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

Meng Xu*, Chenxiong Qian*, Kangjie Lu†, Michael Backes‡, Taesoo Kim*,

*Georgia Institute of Technology
†University of Minnesota
‡CISPA Helmholtz Center i.G.

***Abstract*—During system call execution, it is common for operating system kernels to read userspace memory multiple times (multi-reads). A critical bug may exist if the fetched userspace memory is subject to change across these reads, i.e., a race condition, which is known as a double-fetch bug. Prior works have attempted to detect these bugs both statically and dynamically. However, due to their improper assumptions and imprecise definitions regarding double-fetch bugs, their multi-read detection is inherently limited and suffers from significant false positives and false negatives. For example, their approach is unable to support device emulation, inter-procedural analysis, loop handling, etc. More importantly, they completely leave the task of finding real double-fetch bugs from the haystack of multi-reads to manual verification, which is expensive if possible at all.**

In this paper, we first present a formal and precise definition of double-fetch bugs and then implement a static analysis system—DEADLINE—to automatically detect double-fetch bugs in OS kernels. DEADLINE uses static program analysis techniques to systematically find multi-reads throughout the kernel and employs specialized symbolic checking to vet each multi-read for double-fetch bugs. We apply DEADLINE to Linux and FreeBSD kernels and find 23 new bugs in Linux and one new bug in FreeBSD. We further propose four generic strategies to patch and prevent double-fetch bugs based on our study and the discussion with kernel maintainers.

I. INTRODUCTION

Bugs in operating system kernels can be particularly problematic. In practice, they often lead to vulnerabilities that can be exploited to compromise the entire system and cause all kinds of severe attacks, such as privilege escalation [1], [2], information leaks [3], and denial of service [4]. This fact has drawn serious attention from the security community, and the kernel has been increasingly hardened against various types of memory errors, e.g., kASLR [5], kCFI [6], [7], and UniSan [8]. Unfortunately, these mitigations have limited success in taming attacks that exploit logic bugs.

One class of logic bugs that has recently drawn attention is *double-fetch bugs*, which the Bochspwn project [9] introduced for the Windows kernel. Wang *et al*. also studied *double-fetch bugs* for the Linux kernel [10]. A *double-fetch bug* is a special type of race condition bug in which (typically during syscall execution) the kernel reads a particular userspace memory region more than once with the assumption that the content in the accessed region does not change across reads. However, this assumption is not valid. A concurrently running user thread can "scramble" the same memory region in between kernel reads, leading to data inconsistencies in the execution path, which can lead to exploitable vulnerabilities such as sanity check bypassing, buffer overflow, and confused deputy. In reality, researchers have exploited *double-fetch bugs* to escalate privileges on Windows OS [11], [12].

What makes *double-fetch bug* detection an important problem is that, in kernel, it is common to intentionally read data multiple times from the userspace for performance reasons. We call this situation a *multi-read*. To illustrate, consider fetching a variable-length message with a potentially maximum size of 4 KB from the userspace. One approach is to always pre-allocate a 4 KB buffer and copy 4 KB from the userspace in one shot. However, in most cases, this wastes memory and CPU cycles if the effective message payload is 64 bytes or less. Hence, the kernel handles this scenario by first fetching a 4-byte `size` variable and later allocating the buffer and fetching the `size`-byte message. A quick scan over the Linux kernel reveals that there are over 1,000 *multi-reads*. Then, a follow-up question would be: How many of them are real *double-fetch bugs*? Until now, the only way to answer this question was to manually vet the complicated source code of all *multi-reads*. However, this is certainly a scale beyond manual vetting. It therefore becomes a pressing problem that we have to both 1) formally define and distinguish *double-fetch bugs* and *multi-reads* and 2) automatically verify each *multi-read* to check whether it is a bug.

Unfortunately, neither aspect has been addressed perfectly in prior works. Bochspwn [9] defines *multi-reads* as at least two memory reads from the same userspace address within a short time frame, while Wang *et al*. [10] defines *multi-reads* based on a few empirical static code patterns. Due to the imprecise definitions, both works result in many false positives (i.e., incorrectly identified bugs) and false negatives (i.e., missing bugs). More importantly, neither of them can systematically distinguish *double-fetch bugs* from *multi-reads* in definition and they completely leave it to manual verification.

In this paper, we propose DEADLINE, an automatic tool to statically detect *multi-reads* and *double-fetch bugs* with both high precision and coverage. In particular, DEADLINE covers all drivers, file systems, and other peripheral modules that can be compiled under the x86 architecture for both Linux and the FreeBSD kernels. DEADLINE re-discovered all x86-related *double-fetch bugs* reported in [10] and further found *23 new*

bugs in the Linux kernel as well as *a new* bug in the FreeBSD kernel, which significantly outperforms prior work.

To guide DEADLINE to detect *double-fetch bugs*, we first formally model and mathematically distinguish *double-fetch bugs* from *multi-reads*. In essence, a *multi-read* becomes a *double-fetch bug* when 1) two fetches are guaranteed to read from an overlapped userspace memory region, 2) a relation between the two fetches is established based on the values in the overlap, which 3) can be destroyed by a race condition that changes the value in the overlap. With these definitions, DEADLINE detects *double-fetch bugs* in two steps. In the first step, DEADLINE tries to find as many *multi-reads* as possible and also builds execution paths for each *multi-read* by compiling the kernel source to LLVM intermediate representation (IR) followed by a static code analysis. In the second step, DEADLINE follows the execution paths to vet whether a *multi-read* turns into a *double-fetch bug*. To do this, DEADLINE first transforms the LLVM IR into a symbolic representation (SR) in which each variable is represented by a symbolic expression. After this procedure, DEADLINE detects a *double-fetch bug* by solving symbolic constraints on the SR in accordance with the *double-fetch bug* definitions. A satisfiable result indicates that a *double-fetch bug* exists, while an unsatisfiable result means a bug does not exist.

Although the process sounds intuitive, applying it to kernel code imposes several practical challenges. For example, to detect *multi-reads*, DEADLINE needs to systematically explore paths to collect *multi-reads*, and further trim irrelevant instructions and linearize these execution paths. For *double-fetch bug* vetting, DEADLINE needs to symbolize memory reads and writes, and emulate common library functions. DEADLINE embodies various techniques to address these challenges. In particular, instead of using empirical lexical matching [10], it relies on program analysis to collect *multi-reads* and further applies backward slicing and loop unrolling to prune the execution path. For symbolic checking, we propose our own memory model in extension to the model used by traditional symbolic executors [13], [14], [15] to encode access sequence and memory object information. We also write manual symbolic rules to emulate library functions, which alleviate DEADLINE from having to handle the intricacies in these functions.

Besides detection, we complete the analysis cycle of *double-fetch bugs* by discussing how to exploit *double-fetch bugs* as well as four generic ways to fix *double-fetch bugs* based on our experience in patching these bugs as well as the discussion with kernel maintainers.

Contribution. In summary, this paper makes the following contributions:

- We propose a formal and precise definition of *double-fetch bugs* that eliminates the need to manually verify whether a *multi-read* is a *double-fetch bug*.
- We present the design and implementation of DEADLINE, an end-to-end system to automatically vet kernel code with a tailored symbolic execution model specifically designed for *double-fetch bug* detection.
- With DEADLINE, we find and report 23 new bugs in the Linux kernel and a new bug in the FreeBSD kernel.
- We further propose four generic strategies to patch and prevent *double-fetch bugs* based on our study and the discussion with kernel maintainers.

The rest of the paper provides background on *multi-reads* and *double-fetch bugs* (§II), formally defines *double-fetch bugs* (§III), presents an overview of DEADLINE (§IV) and the design of each component (§V, §VI), reveals implementation details (§VII), reports the *double-fetch bugs* found (§VIII), explores *double-fetch bug* exploitation (§IX) proposes several methods to mitigate *double-fetch bugs* (§X), discusses future works (§XI), compares with related works (§XII), and concludes (§XIII).

II. BACKGROUND

A. Address space separation

In modern operating systems, virtual memory is divided into userspace and kernel-space regions. Most notably, the userspace region is separated for each process running in the system, creating an illusion of exclusive address space for each program. Userspace memory can be accessed from all threads running in that address space as well as from kernel. On the other hand, the kernel memory is system-wide and is accessible from the kernel only.

Furthermore, although userspace memory is accessible to the kernel, in practice, the kernel almost never directly dereferences an address supplied by user processes, as any corrupted address, be it by mistake or by intention, will crash the whole system. Instead, if the kernel requires userspace data for execution (as in the case of many driver IOCTL routines), it first duplicates the data into kernel memory and then works on its internal copy. Special schemes, termed *transfer functions*, are provided for this purpose, such as `copy_from_user`, `get_user` in Linux, and `copyin`, `fuword` in FreeBSD. These schemes not only perform data transfer, but also actively validate userspace accesses and handle illegal addresses or page faults. In fact, extensive manual instrumentations (e.g., the `__user` mark) are placed to ensure that userspace memory can be accessed only through transfer functions. Therefore, in this paper, we assume that any *multi-read* must be done through one or more transfer functions.

B. Multi-read as a common practice

Given the limited number of arguments a user process can directly pass to the kernel for a syscall (e.g., maximum six arguments on x86_64), pointers pointing to block structures in userspace memory are often passed to handle large or complex requests. In this case, the kernel often needs to refer back to userspace memory during the syscall. Theoretically, any *multi-read* can be re-designed to a single-read, as illustrated in §I, by pre-defining the shape of the buffer (e.g., the maximum size) and always copying the whole buffer in one shot. However, in practice, this pattern is rarely used due to the waste of memory and CPU cycles, especially when the effective payload is often much smaller than the maximum allowed. Instead, what is typically done in kernel is to first fetch a request header, often

```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);
}
```

Fig. 1: A dependency lookup *double-fetch bug*, adapted from `__mptctl_ioctl` in file `drivers/message/fusion/mptctl.c`

```c
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);
}
```

Fig. 2: A protocol checking *double-fetch bug*, adapted from `do_tls_setsockopt_txZ` in file `net/tls/tls_main.c`

```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);
}
```

Fig. 3: An information guessing *double-fetch bug*, adapted from `con_font_set` in file `drivers/tty/vt/vt.c`

a few bytes only, and then construct the whole request based on the information in the header. Wang *et al*. [10] identified three scenarios of this pattern, namely, *size checking*, where the actual length of the request depends on a `size` variable; *type selection*, where the actual length of the request depends on the opcode of the action performed; and *shallow copy*, where the request header contains a pointer to the second buffer in userspace.

Our analysis confirms these common scenarios but also discovers more interesting reasons and patterns for *multi-reads*.

Dependency lookup. As shown in Figure 1, in the case where there could be multiple handlers for a request, a lookup, based on the request header, is first performed to find the intended handler, and later the whole request is copied in.

Protocol/signature checking. As shown in Figure 2, the request header is first checked against a pre-defined protocol number. The kernel rejects the request early if the protocol is not honored.

Information guessing. As shown in Figure 3, when certain information is missing, the kernel might first guess this piece of information via a sequence of selective reads from the userspace and later fetch in the whole data. A common rationale behind these cases is to abort the processing early if the request is erroneous and save the cost of buffer allocation and a full request copying.

Summary: It is worth noting that the goal of this analysis and categorization is not to enumerate all possible patterns that might cause *double-fetch bugs*; instead, it motivates us to find a generic, formal, yet comprehensive definition of *multi-reads* and *double-fetch bugs* that can unify all these patterns as well as potentially undiscovered ones.

C. Reflections on prior works

Prior works [9], [10] have been successful in finding *double-fetch bugs*. However, the imprecision in their empirically crafted detection rules makes them suffer from a high number of both false alerts and missing bugs. As shown in §II-B, Wang *et al*. [10] use code patterns to lexically match against the kernel source code. Although this approach is scalable, there is no guarantee that manually defined patterns cover all possible *multi-reads*. For example, Figure 1, 2, 3 are *double-fetch bugs*, but they might not fall into the pre-defined patterns. Furthermore, simply assuming that there are no *double-fetch bugs* across loops or function calls (i.e., lack of inter-procedural analysis) might be dangerous. For example, Figure 3 is one case of a *double-fetch bug* that involves loops. More cases, including inter-procedural *double-fetch bugs*, can be found in Table II. In addition, the underlying pattern-matching engine, Coccinelle [16], does not thoroughly support macro expansion, which is heavily used in kernels and could introduce *double-fetch bugs* when certain configurations are enabled. For example, when `CONFIG_COMPAT` is enabled, functions designed for compatibility reasons will be in effect, and several of them can be buggy, as shown in Table II. Ignoring these functions might lead to missing bugs.

Bochspwn [9] instead is a dynamic approach. It defines a *multi-read* as at least two memory reads to the same userspace virtual address 1) in which both originate from kernel code execution and 2) that happen within a pre-defined time frame. Although this definition conforms to the memory access pattern when a *multi-read* occurs, the dynamic nature of this approach makes it very hard to scale to a full-fledged kernel, and the code coverage is inherently low. This has two implications: 1) Bochspwn is limited to finding bugs within hardware devices that can be emulated, which only account for a fraction of drivers covered in the Linux kernel; and 2) even among the emulated drivers, the actual amount of code that can be tested highly depends on the test suites, which, in most cases, only cover the hot paths.

More critically, neither of these works attempts to distinguish *double-fetch bugs* from *multi-reads* and they completely leave it to manual verification. A quick scan over the Linux kernel, however, reveals that over 1,000 *multi-reads* are present in the kernel. Under the definitions in [9], [10], each of them could be a *double-fetch bug* and requires manual verification. Although simple heuristics can be applied to filter out trivial cases, the number of remaining cases might still be overwhelming for manual effort. Therefore, *it is important to distinguish between multi-reads and double-fetch bugs at definition*.

III. Double-fetch Bugs: a Formal Definition

As discussed in §II-B, relying on empirical code patterns for *double-fetch bug* detection is imprecise and could result in a lot of manual effort to verify that a *multi-read* is indeed a *double-fetch bug*. Instead, DEADLINE labels an execution path as a *double-fetch bug* when the following four conditions are met:

1) There are at least two reads from userspace memory, i.e., it must be a *multi-read*. As discussed in §II-A, a userspace fetch can be identified by transfer functions like `copy_from_user`.
2) The two fetches must cover an overlapped memory region in the userspace. If this condition is met, we call the *multi-read* an *overlapped-fetch*.
3) A *relation* must exist based on the overlapped regions between the two fetches. We consider both control and data dependence as relations.
4) DEADLINE cannot prove that the relation established still holds after the second fetch. In other words, a user process can do a race condition to change the content in the overlapped region to destroy the relation.

Conditions 1) and 2) are straightforward to understand. For condition 3), if the execution path can be deviated based on the values from the first fetch, it implies an assumption about these values, and this assumption should be honored by the second fetch. A typical example is shown in Figure 2, whereby after the first fetch, the control flow is deviated if `header.version != TLS_1_2_VERSION`, i.e., the second fetch can never happen. The fact that line 13 can be reached already implies that `header.version == TLS_1_2_VERSION`, which is not re-checked after the second fetch and this makes it a *double-fetch bug*.

For data dependence, consider the bug shown in Figure 1, where the value `khdr.iocnum` is used to look up the correct adapter, `iocp`, to handle the request. The fact that line 18 (the second fetch) can be reached implies that an adapter is already found and a `mutex` is already held. However, in line 22, the adapter is looked-up again (with `kfwdl.iocnum`), but this time, an adapter different from `iocp` can be found if the `iocnum` is changed, leading to a request performed without the intended adapter whose `mutex` is held.

It is also possible that both control and data dependence exist. This typically happens when a variable representing total message size is fetched in, sanity-checked, and later used to do the second fetch, as shown in Figure 4a. The variable `size` must be within a reasonable range, and `attr->size` should hold the effective size of the `attr` buffer. However, after the second fetch, both relations might not hold anymore.

For condition 4), if the relation established in condition 3) is control dependence only, we need to prove that the same set of constraints still holds for the values copied in after the second fetch. In the example of Figure 2, we should check that `full->version == TLS_1_2_VERSION` still holds. On the other hand, if a data dependence is established, re-checking the conditions is not sufficient and a full equality proof is needed. In the case of Figure 4a, checking that `PERF_ATTR_SIZE_VER0 <= attr->size <= PAGE_SIZE` does not reflect the relation that `attr->size` holds the effective size of `attr`. The correct way is to prove that `attr->size == size` in all cases.

Put the above description in formal terms:

Fetch. We use a pair (A, S) to denote a *fetch*, where A represents the starting address of the *fetch* and S represents the size of the memory (in bytes) copied into kernel.

Overlapped-fetch. Two fetches, (A_0, S_0) and (A_1, S_1), are considered to have an overlapped region if and only if:

$$A_0 <= A_1 < A_0 + S_0 \quad or \quad A_1 <= A_0 < A_1 + S_1$$

Correspondingly, we use a pair (A_{01}, S_{01}) to denote the overlapped memory region for the two fetches and a triple $(A_{01}, S_{01}, i = [0, 1])$ to denote the memory copied in during the first or second fetch.

Control dependence. A variable V is considered to be control dependent if $V \in (A_{01}, S_{01}, 0)$ and V is subject to a set of constraints in order for the second fetch to happen. The set of constraints V must satisfy is denoted as $[V_c]$. To prove that a *double-fetch bug* cannot exist in this case, we have to prove that V' constructed from $(A_{01}, S_{01}, 1)$ must also satisfy $[V_c]$.

Data dependence. A variable V is considered to be data dependent if $V \in (A_{01}, S_{01}, 0)$ and V is consumed, such as being assigned to other variables, involved in calculations, or passed to function calls. To prove that a *double-fetch bug* cannot exist in this case, we have to prove that V' constructed from $(A_{01}, S_{01}, 1)$ must satisfy $V' == V$.

IV. DEADLINE OVERVIEW

The formal modeling of *double-fetch bugs* inspired us to use symbolic checking for *double-fetch bug* detection. Compared with actually executing the code with concrete inputs (like Bochspwn [9]), symbolic execution gives us the power of generality, i.e., *solving* for a case that can meet certain conditions or *proving* that these conditions can never be satisfied. This gives DEADLINE the precision of detection and also alleviates the manual effort. Furthermore, symbolic execution, being a static analysis technique, is not limited by the availability of hardware or machine configurations and can theoretically be applied against all the drivers, file systems, and peripheral modules in the kernel source tree. It also enables DEADLINE to start path exploration and checking from virtually any point, in a manner similar to UC-KLEE [17], instead of from fixed entry points like syscall entry or kernel boot.

However, before rushing into symbolic checking, DEADLINE needs to collect as many *multi-reads* as possible since every *double-fetch bug* must be a *multi-read*. Furthermore, for each *multi-read*, DEADLINE constructs the execution paths for the symbolic execution to follow along. DEADLINE achieves this by first compiling kernel source code into the LLVM intermediate representation (IR) and statically analyzing the IR to identify *multi-reads* and prune associated execution paths.

We choose to work with LLVM IR instead of the C source code for several reasons: 1) LLVM IR preserves most of the information needed by DEADLINE, such as type information, function names and arguments, etc. The only information loss is the `__user` mark, which can be easily added back to the IR by adding the mark to LLVM metadata. The `__user` mark is used to differentiate userspace and kernel memory objects, which will be illustrated in detail in §VI-C. 2) LLVM IR is in the single static assignment (SSA) form, which closely mimics the logic for symbolic execution, i.e., assigning a symbolic value to each definition of a variable. Working with LLVM IR also enables the reuse of many LLVM analysis passes such as call-graph construction, function inlining, etc.

Overall procedure. In short, DEADLINE's detection procedure consists of two steps:

1) Scan the kernel and collect as many *multi-reads* as possible along with their execution paths.
2) Check along each execution path to see if a *multi-read* turns into a *double-fetch bug*.

The high-level procedure for DEADLINE is illustrated in Algorithm 1. The process starts by scanning the kernel and collecting all fetches (line 2). This is an easy step, as each userspace fetch is clearly marked by a call to a transfer function (see details in §II-A). Then, for each fetch found, DEADLINE scans backward and forward along the control flow graph for other fetches (line 4), and if other fetches are found, marks them as fetch pairs, each constitutes a *multi-read*. Afterward, DEADLINE builds all possible execution paths that run through both fetches for each fetch pair (line 6). We elaborate these two steps in §V. Finally, for each execution path constructed, DEADLINE invokes its symbolic execution engine and checks whether the *multi-read* is a *double-fetch bug* based on the formal definitions. This step is depicted in detail in §VI.

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
1  Bugs ← ∅
2  Set_f ← Collect_Fetches(Kernel);
3  for F ∈ Set_f do
4      Set_mr ← Collect_Multi_Reads(F)
5      for < F_0, F_1, Fn > ∈ Set_mr do
6          Paths ← Construct_Execution_Paths(F_0, F_1, Fn)
7          for P ∈ Paths do
8              if Symbolic_Checking(P, F_0, F_1) == UNSAFE then
9                  Bugs.add(< F_0, F_1 >)
10             end
11         end
12     end
13 end
```

V. FINDING MULTI-READS

Finding *multi-reads* is the first step for *double-fetch bug* detection. Prior works either used empirical rules [10] or relied on dynamic memory access patterns [9] to find *multi-reads*, both of which could be problematic (e.g., assuming *multi-reads* are intra-procedural). To inherently improve the finding of *multi-reads*, DEADLINE instead employs static and symbolic program analyses to systematically find *multi-reads* against the whole kernel codebase.

A. Fetch pairs collection

In this step, the goal for DEADLINE is to statically enumerate all *multi-reads* that could possibly occur. In particular, DEADLINE tries to identify all the fetch pairs that can be reached at least statically, i.e., there exists a reachable path in the control flow graph (CFG) between the two fetches (i.e., a fetch pair).

One approach is to 1) identify all fetches in the kernel, i.e., calls to transfer functions; 2) construct a complete, inter-procedural CFG for the whole kernel; and 3) perform pair-wise reachability tests for each pair of fetches. Although 1) is easy, given the scale and complexity of kernel software, both 2) and 3) are hard if not impossible in practice. Therefore, DEADLINE chooses to find fetch pairs in a bottom-up manner, as described in Algorithm 2. In short, starting at each fetch, within the function it resides in, DEADLINE scans through both the reaching and reachable instructions for this fetch and among those instructions, either marks that we have found a fetch pair (line 6, 15) or inline the function containing a fetch and re-executes the search (line 9, 18).

Note that, in addition to the two fetches, the enclosing function Fn is also attached to the pair, and we use this triple to denote a *multi-read* in DEADLINE. Conceptually, this Fn is the deepest function in the global call graph (if it can ever be constructed) that encloses both fetches, and later the execution paths will be constructed within this Fn. This alleviates DEADLINE in constructing execution paths from fixed entry and exit points such as syscall enter and syscall

Algorithm 2: Collect_Multi_Reads(F)

```
In  : F - A fetch, i.e., a call to a transfer function
Out : R - A set of triples < F_0, F_1, Fn > representing multi-reads
Fn ← Function that contains F;
R ← ∅;
Set_up ← Get_Upstream_Instructions(Fn, F);
for I ∈ Set_up do
    if I is a fetch then
        R.add(< I, F, Fn >)
    end
    if I is a call to a function that contains a fetch then
        Inline I, redo the algorithm
    end
end
Set_dn ← Get_Downstream_Instructions(Fn, F);
for I ∈ Set_dn do
    if I is a fetch then
        R.add(< F, I, Fn >)
    end
    if I is a call to a function that contains a fetch then
        Inline I, redo the algorithm
    end
end
```

return, which are usually very lengthy with many irrelevant instructions to the forming of a *double-fetch bug*.

Indirect calls. One special case in this process is an indirect call, which is often used in kernel to simulate polymorphism behaviors. DEADLINE does not attempt to resolve the actual targets of an indirect call (in fact, in many cases, they can only be resolved at runtime). Instead, DEADLINE conservatively identifies all potential targets of an indirect call. Specifically, DEADLINE first collects the address-taken functions and then employs the type-analysis-based approach [18], [19] to find the targets of indirect calls. That is, as long as the type of arguments of an address-taken function matches with the callsite of an indirect call, we assume it is a valid target of the indirect call.

B. Execution path construction

In this step, DEADLINE is given a triple $< F_0, F_1, Fn >$ which represents a *multi-read*, and the goal for DEADLINE is twofold: 1) to find all execution paths within the enclosing function (Fn) that connect both fetches (F_0 and F_1) and 2) to slice out the irrelevant instructions that have no impact on the fetches or are not affected by the fetches, for each execution path.

Both parts can be solved with standard program analysis techniques. The first part can be done by a simple CFG traversal within the function Fn, while the second part can be achieved by slicing the function CFG with the following criteria:

- An instruction is considered to *have an impact on a fetch* if the address or size of the fetch is either derived from it or constrained by it.
- An instruction is considered to *be affected by a fetch* if it is derived from the fetched-in value or it constrains the fetched-in value.

With these criteria, we preserve all the control and data dependence relations that we need to prove and thus to decide whether a *multi-read* is a *double-fetch bug*, as defined in §III.

Linearize an execution path. One last step in the path construction is to linearize the paths into a sequence of IR instructions. For a path without loops, linearization is simply a concatenation of the basic blocks; however, for a path with loops, unrolling is required. DEADLINE decides to unroll a loop only once[1]. This imposes a limitation to DEADLINE: DEADLINE is unable to find *double-fetch bugs* caused by one fetch overlapping with itself when the loop is executed multiple times. In fact, in kernel, such *double-fetch bugs* can almost never happen, as fetches in loops are usually designed in an incremental manner, e.g., `copy_from_user(kbuf, ubuf, len); ubuf += len;`. In this case, the two fetches are always from non-overlapping memory regions and will never satisfy the condition for a *double-fetch bug*. On the other hand, unrolling the loop once does help DEADLINE find *double-fetch bugs* caused by two fetches across loops, as shown in Figure 3.

VI. FROM MULTI-READS TO DOUBLE-FETCH BUGS

Prior works [9], [10] rely on manual verification to check whether a *multi-read* turns into a *double-fetch bug*, which can be time consuming and error-prone. Instead, DEADLINE applies symbolic checking to automatically vet whether a *multi-read* is a *double-fetch bug* based on the formal definitions in §III.

A. A running example

To help illustrate the concepts in this section, we provide a running example in Figure 4. It is a *double-fetch bug* found by DEADLINE in the `perf_copy_attr` function and has been patched in Linux kernel 4.13. In summary, the first fetch (line 8) copies in a 4-byte value `size`, which is later sanity checked (line 12, 13) and also used for the second fetch (line 17). However, after the second fetch, the overlapped region `attr->size` is not subject to any constraints until the end of the function. In this case, a user process could put a proper value, say `uattr->size = 128`, before the first fetch so that it will pass both sanity checks and later uses a race condition to change it, say `uattr->size = 0xFFFFFFFF`, which will be copied to `attr->size` and cause trouble if it is later used without caution (line 24). The memory access pattern is also visualized in Figure 4b, which clearly shows that the two fetches have an overlap of four bytes and the constraints on this overlapped region across different fetches are different.

Given that both control dependence (i.e., the sanity checks) and data dependence (i.e., `size` is used in the second fetch) are established between the two fetches, the correct way to check for a *double-fetch bug* is to try an equality proof (i.e., proving that `size == attr->size`), as explained in §III. Since this cannot be proved, DEADLINE flags this *multi-read* as a *double-fetch bug*.

The symbolic execution procedure is shown in Figure 4c. Note that, for illustration purpose, we use $\$X$ to denote the symbolic value of the variable and $@X$ to denote the object in memory that is pointed to by $\$X$. If $\$X$ is not a pointer, $@X$ is

[1]If there are multiple paths inside the loop body, DEADLINE unrolls the loop multiple times, each covers one path.

nil. A memory object can be accessed by a triple $< i, j, L >$, which means a memory access from byte i to byte j. The label L can be either K or U, indicating whether this is a kernel or userspace access, and for userspace accesses, the labels can be $U0$, $U1$, etc. to denote that this is the first or second access to that memory object region.

Due to space constraints, a much more complicated example that illustrates two additional features of DEADLINE, loop handling and pointer resolving, is shown in Appendix §A, Figure 7.

B. Transforming IR to SR

Transforming the LLVM IR to SR is the same as symbolically executing the LLVM instructions along the path. In particular, each variable has an SR while the instructions and function calls define how to derive these SRs and the constraints imposed. All the SRs are derived from a set of root SRs, which could be function arguments, global variables (both denoted as `PARM`), or two special types of objects, `KMEM` and `UMEM`, that represent memory blobs in kernel and userspace, respectively. Function arguments and global variables are considered roots because their values are not defined by any instructions along the execution path. Similarly, the initial contents in `KMEM` and `UMEM` are unknown, and therefore we also treat them as root SRs, although along the execution their contents can be defined through operations such as `memcpy`, `memset`, and `copy_from_user`.

Symbolic execution of the majority of LLVM instructions is straightforward. For example, to symbolically execute an `add` instruction `%3 = add i32 %2 16`, DEADLINE simply creates a new SR, `$3`, and sets it to `$3 = $2 + 16`. However, three types of instructions need special treatment: branch instructions, library functions/inline assemblies, and memory operations.

Branch instructions. As stated before, DEADLINE does not perform new path discovery during symbolic execution; instead, it only follows along a specific path (i.e., a sequence of IR instructions) prepared before the symbolic execution, as illustrated in detail in §V-B. Therefore, whenever DEADLINE encounters a branch instruction, it looks ahead on the path, checks which branch is taken, and uses this information to infer the constraints that must be satisfied by taking that branch, i.e., whether the branch condition is true or false. After doing that, DEADLINE adds this constraint to its assertion set so that later when solving (or proving), it ensures that this constraint is met (or cannot be met). In the running example in Figure 4, line 10, 11 in 4c illustrate this procedure. This is in contrast to traditional symbolic executors [13], [14], [15] which fork states and try to cover both branches upon encountering a branch instruction.

Library functions and inline assemblies. Although the kernel does not have the notion of standard libraries like `libc`, common functionalities such as memory allocation are abstracted out, and most of them reside in the `lib` directory in the kernel source tree. These library functions can be generally categorized into five types: 1) memory allocations (e.g., `kmalloc`), 2) memory operations (e.g., `memcpy`), 3) string operations (e.g., `strnlen`), 4) synchronization operations (e.g., `mutex_lock`), and 5) debug and error reporting functions (e.g., `printk`). We choose to not let DEADLINE symbolically execute into these functions; instead, we manually write symbolic rules for each of these functions to capture the interactions between their function arguments and return values symbolically. Fortunately, for the purpose of *double-fetch bug* detection, there are only 45 and 12 library functions we need to handle for the Linux and the FreeBSD kernel, respectively, which incurs a reasonable amount of manual effort.

In terms of inline assemblies, although they are commonly found in kernel code, not many of them are related to *double-fetch bug* detection and hence will be filtered out early without showing in the execution path. For those that commonly appear in the execution paths (e.g., `bswap`), we write manual rules to approximate their effects on the symbolic values and ignore the rest, i.e., assuming them to have no effects.

C. Memory model

Traditional symbolic executors model memory as a linear array of bits or bytes and rely on the *select-store axioms* and its extensions [20], [21] to represent memory read and write. The select expression, `select(a, i)`, returns the value stored at position `i` of the array `a` and hence models a memory read, while a `store(a, i)` returns a new array identical to `a`, but on position `i` it contains the value `v` and hence models a memory write. This model has been proven successful by symbolic executors like KLEE [14] and SAGE [15]. However, it cannot be directly applied in DEADLINE for *double-fetch bug* detection.

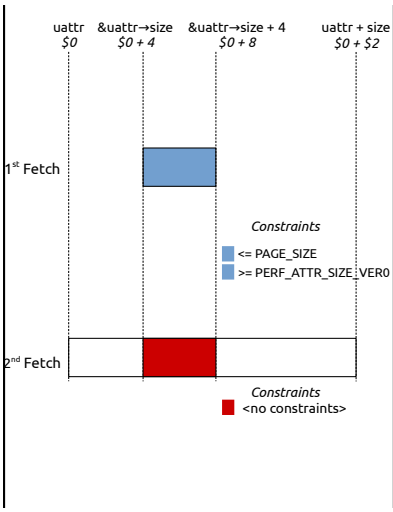
One missing piece in this memory model is that two reads from the same address are assumed to always return the same value if there is no `store` operation to that address between the reads. While this is true for single-threaded programs (or multi-threaded programs with interleavings flattened), it does not hold for userspace accesses from kernel code, as a user process might change the value between the two reads, but those operations will not be backed by a `store` in the trace. In fact, if DEADLINE adopts this assumption, DEADLINE would never find a *double-fetch bug*.

To address this issue, DEADLINE extends the model by encoding a monotonically increasing epoch number in the reads from userspace memory to represent that the values copied in at different fetches can be different. However, for kernel memory reads, DEADLINE does not add the epoch number and does assume that every load and store to the address is enclosed in the execution path. Otherwise, it becomes a kernel race condition, which is out of the scope for DEADLINE and is assumed to be nonexistent. To infer whether a pointer points to userspace or kernel memory, DEADLINE relies on the `__user` mark and considers that any pointer marked as `__user` is a userspace pointer (e.g., variable `uattr` in line 2, Figure 4a), and a pointer without the `__user` mark points to kernel memory (e.g., variable `attr`, in line 3, Figure 4a).

Another extension DEADLINE has to make to the memory model is that instead of assuming the whole memory to be an

```c
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

```
1  // init root SR
2  $0 = $PARM(0),    @0 = $UMEM(0) // uattr
3  $1 = $PARM(1),    @1 = $KMEM(0) // attr
4  ---
5  // first fetch
6  fetch(F1) is {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) is {A = $0, S = $2}
15 @1(0, $2 - 1, K) = @0(0, $2, 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 // --> satisfiable with @0(4, 7, U)
21
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) Symbolic representation and checking

Fig. 4: A *double-fetch bug* in `perf_copy_attr`, with illustration on how it fits the formal definition of *double-fetch bugs* (4b) and DEADLINE's symbolic engine can find it (4c).

```c
static int not_buggy1
  (int __user *uptr1,
   int __user *uptr2) {
  // uptr1 <-- UMEM(0)
  // cannot prove
  // uptr2 == uptr1, so
  // uptr2 <-- UMEM(1)

  int x1, x2;
  get_user(x1, uptr1);
  if(x1 == 0)
    return -EINVAL;

  get_user(x2, uptr2);
  return x2;
}
```

```c
static void *not_buggy2
  (struct request __user *up,
   struct request *kp) {
  // up <-- UMEM(0)

  void __user *ubuf, void *kbuf;
  copy_from_user(kp, up, sizeof(*kp));
  if(!kp->buf)
    return -EINVAL;

  ubuf = kp->buf;
  // cannot prove ubuf == up, so
  // ubuf <-- UMEM(1)
  kbuf = memdup_user(kp->buf, kp->len);
  return kbuf;
}
```

Fig. 5: DEADLINE's memory model cannot prove that the two fetches come from the same userspace object. Therefore, these two cases will not be considered as *double-fetch bugs*.

array of bytes (or bits), DEADLINE uses an array of bytes to represent each single memory object and maps each pointer to one memory object. DEADLINE uses a few empirical rules to create this mapping: 1) Different function arguments or global variables are assumed to be pointing to different memory objects (if they are pointers or integers that can be casted to pointers); 2) Newly allocated pointers (via `kmalloc`, etc) are assumed to be pointing to new memory objects; 3) When an assignment occurs, the object is also transferred (e.g., assigning a function argument to a local variable), meaning that the local variable is pointing to the same object as the argument. In fact, this is implicitly handled by the SSA form of the LLVM IR; 4) For any other pointer, if we cannot prove that its value falls in the range of any existing object, assume it points to a new object. For example, a pointer like (`&req->buf`) is considered as pointing to a subrange of the `req` object, while `req->buf` is considered as pointing to a new object.

Furthermore, when checking for *double-fetch bugs*, DEADLINE considers only the cases where the address pointers are pointing to the same userspace object. For example, in both cases shown in Figure 5, DEADLINE cannot prove that the two fetches come from the same userspace memory object. Therefore, DEADLINE does not mark them as *double-fetch bugs*. This design decision is made based on some implicit programming practices. For example, there is no need to pass in two pointers that point to the same memory region (i.e., the case of `uptr1 == uptr2` on the left side of Figure 5); or it is very uncommon to copy from cyclic buffers (i.e., the case of `up->buf == up` on the right side of Figure 5). However, in the case where DEADLINE can prove that two pointers have the same value, the memory object reference is also transferred, as shown in the case of Figure 7c, line 20.

D. Checking against the definitions

Upon finishing the translation from IR to SR, DEADLINE invokes the SMT solver to check whether all conditions listed in §III can be met.

DEADLINE first checks whether the two fetches, $F_0 =< A_0, S_0 >$, $F_1 =< A_1, S_1 >$, share an overlapped memory region. To do this, on top of the path constraints (which are already added to the solver during symbolization), DEADLINE further adds the constraint $(A_1 \leq A_0 < A_1 + S_1 \quad || \quad A_0 \leq A_1 < A_0 + S_0)$ to the solver (line 18 in the running example Figure 4c). DEADLINE then asks the solver to check whether there exist any overlapped regions with all the assertions. An overlap is represented by a triple $< N, i, j >$ and should be interpreted as that byte i to j in userspace object N being copied into the kernel twice in this *multi-read*. In the running example, there is one overlap, $< 0, 4, 7 >$, as shown in line 20.

If there are no overlapped regions, this *multi-read* is considered safe. Otherwise, for each overlap identified, DEADLINE further checks whether there is control dependence or data dependence established based on this region:

- In the case of control dependence only, collect the constraints for $@N(i,j,U0)$ (denoted as C_0) and $@N(i,j,U1)$ (denoted as C_1) and prove that C_1 is the same as C_0 or is even more restrictive than C_0.
- In the case of data dependence, prove that $@N(i,j,U0) == @N(i,j,U1)$, as shown in line 23 of the running example.
- In the very rare cases where there is no relation found, there is a redundant fetch.

Depending on the result, DEADLINE marks the *multi-read* as safe if the above proofs succeed and a bug otherwise.

VII. IMPLEMENTATION

DEADLINE is implemented as an LLVM pass (6,395 LoC) based on LLVM version 4.0 and uses Z3 [22] version 4.5 as its theorem prover. The rest of this section covers the most important engineering problems we solved when developing DEADLINE, including maximizing code coverage and compiling and linking kernel source into LLVM IR. Due to space constraints, interested readers might refer to Appendix §B for the program slicing and loop unrolling algorithms used in execution path construction.

A. Maximize code coverage

To detect *double-fetch bugs* for the whole kernel, we need to compile not only the kernel base but also as many modules as possible, including drivers, file systems, and peripheral modules that are rarely compiled in the generic configuration. In addition, within a source file, the actual code compiled is usually guided by many `#ifdef` statements. For example, the functions designed to bridge 32-bit applications with 64-bit kernels will be compiled only when `CONFIG_COMPAT` is enabled. We would like to cover these functions too.

To do this, we modify the configuration process for both the Linux and the FreeBSD kernels. For Linux, we rely on the built-in `allyesconfig` setting, which effectively enables all `CONFIG_*` macro (more than 10,000 items). Similarly, for FreeBSD, we rely on the `make LINT` command to output all available options and enable them all to get the build configuration file.

B. Compiling source code to LLVM IR

Since the Linux kernel is not yet compatible with the LLVM toolchain, we compile it with the following steps: 1) we first build the kernel with GCC and collect the build log; 2) we then parse the log to extract compilation flags (e.g., `-I`, `-D`) for each source file and feed the flags to Clang to compile the file again to LLVM IR; 3) we again use the linking information in the build log and use `llvm-link` to merge the generated bitcode files into a single module. Files that are incompatible with LLVM will fail in step 2, which are only eight (out of 15,912 files in Linux 4.13.2).

For the FreeBSD kernel, although it can be successfully compiled with Clang, we cannot directly add the *-emit-llvm* flag to generate LLVM IR because the compilation process checks whether the generated object files are ELF files and will abort if not. Therefore, similar to the Linux kernel compilation, we compile the FreeBSD kernel in the normal way, parse the build log, re-compile the files to IR, and merge them into a single module.

Component	# Multi-Reads		# Double-fetch Bugs	
	Linux	**FreeBSD**	**Linux**	**FreeBSD**
Core modules	25	4	2	0
Drivers	760	86	16	0
Filesystem	246	9	2	1
Networking	73	2	3	0
Total	1,104	101	23	1

TABLE I: Distribution of *multi-reads* and *double-fetch bugs* found by DEADLINE in the Linux and FreeBSD kernels

VIII. FINDINGS

In this section, we show DEADLINE's performance in both detecting *multi-reads* and *double-fetch bugs* in kernel software. Table I summarizes the number of *multi-reads* detected in the Linux and FreeBSD kernels and how many of them are actually *double-fetch bugs*.

A. Detecting multi-reads

This experiment is conducted on version 4.13.3 for the Linux kernel and 11.1 (July, 2017 release) for the FreeBSD kernel. As shown in Table I, DEADLINE reports 1,104 *multi-reads* in the Linux kernel and 101 *multi-reads* in the FreeBSD kernel, as FreeBSD has a much smaller codebase. Furthermore, besides device drivers which have been studied in prior works [9], [10], many other kernel components, including the core modules (e.g., `ipc`, `sched`, etc), might issue multiple fetches from userspace, and some of them can be buggy.

More importantly, the scale of 1,104 *multi-reads* is not suitable for manual verification, not to mention keeping up with the frequent kernel updates. Therefore, this finding supports the claims that formal definitions are needed to define when a *multi-read* turns into a *double-fetch bug* and that automatic vetting is needed to alleviate this manual effort. This motivates the development of DEADLINE.

B. Detecting and reporting double-fetch bugs

Confirming previously reported bugs. We first show that DEADLINE is at least as good as prior works in detecting *double-fetch bugs*. In particular, DEADLINE runs against Linux kernel 4.5, the same version Wang *et al.* [10] used in their work. Out of five bugs reported in [10], DEADLINE found four of them, including `vop_ioctl`, `audit_log_single_ execve_arg`, `ec_device_ioctl_xcmd`, and `ioctl_send_fib`. DEADLINE is unable to detect `sclp_ctl_ioctl_sccb`, as DEADLINE compiles the kernel for the x86 architecture while `sclp_ctl.c` is only compilable for the IBM S/390 architecture. We leave the detection on other architectures for future work.

Finding new bugs. A more important task for DEADLINE is to find new bugs. This experiment is conducted on version

4.12.7 to 4.13.3 for the Linux kernel and 11.1 (July, 2017 release) for the FreeBSD kernel [2].

Out of all *multi-reads* found in the kernels, DEADLINE detected 23 *double-fetch bugs* in Linux and one bug in FreeBSD. We manually checked all the bugs and reported them to the kernel maintainers. The full list of detected *double-fetch bugs* are shown in Table II. At the time of writing:

- Nine bugs have been fixed with the patches we provided.
- Four bugs are acknowledged. We are currently working with the kernel maintainers to finalize the patches.
- Nine bugs are pending for review but no confirmation has been received.
- Two bugs are considered as "won't fix," as the maintainers do not think they are exploitable right now.

In summary, the number of reported bugs is significantly higher than in prior works (six in Linux and zero in FreeBSD). More importantly, while DEADLINE found significantly more *multi-reads*, it further automatically looks for real *double-fetch bugs* in the haystack of *multi-reads*, which is otherwise beyond the scale of manual verification. Furthermore, we anticipate that 14 out of the 24 bugs DEADLINE found could never be found by prior works because of the complications in the bugs, such as falling out of the empirical bug patterns, requiring inter-procedural analysis, loop involvement, and that a function is guarded by `#ifdef` macros.

Bugs marked as "won't fix". We pay special attention to the two bugs rejected for fixing by the developers, as they represent potential false alarms by DEADLINE as well as show the limitations of DEADLINE.

In the case of `uhid_event_from_user`, the developers actually acknowledged that the race condition can occur in userspace; however, they do not believe that this can cause serious harm, as quoted by one of the maintainers: *"With current code, worst case scenario is someone shortcutting the compat-conversion by setting UHID_CREATE after uhid_event_from_user() copied it. However, this does no harm. If user-space wants to shortcut the conversion, let them do so..."*

In the case of `ll_copy_user_md`, DEADLINE falsely reports it due to an assumption on an enclosing function. By constructing execution paths within the enclosing function only, DEADLINE implicitly assumes that if there is an *overlapped-fetch*, careful developers should finish checking that the doubly-fetched values are either the same or subject to the same constraints. In this case, the checking should be `ll_lov_user_md_size(*kbuf) == lum_size` right after the second fetch. Otherwise, once the function returns, the developers lose the opportunity re-assert this relation. However, this implicit assumption does not hold in this case, as the derived value of the first fetch, `lum_size`, is passed out of the function as a return value, and the result of the second fetch, `kbuf`, is passed out by pointer. In other words, even outside this enclosing function, the relation between these two fetches can still be re-checked.

[2]We perform bug finding iteratively as we develop and improve DEADLINE, which explains why several versions of Linux kernel are used.

```
1  char *smb_strdupin
2    (char *s, size_t maxlen) {
3
4    char *p, bt; int error;
5    size_t len = 0;
6    // test and check user strlen
7    for (p = s; ;p++) {
8      if (copyin(p, &bt, 1))
9        return NULL;
10     len++;
11     if (maxlen && len > maxlen)
12       return NULL;
13     if (bt == 0)
14       break;
15   }
16   p = malloc(len, M_SMBSTR);
17   // copy the whole string
18   error = copyin(s, p, len);
19   if (error) {
20     free(p, M_SMBSTR);
21     return NULL;
22   }
23   // BUG: p is not NULL-termed
24   return p;
25 }
```

(a) Buggy function

```
1  // syscall entry point
2  entry: ioctl() {
3    ...
4    // dispatch to device ioctl
5    nsmb_dev_ioctl() {
6      ...
7      // dispatch by command
8      smb_usr_t2request() {
9        ...
10       // [!] double-fetch bug
11       buf = smb_strdupin();
12
13       ...
14
15       smb_t2_request() {
16         ...
17         smb_t2_request_int() {
18           ...
19           // [!] exploitation
20           nmlen = strlen(buf);
21         }
22       }
23     }
24   }
25 }
```

(b) Call stack for an exploit

Fig. 6: An exploitable *double-fetch bug* in the FreeBSD kernel. 6a shows the function flagged as buggy by DEADLINE and 6b shows the end-to-end call stack in the kernel if a user thread tries to exploit this bug by issuing an `ioctl` syscall.

Bug distribution. Aligned with prior research [9], [10], a majority of *double-fetch bugs* are found in the driver code, indicating that drivers are still the most error-prone part in the kernel. This also aligns with the distribution of *multi-reads* where a majority of the *multi-reads* are located in drivers. However, file systems, networking components, or even the core kernel might also be subject to *double-fetch bugs*.

Detection time. On a machine with Intel Xeon E5-1620 CPU (four cores) and 64GB RAM running 64-bit Ubuntu 16.04.3 LTS, DEADLINE finishes detection in four hours for the Linux kernel and one hour for the FreeBSD kernel. Around 20% of the execution time is spent on finding *multi-reads* with static analysis and 80% of the time is spent on symbolic checking on these *multi-reads*.

IX. EXPLOITATION

Exploiting *double-fetch bugs* can be profitable but also challenging. Prior works [9], [10] have identified several ways to exploit a *double-fetch bug* in kernel.

Leaking information. This exploitation typically occurs in a process that does data transfer both to and from userspace, i.e., a request-response situation, as shown in the case of CVE-2016-6130. The bug in CVE-2016-6130 is very similar to the bug in `perf_copy_attr` (Figure 4), where the first fetch sanity checked the `size` value while the second fetch assumes `size` does not change and omitted the sanity check. Later, when the response is copied back to userspace based on the unchecked `size` value, a large chunk of kernel memory will be copied, hence causing a kernel information leak.

Bypassing restrictions. This exploitation typically occurs when the kernel wants to early reject a request from userspace.

#	File	Function	Status	Complication	Patching Strategy
1	block/scsi_ioctl.c	sg_scsi_ioctl	Acknowledged	Macro expansion	Incremental copy
2	drivers/acpi/custom_method.c	cm_write	Submitted	-	Value override
3	drivers/hid/uhid.c	uhid_event_from_user	Won't Fix	Macro expansion	Abort on change
4	drivers/isdn/i4l/isdn_ppp.c	isdn_ppp_write	Patched	-	Single-fetch
5	drivers/message/fusion/mptctl.c	__mptctl_ioctl	Submitted	Inter-proc / Pattern	-
6	drivers/nvdimm/bus.c	__nd_ioctl (1)	Patched	Indirect call	Single-fetch
7	drivers/nvdimm/bus.c	__nd_ioctl (2)	Acknowledged	Indirect call	-
8	drivers/scsi/aacraid/commctrl.c	aac_send_raw_srb	Submitted	-	Value override
9	drivers/scsi/dpt_i2o.c	adpt_i2o_passthru	Submitted	-	-
10	drivers/scsi/megaraid/megaraid.c	mega_m_to_n	Submitted	Unknown pattern	Single-fetch
11	drivers/scsi/megaraid/megaraid_mm.c	mraid_mm_ioctl	Submitted	Inter-procedural	-
12	drivers/scsi/mpt3sas/mpt3sas_ctl.c	_ctl_getiocinfo	Patched	Inter-procedural	Single-fetch
13	drivers/staging/lustre/lustre/llite/llite_lib.c	ll_copy_user_md	Won't Fix	-	Override
14	drivers/tty/vt/vt.c	con_font_set	Patched	Loop / Pattern	Single-fetch
15	drivers/vhost/vhost.c	vhost_vring_ioctl	Submitted	Inter-procedural	-
16	fs/coda/psdev.c	coda_psdev_write	Acknowledged	Unknown pattern	Value override
17	fs/nfsd/nfs4recover.c	cld_pipe_downcall	Acknowledged	-	Value override
18	kernel/events/core.c	perf_copy_attr	Patched	-	Value override
19	kernel/sched/core.c	sched_copy_attr	Submitted	-	Value override
20	net/compat.c	cmsghdr_from_user_compat_to_kern	Patched	Loop involvement	Abort on change
21	net/tls/tls_main.c	do_tls_setsockopt_tx	Patched	Unknown pattern	Single-fetch
22	net/wireless/wext-core.c	ioctl_standard_iw_point	Submitted	-	-
23	sound/pci/asihpi/hpioctl.c	asihpi_hpi_ioctl	Patched	-	Value override
24	netsmb/smb_subr.c	smb_strdupin	Patched	Unknown pattern	Single-fetch

TABLE II: A listing of *double-fetch bugs* found and reported. In the *complication* column, we anticipate the reasons why the bug cannot be found by prior works. For 18 bugs that we submit patches for, we also list the strategy we use to fix the bugs, which is discussed in detail in §X. For the remaining six bugs, the patching is likely to require a lot of code refactoring and we are working with the kernel maintainers to finalize a solution.

For example, in the `tls_setsockopt` case (Figure 2), a malicious user process can bypass the TLS version checking (line 9) by exploiting this double-fetch behavior although the intention of the kernel developers is to reject such requests.

Denial-of-service (DoS). This exploitation typically occurs when a memory operation, e.g., buffer allocation, memory compare, string operations, etc, is affected by the double-fetch procedure. For example, in the case of `smb_strdupin` (Figure 6), it is incorrect to assume that the string copied in after the second fetch is NULL-terminated and later applying the `strlen` on the string is likely to cause an overread into invalid kernel memory regions.

DEADLINE does not attempt to automatically reason about the exploitability of *double-fetch bugs* for two reasons: 1) Unlike memory errors that raise a definitive signal upon exploitation, e.g., an invalid memory access causing a crash, *double-fetch bug* exploitations do not usually raise such a signal and might have to rely on manually defined rules to measure whether the exploit succeeds. 2) Even if we could define all the exploitation rules, constructing them could still be a challenge, as the exploitation point is usually far from to the bug point. In the example shown in Figure 6, the exploitation point `strlen` is two function calls away from the buggy function. In this case, in order to construct an end-to-end exploit, DEADLINE needs to symbolically execute the whole `ioctl` syscall, which would take significantly longer if ever possible. More importantly, even if a *double-fetch bug* is not exploitable right now, it does not mean that it will remain secure in the future. Careless code updates can easily turn a non-exploitable *double-fetch bug* into an exploitable one, as shown in the case of CVE-2016-5728.

X. MITIGATION

Based on our experience in patch creation and our communications with kernel maintainers, there are in general four ways to patch a *double-fetch bug*.

Override with values from the first fetch. In this case, we simply ignore the value copied in during the second fetch and override it with the value from the first fetch. An example is shown in Figure 8, which is actually the patch to Figure 4. By doing so, we ensure that both the control dependence and data dependence established between these fetches are preserved.

Abort if changes are detected. In this case, we add a sanity check after the second fetch to ensure that the intended relation between the two fetches is honored by the user process, as shown in the example of Figure 9, which is actually the patch to Figure 7.

Incremental fetch. In this case, we intentionally skip the bytes copied in during the first fetch. In other words, we start the second fetch from an offset equal to the length of the first fetch. An example is shown in Figure 10. By doing this, these two fetches are now from non-overlapped userspace memory regions and will never constitute a *double-fetch bug*.

Refactor into a single-fetch. If there is only control dependence between the two fetches, we could reduce this double-fetch behavior into a single-fetch, as shown in the example in Figure 11. This approach generally improves the performance as we eliminate one fetch but might result in more lines of code, as now we need to multiplex the `if` checks every time a fetch occurs.

In principle, all these strategies have the same effect—preventing a *double-fetch bug* from being exploited. However, which strategy is taken for a specific *double-fetch bug* is usually based on case-by-case considerations such as performance concerns, number of lines changed, accordance with existing sanity checks, and the maintainers' personal preferences. Besides, several bugs cannot be patched within 50 lines of change due to the complications in current codebases. We are working with the maintainers to finalize the patch.

Preventing exploits with transactional memory. As the root cause of a *double-fetch bug* is the lack of atomicity and consistency in userspace memory accesses across fetches, transactional memory (e.g., Intel TSX) can be a generic solution. Conceptually, one could mark transaction start before the first fetch and mark transaction end after the second fetch. If a race condition occurs, the transaction will abort and the kernel will be notified. DECAF [23] is a proof-of-concept based on these insights. However, it over-simplifies the kernel code by failing to consider the cases of 1) false aborts due to large memory access footprint (which is very likely for *multi-reads*), 2) multiple exit points in the syscall execution (e.g., returning before second fetch), and 3) mixing of TSX-enabled and non-TSX code (e.g., a function can be called within or without a transactional context). Furthermore, DECAF still requires the developers to manually inspect and instrument kernel code. Therefore, to make the TSX-based solution practical, these technical challenges should be addressed and *automated* integration of TSX APIs and kernel code is necessary.

XI. Discussion and Limitations

A. Applying DEADLINE beyond kernels

It is worth-noting that *double-fetch bugs* are not specific to kernels. In theory, it might exist in software systems in which 1) the memory region is separated into sub-regions with various levels of privilege and 2) multi-threading is supported. Therefore, software systems beyond kernels such as Xen, SGX, and even userspace programs like the Chrome browser are also subject to *double-fetch bugs*.

To apply DEADLINE to these software systems, we need to clearly identify the boundary of privileges and the interfaces for transferring data from a low-privilege memory region to a high-privilege memory region. That is, DEADLINE requires pre-defined "fetching" interfaces. Fortunately, we observed that privileged software systems typically have limited interfaces for fetching data from low-privilege to high-privilege memory regions. This is arguably to better maintain the boundary of separated regions. As such, we believe that it is feasible to collect "fetching" interfaces with reasonable engineering effort.

B. Limitations of DEADLINE

We discuss the limitations of DEADLINE from three aspects: 1) which part of kernel source code DEADLINE cannot cover, 2) what kind of execution paths DEADLINE cannot construct for *multi-reads*, and 3) when the symbolic checking for *double-fetch bugs* fails.

Source code coverage. Although DEADLINE covers a majority of kernel codebase, there are two cases DEADLINE currently cannot handle:

1) Files not compilable under LLVM cannot be analyzed by DEADLINE. For Linux 4.13.2, they include three filesystem files and four driver files, which are likely to contain both *multi-reads* and *double-fetch bugs*. We believe this will not be addressed soon with the synergy between the kernel and LLVM community.

2) Although DEADLINE enables all the config options during compilation, DEADLINE certainly misses the code pieces that are compiled when a `CONFIG_*` should be disabled. However, a complete solution would require tweaking Y and N for over 10000 config items, which is unrealistic.

Path construction. DEADLINE aims to find all execution paths associated with a *multi-read*. However, due to the complexity in kernel code, DEADLINE's path construction has three limitations:

1) DEADLINE enforces a limit (currently 4096) to the number of execution paths constructed within an enclosing function. Although in most of the cases there are less than 100 paths, we did observe 17 functions that exceed this limit. Therefore, DEADLINE could have missed *double-fetch bugs* should they exists in those unconstructed paths.

2) DEADLINE also enforces a limit (currently 1) to the loop unrolling, with the assumption that fetches in loops are usually designed in an incremental manner. However, this assumption might be wrong and the fetch inside the loop itself makes a *double-fetch bug* when the loop is unrolled multiple times. Furthermore, there could be cases when cross-loop *double-fetch bugs* occur when a loop is unrolled to a specific time. Although we believe both cases are rare, we cannot prove that they do not exist in kernel.

3) If there is a branch inside a loop, DEADLINE picks only one subpath to unroll the loop. However, there might be cases when a *double-fetch bug* occurs when the subpaths are taken in a specific order when unrolling the loop multiple times, e.g., the true branch is taken in the first unrolling and the false branch is taken in the second unrolling.

Symbolic checking. DEADLINE symbolic checker is limited by how well we model complicated code pieces like assemblies and library function calls as well as the assumptions in the memory model.

1) DEADLINE ignores a majority of inline assemblies, i.e., assuming they have no impact on the symbolic checking. This could lead to missing constraints or incomplete SR assignment, especially when the assemblies issue memory operations. In addition, for the library functions that we manually write rules for, there might be imprecision. For example, we assume that `strnlen` might return any value between 0 and the `len` argument, but actually it is also constrained by the string buffer, which we do not model in the rule.

2) DEADLINE's empirical mapping from pointers to memory object might not reflect the actual situation. As shown in Figure 5, if the function calling `not_buggy1` already ensures

that `utrp1 == uptr2`, or the `struct request` is designed in such a way that `up->buf == up`, then it would be wrong for DEADLINE to treat them as non-buggy.

3) DEADLINE's assumption about the enclosing function might be incomplete. As shown in the "won't fix" case `ll_copy_user_md`, it is possible that the information to assert the relations established between the two fetches are passed out of the enclosing function and re-checked elsewhere.

XII. Related Work

Besides being closely related to the recent work on *double-fetch bug* detection [9], [10], DEADLINE is also related to the research on race condition detection and symbolic execution.

Race condition detection. A *double-fetch bug* is a special type of race condition. Researchers have studied the generic race condition problem extensively and proposed numerous detection techniques, such as model checking [24], [25], type-based systems [26], [27], and data-flow analyses [25], [28], [29]. These static methods are able to check all execution paths with attractive efficiency but generally suffer from a high false positive rate. In contrast, dynamic detection offers precise detection but at the cost of performance and scalability. Much work along the dynamic direction has focused on improving the performance [30], [31], [32], [33].

Unfortunately, none of these techniques can be directly applied to *double-fetch bug* detection. A prerequisite for these methods is to have a complete view of the concurrently running threads (either at source code level or instruction level), which is not true for the *double-fetch bug*, as we have no information on how the user threads might behave. Even when we manage to incorporate the behaviors of user threads into the picture, a race condition detected by these methods can only indicate the existence of an *overlapped-fetch* and we still need to check whether the *overlapped-fetch* turns into a *double-fetch bug*.

Symbolic execution for bug finding. With the recent advances in SMT solvers [34], symbolic execution has proven to be an effective technique in finding bugs in complex software applications [13], [14], [15], [35], [36]. Recent research has further made trade-offs between scalability and path coverage of symbolic execution. A few symbolic execution techniques are now able to analyze even OS kernels such as S2E [37] and FuzzBALL [38], [39]. In particular, S2E employs selective symbolic execution and relaxed execution consistency models to significantly improve the performance. A number of tools (e.g., SymDrive [40], Stack Spraying [41], and CAB-Fuzz [42]) built on top of S2E have been designed to analyze kernel code for various purposes.

DEADLINE also leverages the power of SMT solvers for *double-fetch bug* detection and uses a similar way to collect constraints and assign SR to variables as traditional symbolic executors. However, DEADLINE can be differentiated from them in two ways:

Path exploration strategy: DEADLINE performs path exploration offline and symbolically executes only within a particular path instead of exploring paths online by forking states whenever a conditional branch is encountered. This is because, unlike traditional symbolic executors whose primary goal is path discovery, DEADLINE is not bounded to execute the instructions that are irrelevant to the cause of a *double-fetch bug*, and DEADLINE takes full advantage of that by first filtering out these irrelevant instructions and then constructing paths that must go through at least two fetches and only checks along these paths.

Memory model: DEADLINE extends the memory model used in traditional symbolic executors in two aspects. 1) DEADLINE adds an epoch number to a memory read when it crosses the kernel-user boundary to denote that different userspace fetches from the same address can be different, which is effectively the root cause of a *double-fetch bug*. 2) Instead of assuming a pointer can point to anywhere in the memory, DEADLINE keeps a mapping of pointers to memory objects and uses this to filter out *multi-reads* that are in fact unrelated fetches.

XIII. Conclusion

Detecting *double-fetch bugs* without a precise and formal definition has led to a lot of false alerts where manual verification has to be involved to find real *double-fetch bugs* from the haystack of *multi-reads*. At the same time, oversimplified assumptions about how a *double-fetch bug* might appear have also caused true bugs to be missed.

To systematically approach *double-fetch bug* detection, we first formally model *double-fetch bugs*, which unambiguously distinguishes *double-fetch bugs* from *multi-reads* in mathematical notions Based on the formal model, we implement DEADLINE, a static analysis system that automatically scans though the kernel for both *multi-read* and *double-fetch bug* detection. In particular, *multi-read* detection is done through scalable and efficient static program analysis techniques, while the specialized symbolic checking engine vets each *multi-read* by precisely checking whether it satisfies all the conditions in the formal definition to become a *double-fetch bug*.

As a result, we found and reported 23 new bugs in the Linux kernel and one new bug in the FreeBSD kernel, of which nine have been patched and four acknowledged. This shows the power of symbolic checking for finding complex logic bugs. In addition, we summarized four generic strategies for patching and preventing *double-fetch bugs* based on our experience in patch creation and communication with the kernel maintainers.

XIV. Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This research was supported, in part, by the NSF under award DGE-1500084, CNS-1563848, CNS-1704701, and CRI-1629851, ONR under grants N00014-15-1-2162 and N00014-17-1-2895, DARPA TC (No. DARPA FA8650-15-C-7556), ETRI IITP/KEIT[B0101-17-0644], the German Ministry of Education and Research (BMBF), and gifts from Facebook, Mozilla and Intel.

References

[1] N. Wilfahrt, "Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel," 2016, https://dirtycow.ninja/.

[2] S. Khandelwal, "11-Year Old Linux Kernel Local Privilege Escalation Flaw Discovered," 2017, http://thehackernews.com/2017/02/linux-kernel-local-root.html.

[3] MITRE, "CVE-2017-2584," 2017, https://cve.mitre.org/cgi-bin/cvename.cgi?name=2017-2584.

[4] A. Konovalov, "Exploiting the Linux Kernel via Packet Sockets," 2017, https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[5] J. Edge, "Kernel Address Space Layout Randomization," 2013, https://lwn.net/Articles/569635/.

[6] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[7] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P)*, SaarbrÃijcken, Germany, Mar. 2016.

[8] K. Lu, C. Song, T. Kim, and W. Lee, "UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[9] G. C. Mateusz Jurczyk, "Bochspwn: Identifying 0-days via System-wide Memory Access Pattern Analysis," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug. 2013.

[10] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How Double-Fetch Situations Turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[11] M. Jurczyk and G. Coldwind, "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns," 2013, https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42189.pdf.

[12] I. Institute, "Exploiting Windows Drivers: Double-fetch Race Condition Vulnerability," 2016, http://resources.infosecinstitute.com/exploiting-windows-drivers-double-fetch-race-condition-vulnerability.

[13] C. Cadar, V. G. abd Peter M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2006.

[14] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[15] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[16] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding Collateral Evolution in Linux Device Drivers," in *Proceedings of the 1st European Conference on Computer Systems (EuroSys)*, Leuven, Belgium, Apr. 2006.

[17] D. A. Ramos and D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[18] B. Niu and G. Tan, "Modular Control-Flow Integrity," in *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.

[19] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, ÃŽlfar Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[20] J. McCarthy and J. Painter, "Correctness of a compiler for arithmetic expressions," *Mathematical Aspects of Computer Science*, vol. 1, 1967.

[21] L. de Moura and N. Bjorner, "Generalized, Efficient Array Decision Procedures," Microsoft Research, Tech. Rep., Sep. 2009.

[22] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Berlin, Heidelberg, Mar. 2008.

[23] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features," *ArXiv e-prints*, Nov. 2017.

[24] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[25] E. Yahav, "Verifying Safety Properties of Concurrent Java Programs Using 3-valued Logic," in *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL)*, London, United Kingdom, Jan. 2001.

[26] M. Abadi, C. Flanagan, and S. N. Freund, "Types for Safe Locking: Static Race Detection for Java," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, Mar. 2006.

[27] D. Grossman, "Type-safe Multithreading in Cyclone," in *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, ser. TLDI '03, 2003.

[28] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code," in *Proceedings of the 15th European Software Engineering Conference (ESEC) / 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Dubrovnik, Croatia, Sep. 2007.

[29] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[30] C. Flanagan and S. N. Freund, "RedCard: Redundant Check Elimination for Dynamic Race Detectors," in *Proceedings of the 27th European Conference on Object-Oriented Programming*, ser. ECOOP'13, 2013.

[31] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang, "OCTET: Capturing and Controlling Cross-thread Dependences Efficiently," in *Proceedings of the 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Indianapolis, IN, Oct. 2013.

[32] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, Oct. 2005.

[33] D. Rhodes, C. Flanagan, and S. N. Freund, "BigFoot: Static Check Placement for Dynamic Race Detection," in *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, Jun. 2017.

[34] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[35] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," University of California at Berkeley, Tech. Rep., Sep. 2008.

[36] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, May 2010.

[37] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[38] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, Canada, Jul. 2011.

[39] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[40] M. J. Renzelmann, A. Kadav, and M. M. Swift, "SymDrive: Testing Drivers Without Devices," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[41] K. Lu, M.-T. Walter, D. Pfaff, S. NÃijrnberger, W. Lee, and M. Backes, "Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[42] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

APPENDIX

A. A complicated symbolic checking example

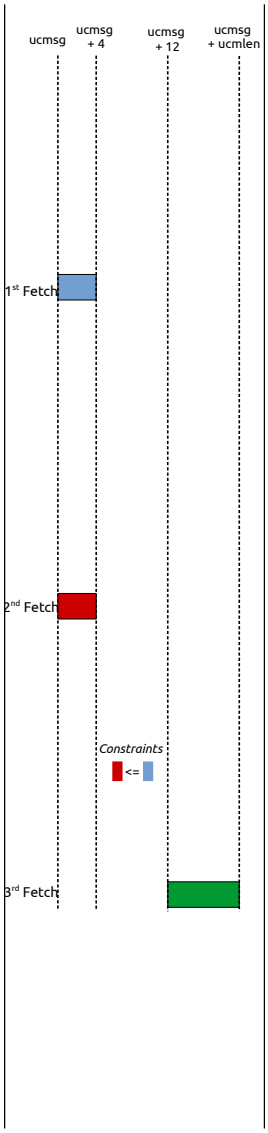
Figure 7 presents a much more complicated example to illustrate DEADLINE's symbolic execution process. In particular, this examples shows two features of DEADLINE:

- Loop unrolling: each of the two `while` loops are unrolled once which is further reflected in the symbolic execution trace: line 9-16, 22-35 (7c).
- Pointer resolving: in DEADLINE's memory model, if DEADLINE can prove that two pointer values are the same, they should be pointing to the same object, which is reflected in line 20 (7c).

This example also shows that although developers tend to exercise precaution to prevent *double-fetch bugs*, for example, by placing the sanity checks at line 36 (7a), such checks might not be sufficient as shown in line 28 (7c).

```
1  int cmsghdr_from_user_compat_to_kern
2    (struct msghdr *kmsg, char *kbuf) {
3
4    struct compat_cmsghdr __user *ucmsg;
5    compat_size_t ucmlen;
6    struct cmsghdr *kcmsg;
7    __kernel_size_t kcmlen, tmp;
8
9    // 1st loop: calculate message length
10   kcmlen = 0;
11   ucmsg = kmsg->msg_control;
12   while (ucmsg != NULL) {
13     // first batch of fetches
14     if (get_user(ucmlen, &ucmsg->cmsg_len))
15       return -EFAULT;
16
17     tmp = ucmlen + sizeof(struct cmsghdr)
18           - sizeof(struct compat_cmsghdr);
19
20     kcmlen += tmp;
21     ucmsg = (char *)ucmsg + ucmlen;
22   }
23
24   // 2nd loop: copy the whole message
25   kcmsg = kbuf;
26   ucmsg = kmsg->msg_control;
27   while (ucmsg != NULL) {
28     // secind batch of fetches
29     if (get_user(ucmlen, &ucmsg->cmsg_len))
30       return -EFAULT;
31
32     tmp = ucmlen + sizeof(struct cmsghdr)
33           - sizeof(struct compat_cmsghdr);
34
35     // sanity check, but insufficient
36     if (kbuf + kcmlen - (char *)kcmsg < tmp)
37       return -EINVAL;
38
39     // irrelevant fetch
40     if (copy_from_user(
41         (char *)kcmsg + sizeof(*kcmsg),
42         (char *)ucmsg + sizeof(*ucmsg),
43         (ucmlen - sizeof(*ucmsg))))
44       return -EFAULT;
45
46     kcmsg = (char *)kcmsg + tmp;
47     ucmsg = (char *)ucmsg + ucmlen;
48   }
49
50   // BUG: the actual message length != kcmlen
51   kmsg->msg_controllen = kcmlen;
52   return 0;
53 }
```

(a) C source code



(b) Memory access patterns

```
1  // init root SR
2  $0  = $PARM(0),              @0  = $KMEM(0)  // kmsg
3  $1  = $PARM(1),              @1  = $KMEM(1)  // kbuf
4  ---
5  // prepare for the 1st batch of fetches
6  $2  = 0,                     @2  = nil       // kcmlen_0
7  $3  = @0(48, 55, K),         @3  = $UMEM(0)  // ucmsg_0
8  ---
9  // unroll 1st loop
10 assert $2 != NULL
11 fetch(F1) is {A = $3 + 0, S = 4}
12 $4  = @3(0, 3, U0),          @4  = nil       // ucmlen_0
13 $5  = $4 - 12 + 16,          @5  = nil       // tmp_0
14 $6  = $2 + $5,               @6  = nil       // kcmlen_1
15 $7  = $3 + $4,               @7  = $UMEM(1)  // ucmsg_1
16 assert $7 == NULL (i.e., @7 = nil)           // exit loop
17 ---
18 // prepare for the 2nd batch of fetches
19 $8  = $1                     @8  = $KMEM(1)  // kcmsg_0
20 $9  = @0(48, 55, K) == $3,  @9  = @3        // ucmsg_2
21 ---
22 // unroll 2nd loop
23 assert $9 != NULL
24 fetch(F2) is {A = $9 + 0, S = 4}
25 $10 = @3(0, 3, U1),          @10 = nil       // ucmlen_1
26 $11 = $10 - 12 + 16,         @11 = nil       // tmp_1
27
28 assert $1 + $6 - $8 >= $11  --> @3(0, 3, U0) >= @3(0, 3, U1)
29
30 fetch(F3) is {A = $9 + 12, S = $10 - 12}
31 @8(12, $10 - 13, K) = @3(12, $10 - 13, U0)
32
33 $12 = $8 + $11,             @12 = $KMEM(2)  // kcmsg_1
34 $13 = $9 + $10,              @13 = $UMEM(3)  // ucmsg_3
35 assert $13 == NULL (i.e., @13 = nil)         // exit loop
36 ---
37
38 // check fetch overlap
39 assert F2.A <= F1.A < F2.A + F2.S
40    AND F1.A <= F2.A < F1.A + F1.S
41 // --> satisfiable with @3(0, 3, U)
42
43 assert F3.A <= F1.A < F3.A + F3.S
44    AND F1.A <= F3.A < F1.A + F1.S
45 // --> unsatisfiable
46
47 assert F3.A <= F2.A < F3.A + F3.S
48    AND F2.A <= F3.A < F2.A + F2.S
49 // --> unsatisfiable
50
51 // check double-fetch bug
52 prove @3(0, 3, U0) == @3(0, 3, U1)
53 // --> fail, as @3(0, 3, U0) >= @3(0, 3, U1)
```

(c) Symbolic representation and checking

Fig. 7: A *double-fetch bug* in `cmsghdr_from_user_compat_to_kern`, with illustration on how it fits the formal definition of *double-fetch bugs* (7b) and DEADLINE's symbolic engine can find it (7c).

B. Static analysis on IR

Slicing and stitching. In order to obtain execution paths to feed to the symbolic engine, we do *backward slicing* to get sensitive instructions that may affect or constraint the address and size argument of transfer functions and do *forward slicing* to get sensitive instructions that may be affected by the fetch-in values of transfer functions, respectively. We stitch the sensitive instructions to construct paths that are possibly executed at runtime. Algorithm 3 shows our slicing and stitching algorithm: the input is a function F that contains a pair of callsites, $C1$ and $C2$, which invoke transfer functions; the output is a set of paths (i.e., P) that contain only sensitive instructions. Pseudo-code (i.e., line 4 – 16) shows the backward slicing algorithm: starting from the parameters of the callsites, which are initialized as the seed vector V_v, we recursively identify all sensitive instructions (i.e., S_i) that may affect the values in V_v according to data and control dependencies, and recursively update V_v according to the use-def chains. We also use a set S_v to record visited values to avoid revisiting the same values. Similarly with the backward slicing, pseudo-code (i.e., line 17 – 30) shows the forward slicing algorithm, in which the instructions affected by the arguments are obtained by checking a value v's $users$, which can be a `LoadInst` loading data from v, or a `BranchInst` using v in its condition, etc. With all sensitive instructions (i.e., S_i) generated from forward and backward slicing, we refine F's CFG by cutting off the instructions not in S_i (i.e., line 31), which provides a refined CFG used for constructing paths (i.e., line 32). We follow original control flows to construct paths that are possibly executed at runtime.

Algorithm 3: do_slicing_and_stitching(F, $< C1, C2 >$)

```
   In  : F - A function contains a double-fetch pair <C1, C2>
   In  : < C1, C2 > - A double-fetch pair callsites in F
   Out : P - A set of paths
1  S_i ← ∅;
2  S_v ← ∅;
3  G ← F's CFG;
   /* do backward slicing to identify instructions that can
      affect the double-fetch pair                          */
4  V_v ← C1.params ∪ C2.params;
5  while !V_v.empty() do
6      v ← V_v.pop();
7      S_v.insert(v);
8      if v.isInst() then
9          S_i.insert(v);
10         for Use u : v.operands() do
11             if !S_v.find(u) then
12                 V_v.append(u);
13             end
14         end
15     end
16 end
   /* do forward slicing to identify instructions that are
      affected by the double-fetch pair                     */
17 V_v ← C1.params ∪ C2.params;
18 S_v.clear();
19 while !V_v.empty() do
20     v ← V_v.pop();
21     S_v.insert(v);
22     if v.isInst() then
23         S_i.insert(v);
24     end
25     for User u : v.users() do
26         if !S_v.find(u) then
27             V_v.append(u);
28         end
29     end
30 end
31 refineCFG(G, S_i) ;
32 P ← getPaths(G) ;
```

Loop unrolling. It is impossible to statically obtain all paths when a program contains unbounded loops. Even for the loops with fixed bounds, exploring all paths is inefficient. Therefore, we unroll each loop n times (n is configurable) to cover as many runtime paths as possible. Algorithm 4 shows how we unroll loops in a CFG. In procedure $merge$: for each loop in a CFG, if the loop does not contain embedded loops, we merge all instructions inside the loop as a single node, which simplifies the CFG to a directed acyclic graph (DAG), in which we can directly get all paths. Procedure $unroll$ takes in a path from the DAG, and recursively unroll the loop nodes for n times until there are no loop nodes on the path. Although this unrolling algorithm is simple, it is proved to be efficient and effective.

Algorithm 4: merge_and_unroll

```
1  Function merge(cfg):
2      while cfg.hasLoop() do
3          for Loop loop : cfg.loops() do
4              if loop.isAtomic() then
5                  loop.merge();
6              end
7          end
8      end
9  Function unroll(path, n):
10     changed ← true;
11     while changed do
12         changed ← false;
13         for Node node : path do
14             if node.isLoopNode() then
15                 for int i = 0; i < n; i = i + 1 do
16                     node.unroll();
17                 end
18                 changed ← true;
19             end
20         end
21     end
```

C. Samples of double-fetch bug patches with different patching strategies

```
 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;
```

Fig. 8: The patch to `perf_copy_attr` follows the override strategy

```
 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;
```

Fig. 9: The patch to `cmsghdr_from_user_compat_to_kern` follows the abort on change strategy

```
 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))
```

Fig. 10: The patch to `sg_scsi_ioctl` follows the incremental copy strategy

```
 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);
```

Fig. 11: The patch to `isdn_ppp_write` follows the refactoring to single-fetch strategy