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

Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, Taesoo Kim

Background

- Virtual memory is divided into userspace and kernel-space regions
- Userspace memory can be accessed from all threads running in that address space as well as from kernel
- Kernel almost never directly dereferences an address supplied by user processes
- Kernel duplicates the data into kernel memory with transfer functions.(copy_from_user, get_user)
- (_user mark) are placed to ensure userspace memory can be accessed only through transfer functions.

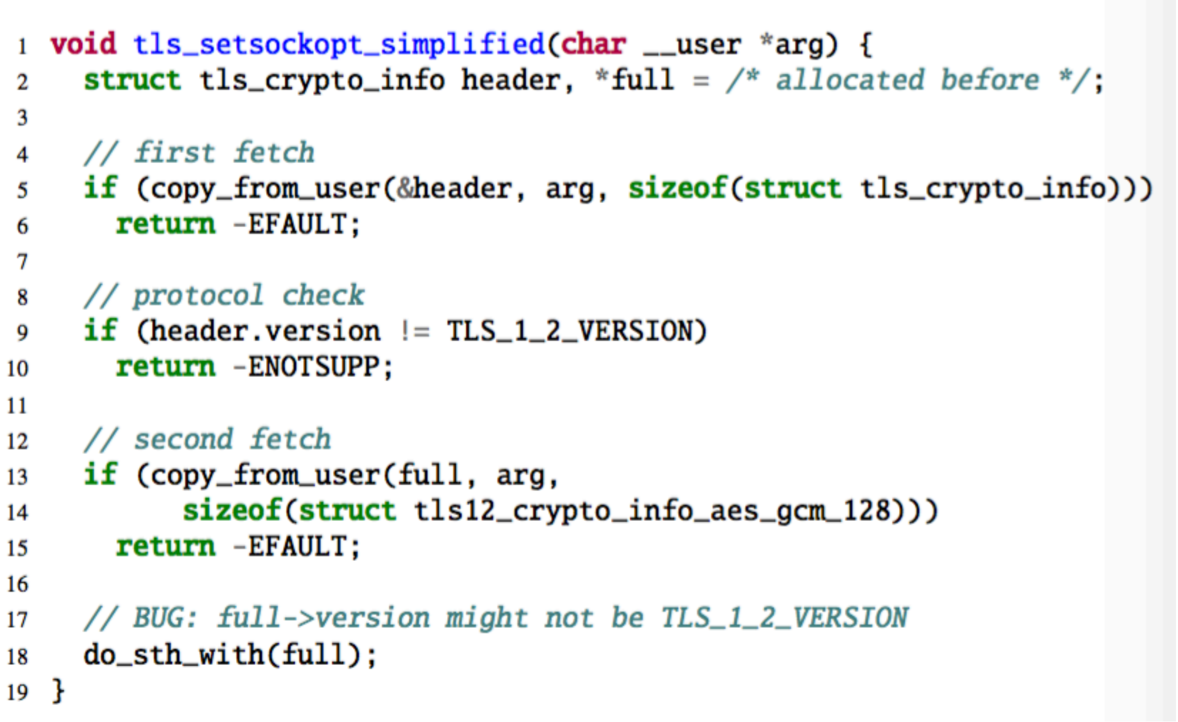
Prior works

- False alerts and missing bugs
- Manually defined patterns can not cover all possible multi-reads
- No attempts to distinguish double-fetch bugs from multi-reads
- No systematic work

Double-fetch bugs

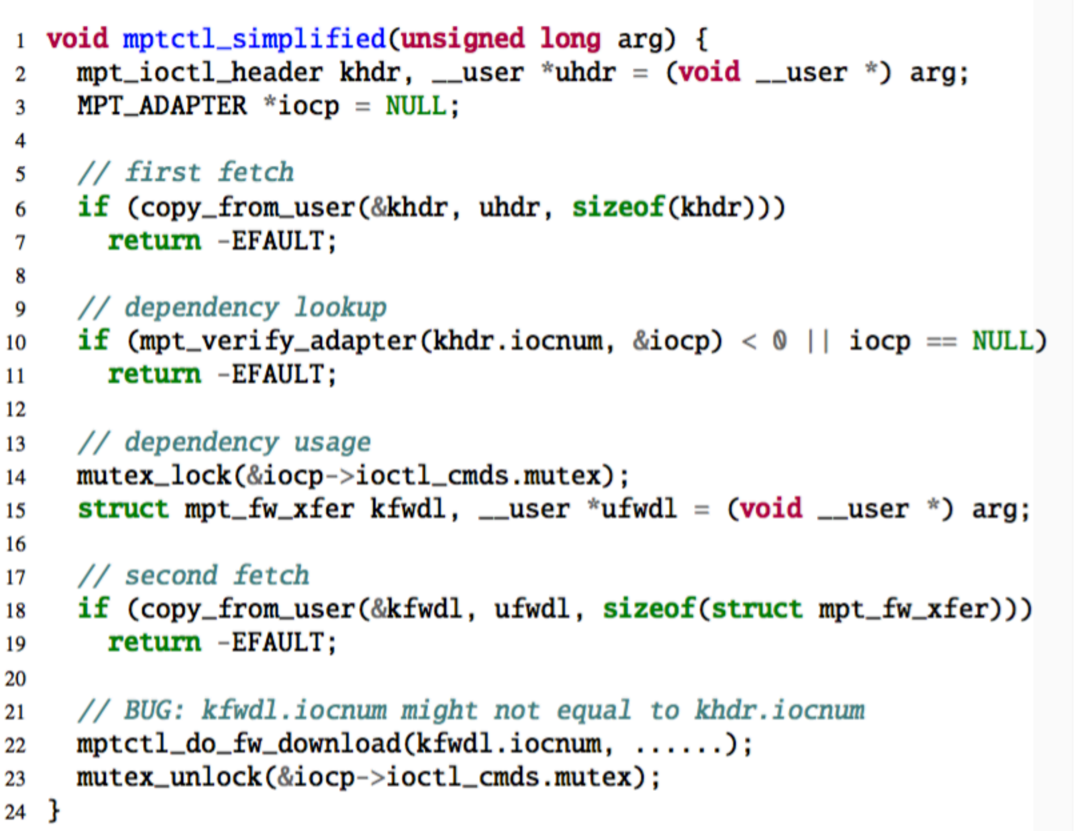
- Multi-read: there are at least two reads from userspace memory
- Overlapped-fetch:The two fetches must cover an overlapped memory region in the userspace
- A relation must exist based on the overlapped regions between the two fetches(control and data dependence)
- Bugs: Cannot prove that the relation established still holds after the second fetch

Examples: Control dependence



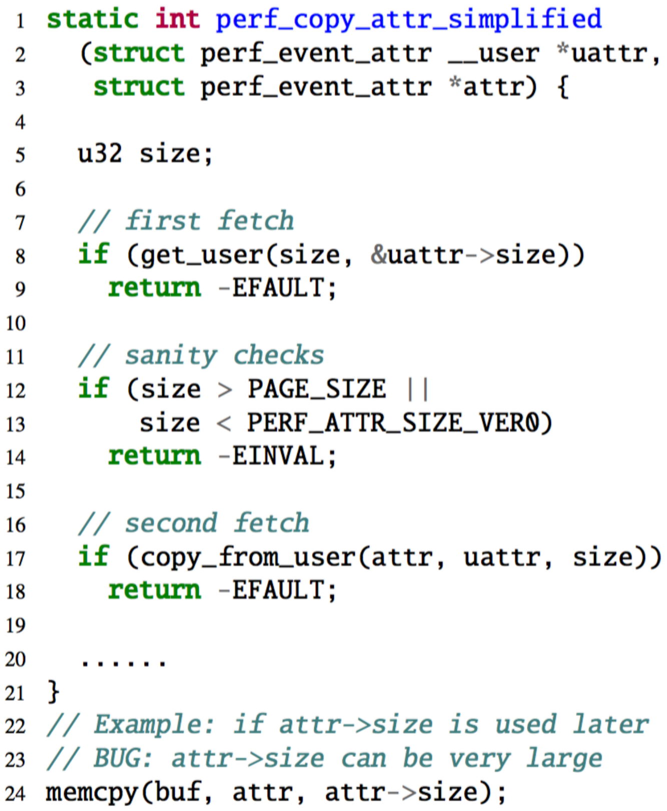
```
void tls_setsockopt_simplified(char __user *arg) {
  struct tls_crypto_info header, *full = /* allocated before */;

  // 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);
}
```

Examples: Data dependence



```c
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);
}
```

Examples: Both

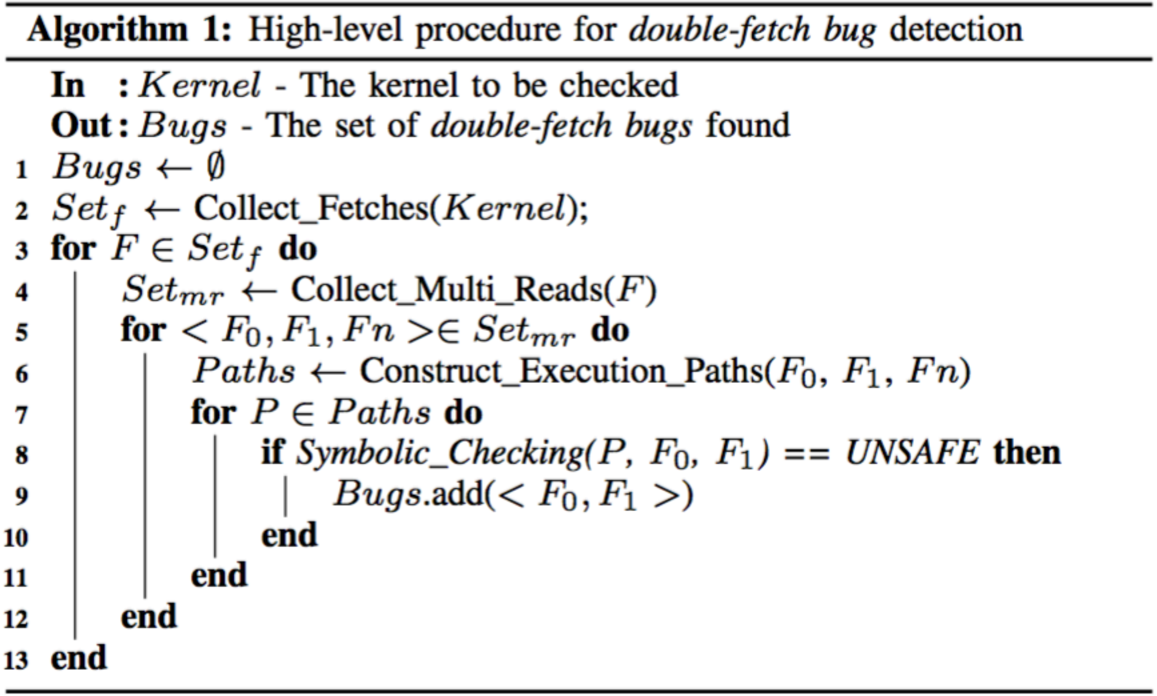


```
static int perf_copy_attr_simplified
  (struct perf_event_attr __user *uattr,
   struct perf_event_attr *attr) {

  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;

  ......
}
// Example: if attr->size is used later
// BUG: attr->size can be very large
memcpy(buf, attr, attr->size);
```

Formal terms

- (A,S) to denote a fetch. A-> start address, S -> size of the memory
- Two fetches: (A0,S0) & (A1,S1); Overlapped memory(A01,S01);
  - A0<=A1<A0+S0 or A1<=A0<A1+S1
- Control dependence: V` must satisfy V
- Data dependence: V` == V

Overview



Algorithm 1: High-level procedure for *double-fetch bug* detection

```
In  : $Kernel$ - The kernel to be checked
Out : $Bugs$ - The set of double-fetch bugs found
$Bugs \leftarrow \emptyset$
$Set_f \leftarrow$ Collect_Fetches($Kernel$);
for $F \in Set_f$ do
    $Set_{mr} \leftarrow$ Collect_Multi_Reads($F$)
    for $< F_0, F_1, Fn > \in Set_{mr}$ do
        $Paths \leftarrow$ Construct_Execution_Paths($F_0$, $F_1$, $Fn$)
        for $P \in Paths$ do
            if Symbolic_Checking($P$, $F_0$, $F_1$) == UNSAFE then
                $Bugs$.add($< F_0, F_1 >$)
            end
        end
    end
end
```

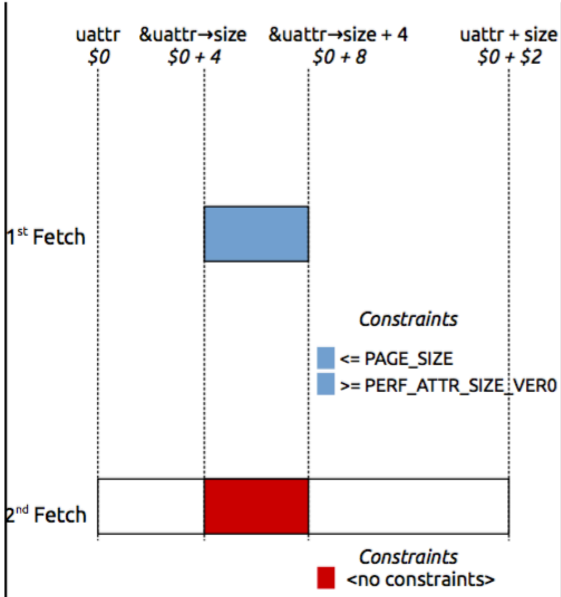
Finding multi-reads

- Identify all fetches in the kernel
- Construct a complete, inter-procedural CFG for the whole kernel
- Perform pairwise reachability tests for each pair of fetches

From multi-reads to 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;

  ......
}
// Example: if attr->size is used later
// BUG: attr->size can be very large
memcpy(buf, attr, attr->size);
```

(a) C source code



(b) Memory access patterns

```
// init root SR
$0 = $PARM(0),    @0 = $UMEM(0) // uattr
$1 = $PARM(1),    @1 = $KMEM(0) // attr
---
// first fetch
fetch(F1) is {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) is {A = $0, S = $2}
@1(0, $2 - 1, K) = @0(0, $2, U1)
---
// check fetch overlap
assert F2.A <= F1.A < F2.A + F2.S
    OR F1.A <= F2.A < F1.A + F1.S
// --> 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)
```

(c) Symbolic representation and checking

Implementation

- Compile source code to LLVM IR
  - Extract build log to collect the compilation flags
  - Use Clang to compile
  - llvm-link to merge

Findings

- 1104 multi-reads
- Confirming previous reported bugs
  - one is miss, due to the IBM S/390 architecture
- New bugs
  - Nine have been fixed
  - four are acknowledged
  - Nine are pending
  - Two won’t be fix (Doesn’t harm)