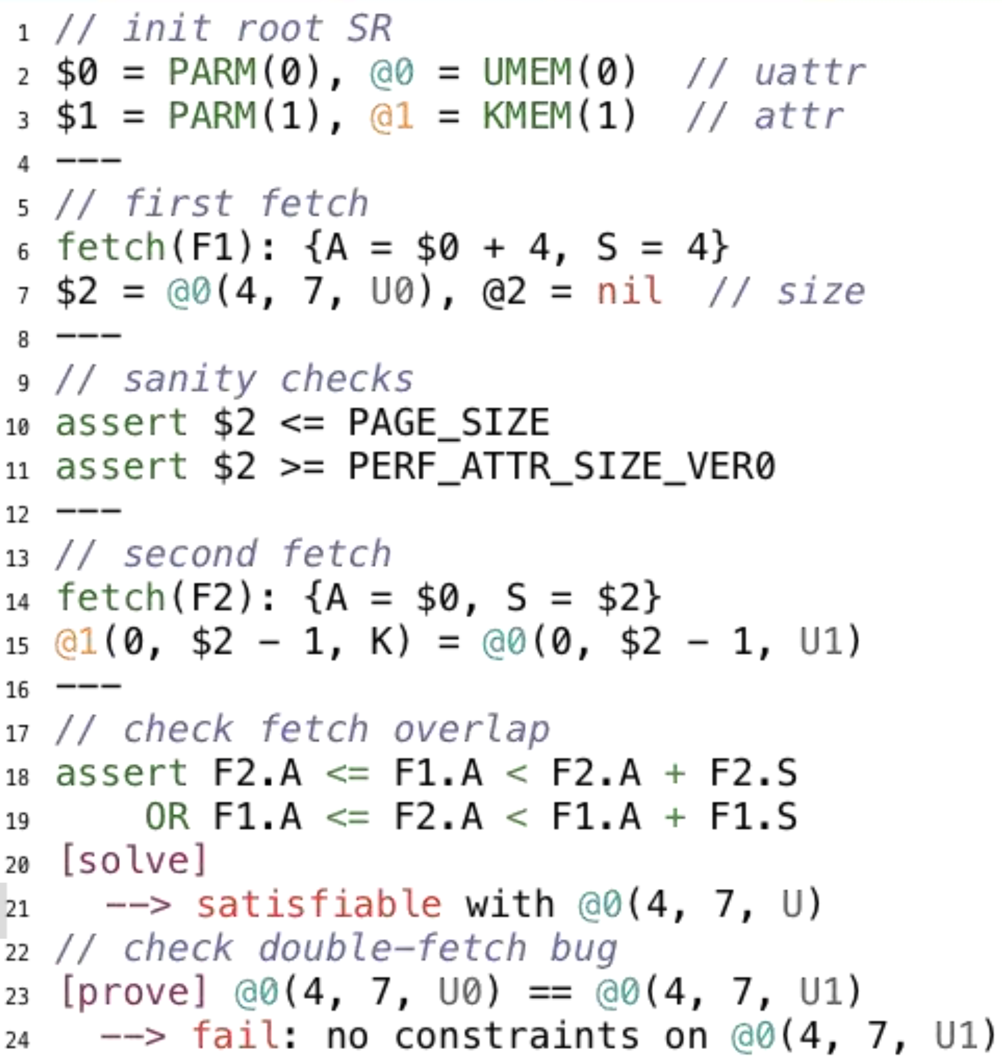
```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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)
```

# 符号执行 (Symbolic Checking)

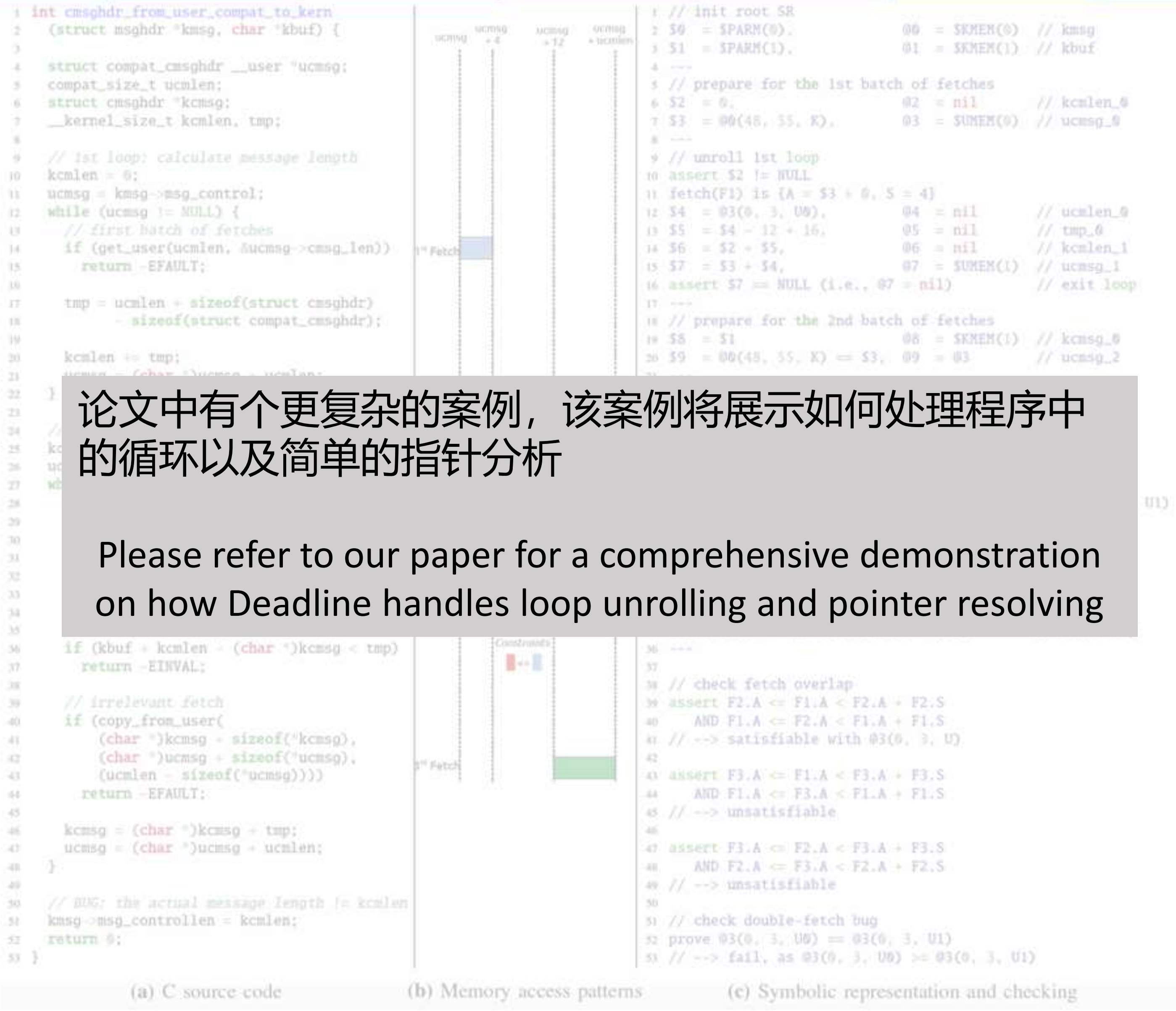


```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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)
// check double-fetch bug
[prove] @0(4, 7, U0) == @0(4, 7, U1)
  --> fail: no constraints on @0(4, 7, U1)
```

# 符号执行 (Symbolic Checking)



```
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;
  }
```

```
    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
```

```
---

// 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

论文中有个更复杂的案例，该案例将展示如何处理程序中的循环以及简单的指针分析

Please refer to our paper for a comprehensive demonstration on how Deadline handles loop unrolling and pointer resolving

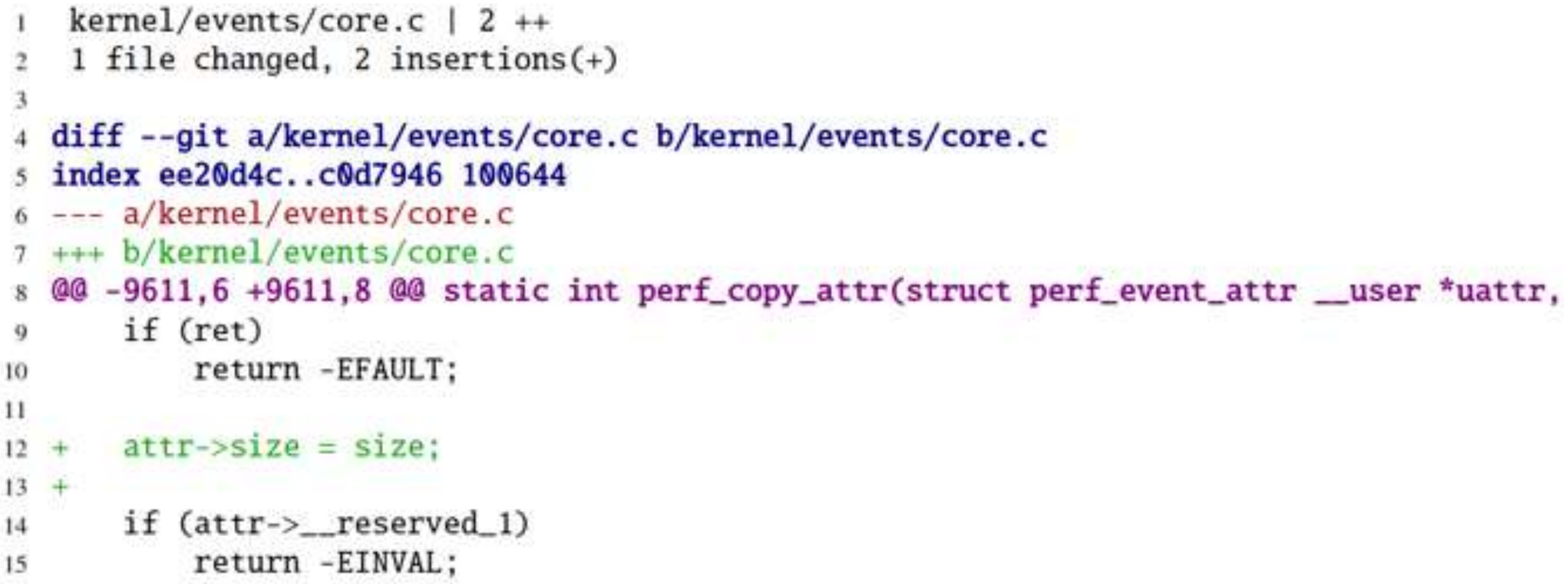
# 找到的漏洞 (Findings)

1. 一共找到24个漏洞
   24 Bugs found in total.
   - 其中23个在LINUX内核，1个在FREEBSD内核
     23 bugs in Linux kernel and 1 in FreeBSD kernel
2. 我们为10个漏洞提供了补丁并且已经应用于代码中
   10 bugs have been patched with the fix we provide
3. 5个漏洞被维护者认可，但是相关补丁还没有发布
   5 bugs are acknowledged, we are still working on the fix
4. 7个漏洞还在审核之中
   7 bugs are pending for review
5. 2个漏洞被标记为“不处理”
   2 bugs are marked as “won’t fix”

# “二次获取”漏洞的补丁模式 (Patching Double-Fetch Bugs)

1. 过载第二次获取的内容

   Override the second fetch

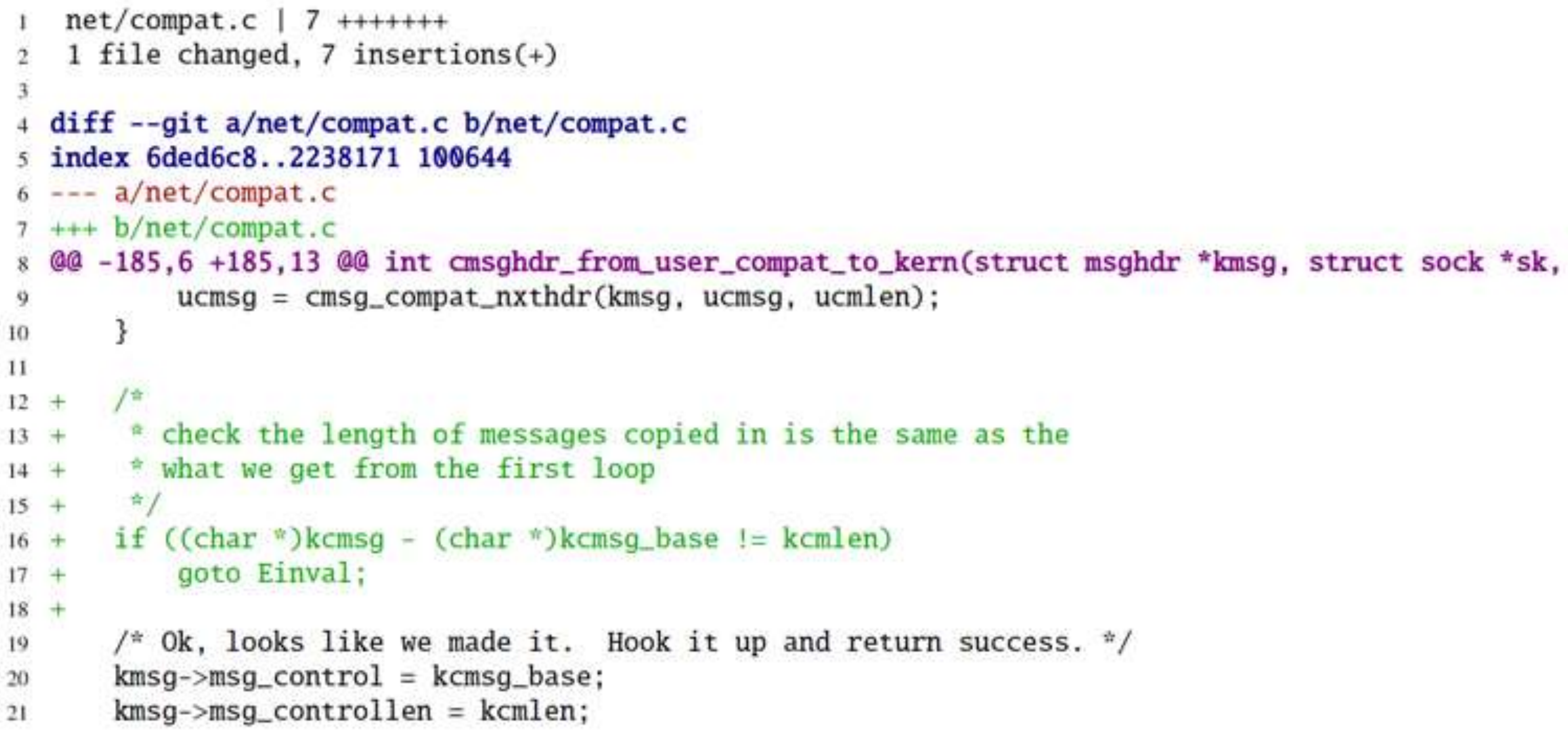


```
 kernel/events/core.c | 2 ++
 1 file changed, 2 insertions(+)

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;
```

# “二次获取”漏洞的补丁模式 (Patching Double-Fetch Bugs)

2. 检查两次获取的内容是不是一致

Abort on change detected

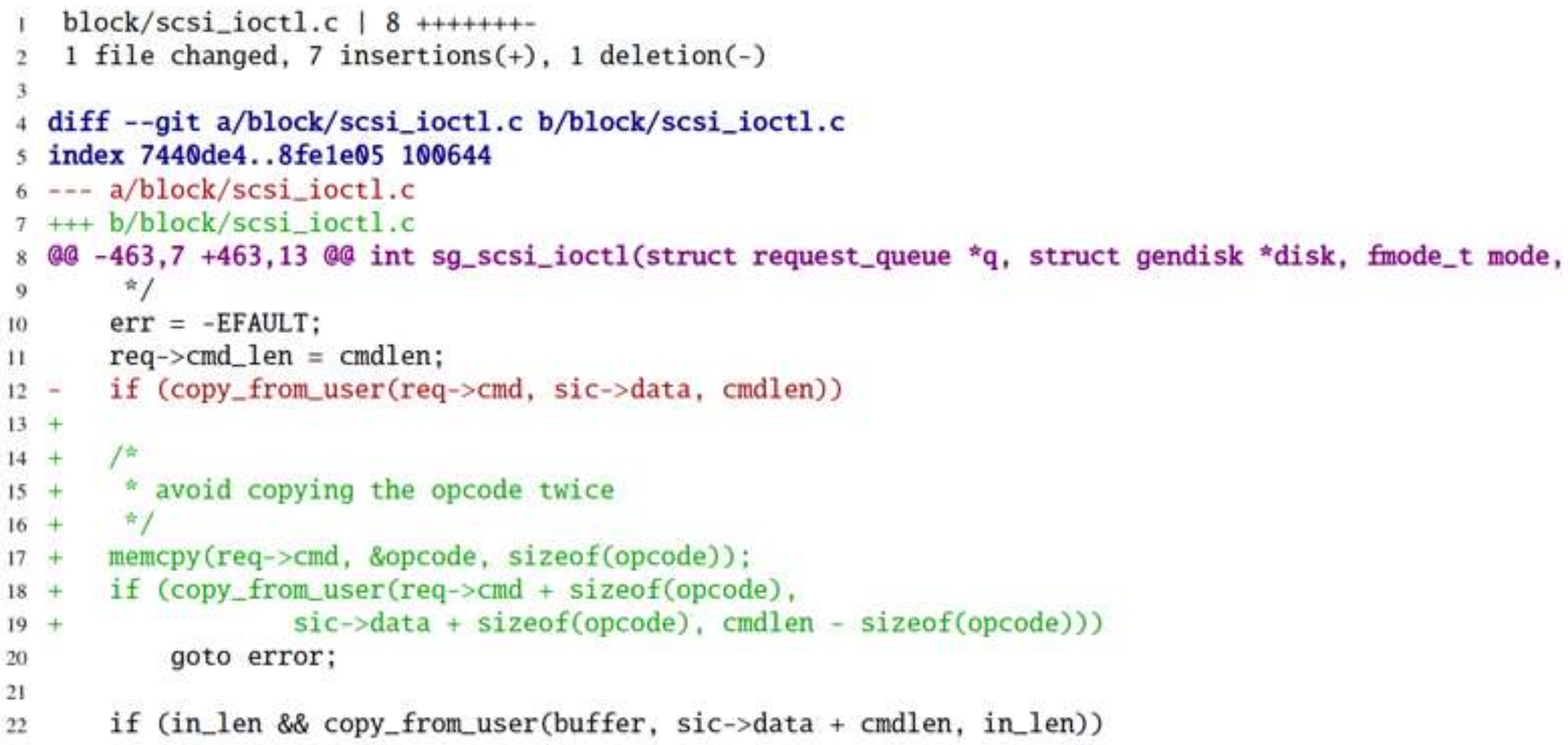


```
  net/compat.c | 7 +++++++
  1 file changed, 7 insertions(+)

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;
```

# “二次获取”漏洞的补丁模式 (Patching Double-Fetch Bugs)

3. 将两次获取重构成不重叠的两次获取

Refactor overlapped copies into incremental copies.

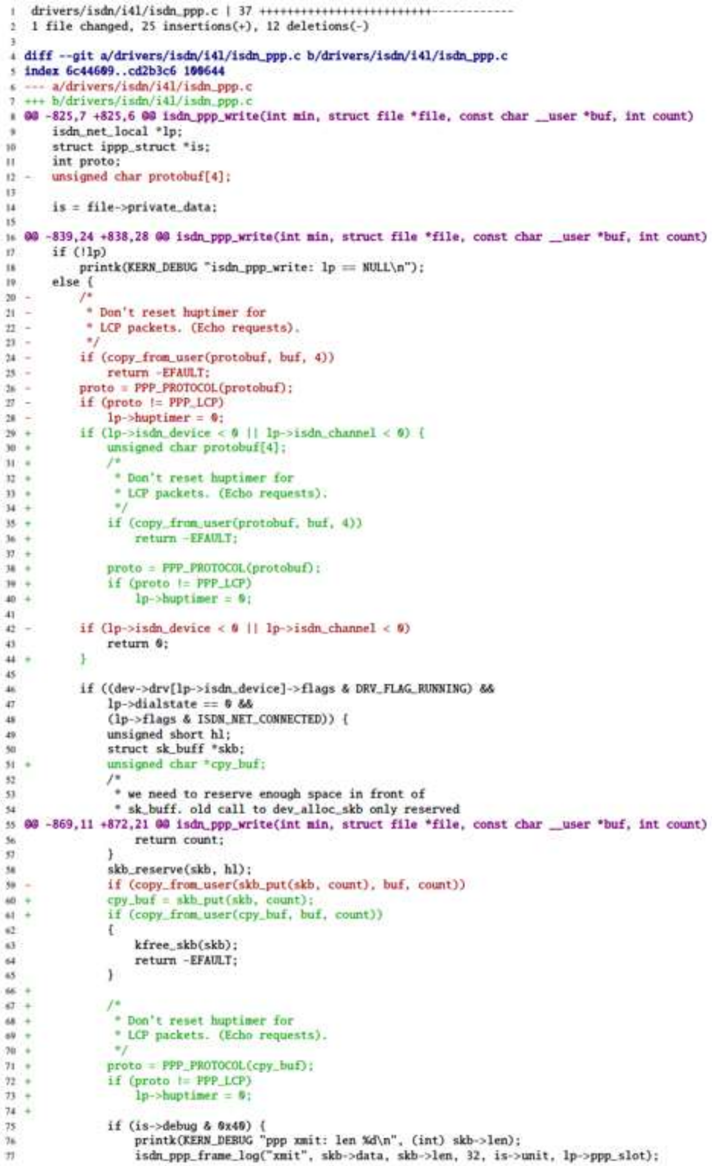


```
 block/scsi_ioctl.c | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

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))
```

# “二次获取”漏洞的补丁模式 (Patching Double-Fetch Bugs)

4. 将两次获取重构成单次获取

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);
```

# “二次获取”漏洞的一般性补丁 (Generic Patch for Double-Fetch Bugs)

并非所有“二次获取”漏洞都可以有一般性补丁或者都可以用以上几种模式来修补。某些漏洞需要复杂的代码重构或者重新设计用于信息传递的数据结构，这些都需要大量的工作。

Unfortunately, not all double-fetch bugs can be patched with these patterns. Some requires heavy refactoring of existing codebase or re-designing of structs, which requires substantial manual effort.

最近我们注意到了“DECAF”这个工作，似乎提供了一个很有价值也很有前景的思路：利用INTEL CPU的TSX技术来保证在一个系统调用中对用户层内存的访问是原子性的。

Recently, DECAF has provided a promising solution in using TSX-based techniques to ensure user space memory access **automaticity** in syscall execution.

# 结语 (Conclusion)

有一个精确严谨的定义对寻找逻辑漏洞有重要意义，这样一个定义可以帮助排除误判，更精确的寻找漏洞

Detecting double-fetch bugs without a precise and formal definition has led to many false alerts and tremendous manual effort.

我们的系统，DEADLINE，可以被用于不止是内核层“二次获取”漏洞的检测，我们相信在其他应用中也存在类似的漏洞，比如虚拟机管理程序，浏览器，TEE等

Application beyond kernels: hypervisors, browsers, TEE, etc.

在内存安全漏洞之后，逻辑漏洞也应当引起足够的重视。我们希望越来越多的逻辑漏洞可以被系统的建模并检测

Logic bugs are on the rise! We hope that more logic bugs can be modeled and checked systematically

# 谢 谢！

2018 ISC 互联网安全大会 中国 · 北京

Internet Security Conference 2018 Beijing · China

（原中国互联网安全大会）