KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel

Changming Liu
Northeastern University
liu.changm@northeastern.edu

Yaohui Chen
Facebook Inc.
yaohway@gmail.com

Long Lu
Northeastern University
l.lu@northeastern.edu

***Abstract*—Undefined Behavior bugs (UB) often refer to a wide range of programming errors that mainly reside in software implemented in relatively low-level programming languages e.g., C/C++. OS kernels are particularly plagued by UB due to their close interactions with the hardware. A triggered UB can often lead to exploitation from unprivileged userspace programs and cause critical security and reliability issues inside the OS. The previous works on detecting UB in kernels had to sacrifice precision for scalability, and in turn, suffered from extremely high false positives which severely impaired their usability.**

We propose a novel static UB detector for Linux kernel, called KUBO which simultaneously achieves high precision and whole-kernel scalability. KUBO is focused on detecting critical UB that can be triggered by userspace input. The high precision comes from KUBO's verification of the satisfiability of the UB-triggering paths and conditions. The whole-kernel scalability is enabled by an efficient inter-procedural analysis, which incrementally walks backward along callchains in an on-demand manner. We evaluate KUBO on several versions of whole Linux kernels (including drivers). KUBO found 23 critical UBs that were previously unknown in the latest Linux kernel. KUBO's false detection rate is merely 27.5%, which is significantly lower than that of the state-of-the-art kernel UB detectors (91%). Our evaluation also shows the bug reports generated by KUBO are easy to triage.

I. Introduction

OS kernels provide critical system services and hardware abstractions to applications. Its security and stability have been the top priority for OS developers. However, bugs and vulnerabilities are often inevitable in practice when developing such large and complex codebases. As shown in the NVD survey [1], on average over 400 new vulnerabilities are found in the Linux kernel each year.

One class of the often exploited vulnerabilities in kernels is Undefined Behavior (UB) bugs, which, broadly speaking, consist of all undefined behaviors specified by the language standard, for example, ANSI C [9] explicitly specifies many behaviors to be undefined for softwares implemented in C language. Typical examples of UB are integer overflow, division by zero, null pointer deference, out-of-bound access, etc. UBs are much broader in scope than many other specific bugs and can cause a wide range of security and reliability issues [17], [36].

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24461
www.ndss-symposium.org

When viewed individually out of program context, a UB may seem fairly obvious and easy to detect because almost every type of UB has its well-understood triggering condition (*i.e.,* UB condition) and is caused only by a few distinctive instructions (*i.e.,* UB instructions). For instance, an integer overflow occurs when the result of an integer arithmetic instruction goes out of the integer value range.

The previous works on detecting UB in the kernel mostly followed the same basic idea. They focus on UB instructions in code and try to determine if the corresponding UB conditions can be satisfied. Some UB detectors [5] follow the dynamic analysis approach and instrument every potential UB instruction with runtime checks on the UB conditions. This approach suffers from extra overhead and incompleteness i.e., struggle to cover all code. Other UB detectors [38], [37] are purely static and thus in theory can be complete. However, they trade precision for scalability by limiting themselves to intra-procedural analysis. Therefore, these detectors produce extremely high volumes of false positives (*i.e.,* the majority of the detected UBs are in fact infeasible or non-triggerable because the UB conditions can never be met, which cannot be determined by regional or intra-procedural analysis). Moreover, previous works generally do not consider the actual consequence of detected UBs. As shown in [13], many UBs, even if triggerable, do not have a real harmful impact on programs as they are often sanitized right after being triggered.

As a consequence, a scalable UB detector that can produce high-quality bug reports is desired. However, the existing detectors all fall short and often require an impractical amount of manual efforts to validate reported bugs. For instance, KINT [37] generated 125,172 bug reports on the Linux kernel but only 17 of them were confirmed after the team conducted two bug review marathons that covered merely 838 of the reports.

In this work, we propose KUBO (Kernel-Undefined Behavior Oracle), trying to bridge this gap that calls for scalable and precise detection for critical UBs in the kernel.

KUBO focuses on critical UBs, which are triggerable by userspace input. For instance, if the denominator of a division operation is directly sourced from userspace input, it is considered a critical Denial-of-Service (DoS) bug as it can cause an instant kernel panic. KUBO incorporates a new backward userspace input tracking technique to check if a UB condition is modified by such input.

To achieve scalable and high precision detection, KUBO performs an on-demand incremental inter-procedural analysis,

starting from each UB instruction and tracing back to potential userspace input sites. Named incremental call chain upwalk, this analysis only backtracks to a selected caller function on a callchain when needed. The need is determined by an empirical UB triggerability indicator, called Bug Triggerability Indicator (BTI). After BTI is analyzed to be true, KUBO then scans the dependent parameters and only dive into the callers that taint them. This design of inter-procedural analysis allows KUBO to scale and analyze the entire kernel codebase without sacrificing the detection precession or ignoring inter-procedural data and control dependencies as the previous work did.

KUBO collects path constraints along the way and solves them together with the UB condition using an SMT solver[15], which ensures that every detected UB is reachable and controllable by userspace input. Thanks to the on-demand and incremental nature of this analysis, KUBO overcomes the path-explosion problem that prevents standard symbolic execution from being scalable and analyzing large codebases. Finally, KUBO analyzes the consequence of each detected UB via a post-bug analysis, which checks if the value affected by a UB instruction is later used in an unintended way.

Based on our evaluation § V, KUBO is able to scale to the entire Linux kernel (including all drivers) and finishes analyzing 27.8 million LoC in under 33 hours whilst 95% of the subsystems are finished in 15 hours. We apply KUBO to the latest Linux kernel. KUBO found 23 critical UB bugs that are triggered by userspace inputs and pose a real impact on the system. We report all these bugs to kernel developers and received prompt and positive responses:

> "*Thanks for the bug report. The violating code enable accessing the memory regions controlled by the BMC, opening a security hole. It's certainly not expected behavior and should be fixed...*"

14 reported bugs have been confirmed or patched so far. Moreover, motivated by our report, one developer proposed to replace all `char` with `u8` in a module due to our findings of the signed integer overflows directly triggerable by userspace inputs.

We also evaluate KUBO in terms of its precision in two controlled experiments using independently established ground truth. When tested on an old kernel version, which contains 19 known UBs (CVEs), KUBO detects 12 of the 19 UBs (*i.e.,* a false negative rate of 36.8%). When tested on the latest kernel version, KUBO reported a total of 40 UBs, among which 29 are manually verified to be true bugs, scoring a false detection rate of 27.5%. This is significantly lower than that of the existing UB detectors that can work on large codebases [37] (91%). A false detection rate as low as ours was only achieved by some UB detectors [29], [33], [43] that use much heavier analysis techniques, solely focus on a subset of UB and cannot scale to large codebases such as Linux kernels.

Overall, this paper makes the following contributions:

- *Userspace triggerable bugs.* KUBO focuses on detecting UBs in OS kernel that can be triggered by userspace inputs. It uses a light-weight, summary-based dataflow analysis to track UB's dependencies on data fetch from userspace. As a result, 23 critical userspace-triggerable UBs were found in the Linux kernel. 14 of them have been confirmed or patched so far.
- *High-precision detection.* Unlike the previous works on detecting UB in the kernels, which suffer from high false positives, KUBO features a new inter-procedural analysis that tracks data and control dependencies across function calls.
- *Scalable inter-procedural analysis.* The on-demand, incremental call chain upwalk analysis, centering around user-controlled data, allows KUBO to analyze entire Linux kernels by tracking inter-procedural dataflows.
- *Open source.* The source code of KUBO is available at `https://github.com/RiS3-Lab/kubo`.

The rest of the paper is organized as follows. §II provides the background of UB in kernel and the motivation of KUBO. §III and §IV present the system design and implementation details. We evaluate the effectiveness and precision of KUBO in §V and discuss the limitations of KUBO in §VI. Finally, we compare KUBO with related works in §VII and conclude the paper in §VIII.

II. Background and Motivation

A. Undefined behaviors in kernel

Undefined behaviors(UBs) , especially UBSan [4] instrumentable UBs, introduce both security and reliability problems into the kernel. As studied by Xi et al. [38], when a code fragment exhibits behaviors designated as undefined by language standard, the compiler is entitled to do aggressive optimization by falsely assuming that UB would never happen and cause disastrous side effect.

Other security issues caused by UB are also frequently reported. For instance, integer overflow may lead to kernel space code execution(CVE-2018-8781 [3]). To get a full picture of the damages UB can cause, we surveyed the CVEs in Linux kernel directly linked to a UB. Although there is no specific common weakness enumeration (CWE) entry dedicated for UB-related vulnerabilities, we found a large part of CWE-682 and its children to be relevant. We further expanded the list to CWE 128, 190-197, 369, 468, 681, 682. Then we filtered out the CVEs that either its patch is unavailable or it is not directly caused by a UB. As a result, we collected a set of 78 CVEs caused by UB in the Linux kernel. The consequences of these CVEs are shown in Table I. As shown in the table, a good many of them have security implications, they are either directly exploitable (*e.g.,* a malicious user may craft a malformed input and exploit the bug) or causing the system to crash (*e.g.,* nullptr dereference).

We further divide these 78 CVEs into two different groups, the first group, S_{eval}, contains 19 most recent CVEs which we use as groud truth to evaluate our tool's accuracy. The remaining 59 CVEs forms S_{survey} which is used to help us make informed design choices.

B. Prior efforts on finding UB

Due to the severe problems introduced by UB, previous researches [38], [37], [13], [29] trying to eradicate such bugs face major drawbacks: They either cannot scale to large

DoS	Privilege Escalation	Memory Corruption	Arbitrary Code Execution	Information Leak	Unknown
47	15	7	5	3	1

TABLE I: Survey for consequences caused by 78 UB related CVE in the Linux kernel obtained from CVE vulnerability description.

codebases like the kernel or struggle to produce high precision results, which contains large number of false positives or low impact bugs.

The reasons for these limitations are two folds. Firstly, static analysis based approaches [40], [37] struggle to balance between scalability and precision. On one hand, expensive inter-procedural analysis (that are flow- and context- sensitive) consume too much resource when analyzing the kernel. On the other hand, light-weight analysis often has limited analysis scope, thus suffers from a high volume of false positives due to the loss of inter-procedural context. For instance, KINT [37] focus on finding integer overflows in the kernel, reported over 125K cases, and only a fraction of them was confirmed as true bugs.

The second, which is unique to UB, is that the tool needs to prove the triggered UB indeed has a real impact on the program (*e.g.,* correctness, stability, or security). Such impact varies largely in different program contexts. For instance, SAVIOR [13] is a hybrid fuzzing tool aiming to detect UB bugs. It instruments the program under test with Undefined Behavior Sanitizer (UBSAN) [4]. As a result, all the UBs reported by SAVIOR are considered true bugs[1]. However, out of the 481 UBSAN bug reports, 238 of them are deemed harmless by the developers (*i.e.,* triggering these bugs does not have any impact on the affected program). We refer interested readers to their paper for a more detailed discussion.

Due to this extra level of complexity, simply submitting the bug reports from existing automatic tools without first conducting a non-trivial manual analysis may face strong pushback from the developers. As one example:

> "*You need to make deep investigations on your own, before sending mails to the developer mailing list. Static analysis tools having too many false positive are not worth the time spent by humans.*"

Such experience motivates us to develop an automatic UB detector for the kernel that only reports high precision and critical UBs, namely, the UB is triggerable by userspace inputs and once triggered, has real system impact.

C. Severity of kernel UBs

```
    if (flags != TIMER_ABSTIME) {
            ktime_t now = alarm_bases[type].gettime();
-           exp = ktime_add(now, exp);
+
+           exp = ktime_add_safe(now, exp);
    }
    restart->nanosleep.expires = exp;
```

Fig. 1: CVE-2018-13053: A low severity UB in kernel caused by inputs that is not directly user controllable.

[1]These UBs are triggered by fuzzing during run time.

One major factor affecting the severity of UB in the kernel is whether the UB can be easily triggered by malicious userspace code, namely how much effort does it take for an ill-intent actor to exploit the UB.

As mentioned in Section II-B, the impact of UBs varies greatly under different program contexts, they range from harmless low severity to critical exploitable bugs. For instance, if an array index can be set to an arbitrary value without checking via triggering an integer overflow, and the value is directly passed in from userspace (*e.g.,*`ioctl`), such a bug is considered high severity as the system can be easily exploited by userspace code. On the contrary, if the triggering condition is out of the user's control or even purely random, such a bug would be considered having less impact. For instance, Figure 1 shows a UB patch for CVE-2018-13053. In order to trigger this bug, a large timeout is required to overflow this value. As a result, this bug is rated as low severity–with very minimal impact and exploitability [2].

Inspired by this observation, we aim to focus our automatic analysis on finding UBs that are directly triggerable or controlled by the most straightforward attack surface–user-facing interfaces such as data fetch [40] and system calls (*e.g.,*`ioctl`). As shown in Section V, this design choice allows us to find more critical UBs–those with security impact on the kernel.

III. KUBO System Design

A. Key concepts and terms

Before we discuss the design details of KUBO, here we explain several key concepts and terms necessary for understanding our design.

Userspace input: This represents a range of inputs to the kernel from userspace programs. Such inputs are untrusted

TABLE II: All types of supported UB, and their triggering conditions. V denotes the value that is instrumented and checked by UBSan for the corresponding UB. *i*,*ii*... are the different conditions that if anyone is met, the UB is deemed triggered.

Supported UBs	UB condition
out-of-bound bool	*i*. V < `0` *ii*. V > `1`
out-of-bound array index	*i*.V > `array bound`
out-of-bound enum	*i*.V > `enum max value`
integer overflow	if unsigned: *i*.V > `max(uint)` if signed: *i*.V > `max(sint)` *ii*.V < `min(sint)`
divide by zero	*i*.V == `0`
null pointer dereference	*i*.V == `0`
out-of-range object size	*i*. V > `object size`
pointer overflow	*i*.V > `max(uint)`
shift out of bound	for V1 `left/right shift by` V2: *i*.V1<`0` *ii*.V2<`0` if left shift: *iii*. V1*2^{V2} overflows if right shift:*iii*V2 >= bitwidth(V)

(*i.e.,* should be properly validated) by the OS and may enter the kernel space via several different interfaces. According to our survey, 71% (42) of the CVEs in S_{survey} shown in Table I can be directly attributed to missing or incomplete validation on userspace inputs, which justifies our design choice to focus on detecting UBs triggered by userspace inputs. Guided by S_{survey}, below we list the supported userspace input channels by KUBO.

1) *parameters of syscall/ioctl:* The most common system interfaces exposed by the kernel or the device drivers for receiving service requests from userspace. As a consequence, the parameters of these functions are directly provided by the user.
2) *sysctl/sysfs:* This mechanism lets a privileged user/program read or write kernel global variables in the form of file access, such as files under `/proc/sys` and `/sys` directories usually for the purpose of being able to configure the kernel on the run. These kernel variables form a channel for userspace inputs.
3) *Memory data transfer functions:* Apart from user directly feeding input to the parameters of *syscall/ioctl* functions, userspace input can also be fetched from user programs by kernel via designated transfer functions (*e.g.,*`copy_from_user()`), which take as input a user-supplied pointer to a userspace memory buffer. Such pointers should be annotated using the *__user* macro in the kernel code, preventing direct (accidental) dereference by the kernel.

We note that there exist other types of untrusted UB-triggering inputs, such as network payload flowing through the kernel and input from I/O devices. However, based on our survey, these inputs account for a fairly small portion of our surveyed CVEs. Also covering them requires synergy from other lines of works that tries to precisely identify these inputs e.g., PeriScope [32] did a good job identifying input from I/O devices. As a result, as a userspace input oriented approach, KUBO does not consider these untrusted inputs in its design. We discussed the limitation of not considering these inputs in Section VI-A.

Discussed UB scope and triggering condition: As the popular open source OS such as Linux and FreeBSD are implemented in C, the types of UB this paper discussed are also C related. In this context, UB, as specified in the C language standard, has a wide range of causes and it covers varieties of bugs such as uninitialized read and integer overflow, just to name a few. In this paper, we focus on the most typical and critical UBs that are covered by UB sanitizers [4]. Table II lists these UBs and the condition for each type of UB to be triggered (called *UB conditions*). All UBs observed in S_{survey} are covered in this table.

UB instruction: If an instruction can directly produce a UB (*i.e.,* a UB condition is met immediately after the execution of the instruction), it is referred to as a *UB instruction*.

B. System overview

Previous work [38], [37] on statically detecting UB in kernels used simple *intra-procedural* analysis, which can scale up to the whole kernel but scarifies accuracy and thus suffers from high false positives (*i.e.,* the majority of reported UBs are in fact false or non-triggerable).

In comparison, KUBO performs a backward and on-demand *inter-procedural* analysis, which simultaneously achieves the whole-kernel scalability and a much lower false detection rate. Moreover, KUBO is focused on detecting critical UBs that are triggered by userspace inputs.

Figure 2 shows the high-level workflow of KUBO. It starts from each potential UB instruction (flagged by UBSan) and traverses back along the code paths to verify whether the UB is triggerable by userspace inputs. KUBO determines the triggerability based on two requirements pertaining to a traversed code path P: ($R1$), the variable(s) involved in the UB condition must fully depend on (or be solely modified by) userspace inputs; ($R2$), the path constraints on P and the UB condition are satisfiable.

KUBO verifies $R1$ by statically tracking the data and control dependencies of each UB triggering variable (§ III-D). If $R1$ cannot be determined in the current function (① in Figure 2, *e.g.,* the data sources of a UB conditional variable or its dependence are yet to be found), KUBO checks whether an inter-procedural analysis is needed in this case by checking an empirical indicator called BTI. If BTI is true (②), KUBO can have high confidence that the UB can be triggered by userspace inputs (§ III-E). BTI reduces the number of inter-procedural paths that KUBO needs to analyze. If BTI is false (③), KUBO launches the incremental call chain upwalk and continues the backward analysis into selected caller functions, to verify if $R1$ holds (*i.e.,* UB conditional variables take values solely from userspace inputs). The incremental call chain upwalk chooses one caller function at a time and sends the data-flow summary of the function to the input tracking component (④) for another round of checking on $R1$. This process stops when either $R1$ is confirmed/defined or the number of function hops reaches a limit (§ III-F).

After $R1$ is confirmed (either ② or ⑤), as part of the post-bug analysis (§ III-G), KUBO verifies $R2$ by checking if the path constraints and UB conditions are satisfiable using an SMT solver. If $R2$ is met as well, KUBO then performs the rest of the post-bug analysis, which finally confirms whether the UB may cause unintended consequences, and if so, produces a bug report (⑥).

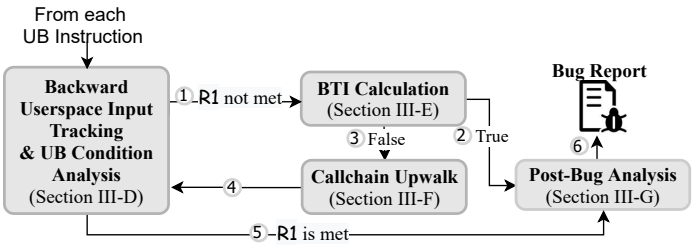

Fig. 2: KUBO workflow overview

C. Identifying UB instructions

To identify all potential UB instructions, KUBO applies the UB Sanitizer (UBSan) [4] on the whole kernel source code and analyzes the code instrumentations done by UBSan (UBSan

TABLE III: Data tags and propagation rules of the back-ward user-space input tracing.

Symbol Definitions	Variables: $v \in V$ Tags: $\Phi = \{N, U, C\}$, where $N > U > C$ Tag query on v: $\mu(v) \rightarrow \phi$ Tag update on v: $\mu[v \rightarrow \phi]$
Tag Propagation Rules	$\frac{\phi' = \mu(v_1)\ \mu' = \mu[v \rightarrow \phi']}{\mu, v = \langle uniop \rangle v_1 \rightsquigarrow \mu'} UNIOP$ $\frac{\phi' = max(\mu(v_1), \mu(v_2))\ \mu' = \mu[v \rightarrow \phi']}{\mu, v = v_1 \langle binop \rangle v_2 \rightsquigarrow \mu'} BINOP$ $\frac{\mu' = \mu[*ptr_1 \rightarrow U]}{\mu, fetch(dst{:}ptr_1, src{:}ptr_2, \ldots) \rightsquigarrow \mu'} FETCH$ (userspace) $\frac{\mu' = \mu[*ptr \rightarrow N]}{\mu, foo(ptr, \ldots) \rightsquigarrow \mu'} CALL$ $\frac{\mu' = \mu[*ptr \rightarrow \mu(v)]}{\mu, store(ptr, v) \rightsquigarrow \mu'} STORE$ $\frac{\mu' = \mu[v \rightarrow \mu(*ptr)]}{\mu, v = load(ptr) \rightsquigarrow \mu'} LOAD$ $\frac{\text{if} \mu(v) == N, \mu' = \mu[\text{values defining } v \rightarrow N]}{\mu, \text{if } v \text{ then goto } v_1 \text{ else } v_2 \rightsquigarrow \mu'} BRANCH$

instruments every instruction that may produce a UB by placing a simple UB condition assertion right before it). Although comprehensive, the majority of the potential UB instructions identified this way is false due to its non-triggerability.

This is not an issue for UBSan, a dynamic sanitizer, because the false UBs never happen during runtime and thus the assertions are never invoked. Being a static analyzer, KUBO treats the UB instructions identified by UBSan merely as possible UB candidates and takes multiple analysis steps (Figure 2) to filter out false UBs and identify true UBs that can be triggered by userspace input.

D. Backward userspace input tracking & UB condition analysis

Starting from each potential UB instruction, KUBO first performs a backward slicing to the beginning of the enclosing function. The slice contains all instructions that the UB instruction has data- or control-dependence on. KUBO then performs a path- and field-sensitive data-flow analysis on the slice, checking if $R1$ is met (*i.e.,* whether a path exists that allows the UB condition to be solely influenced by userspace input). Specifically, for each path in the slice, KUBO tracks the propagation of the following data tags:

- **Userspace Input:** Values originated from the userspace sources (§ III-A) are assigned this tag.
- **Const:** This tag is assigned to constant values.
- **Not known yet:** This tag is assigned to values whose source cannot be determined yet and further backward tracking (inter-procedural) might be needed. Function parameters and some global variables are examples of values with this tag.

The tag propagation rules are defined in Table III using the operational semantics in the form of:

$$\frac{calculation}{context, analyzed\ statement \rightsquigarrow new\ context} \quad (1)$$

Table III listed some of the most important rules for tag propagation given that the overall principle of this tag propagation is simple: When two tags are merged together through a binary operation, the result of this operation takes the most *significant* tag carried by the operands. The tags, ranked by their significance from high to low, are N, U, and C.

Apart from the merging rule, some other rules worth noting are 1). If a U tagged variable is compared with a N tagged variable, first, the result boolean variable will be tagged N according to the $BINOP$ rule, secondly, if this boolean variable is used in a UB-control-dependent branch instruction (the $BRANCH$ rule), the U tagged variable must also be changed to N as this branch instruction exert unknown constraint onto this U tagged variable. 2). If a pointer is passed into a call instruction, then the tag assigned to the content it points to must be changed to N.

This order among the tags and the propagation rules are defined to enable quick determination of $R1$.

After analyzing the current slice, KUBO checks the tag propagated to the UB conditional variable. If it is tagged as U, KUBO confirms that $R1$ is met and it has found a path through which some userspace inputs fully modify the UB condition. If the UB conditional variable is tagged as N (*i.e.,* its data source has not been fully identified yet), KUBO needs to calculate BTI and determines if the backward userspace input tracking needs to be continued to upstream caller functions.

E. Bug triggerability indicator (BTI)

We introduce BTI as an optimization to KUBO. As KUBO seeks to trace the input source dependencies for a targeted UB, we found cases where a full scale search is not necessary, thus we can early report with confidence without really diving into other function(s). We use BTI to distinguish these cases. BTI helps reduce the number of cases where KUBO may otherwise perform inter-procedural userspace input tracking. These cases represent UBs that can be reported with high confidence without fully tracking the origins of all UB conditional variables e.g. meeting the requirement $R1$.

The idea behind BTI is fairly simple, which can be demonstrated using a snippet of real kernel code shown in Figure 3. On Line 8, a signed integer overflow/underflow UB can be triggered solely by knowing that `sr.l_start` is userspace input (tagged U) fetched on Line 5, even though the other operand, namely `filp->f_pos`, is tagged N, i.e., do not know it can be influenced by userspace input or not. In cases like this, we can confidently report the UB without further tracking input sources of UB conditional variables.

Base on this idea, we define BTI, a boolean value for indicating if a UB is already detected even when some inter-procedural data dependencies of the UB condition are not fully established yet. BTI is set to `True` when KUBO's backward input tracking finds that an integer arithmetics operation takes at least one operand whose value originates from userspace (*i.e.,* the operand is tagged as U); otherwise, BTI is set to `False` and KUBO continues onto the incremental call chain upwalk to further analyze UB data sources and the UB condition.

F. Incremental call chain upwalk

KUBO performs on-demand and efficient inter-procedural analysis, called incremental call chain upwalk, for resolving the data dependencies of the UB condition. This analysis is another technical contribution of our work and sets KUBO apart from the previous work, which had to limit themselves to intra-procedural techniques in order to analyze the whole

```
1  int ioctl_preallocate(struct file *filp,void __user
     ↪  *argp)
2  {
3      struct space_resv sr;
4      ...
5      if (copy_from_user(&sr, argp, sizeof(sr)))
6          return -EFAULT;
7      ...
8      sr.l_start += filp->f_pos;
```

Fig. 3: A real example demonstrating the basic idea of BTI

kernel. Without knowing the inter-procedural context and data dependencies, the previous work suffers from very high false positive rates.

The incremental call chain upwalk allows KUBO to achieve both whole-kernel scalability and high precision when detecting UBs. It is on-demand and incremental in that the analysis only backtracks to the selected function callers and does so one layer at a time. To speed up the analysis and avoid repetition, KUBO generates per-function dataflow summaries. When the incremental call chain upwalk reaches a function, KUBO retrieves the dataflow summary for the function, plugs it into the callchain analyzed so far and quickly performs the userspace input tracking on the extended call chain.

Per-function dataflow summary: The dataflow summary captures how, inside a function, the parameters and userspace inputs, if any, propagate outside the function (*e.g.,* via calls and returns). Userspace inputs are identified by the list of system interfaces discussed in § III-A. A summary is generated via a path- and field-sensitive dataflow analysis. The summary generation happens offline. During the incremental call chain upwalk, KUBO adds the dataflow summary of the most recently traversed function to the dataflow model of the call chain, on which a new round of tag propagation can be quickly performed as described in § III-D.

Caller selection: By checking the dataflow summaries of all the functions that may call the current function, the incremental call chain upwalk only selects the caller instructions that can propagate their parameters and/or userspace input to the UB condition variables in the current function. This avoids unnecessary backtracks into caller functions that cannot influence the UB conditions.

For example, when we are about to backtrack a caller function *foo* which has 3 parameters namely p_1, p_2, p_3. Through analyzing the current function, we can obtain the information of what are the parameters that the current UB is depending on, say, p_1 and p_2. We first collect all the call instructions that call this function *foo* from the call graph, for example, c_1, c_2...c_{10}. Then we look at the dataflow summary that contains each call instruction. More specifically, in each dataflow summary, we check whether or not the userspace input can propagate outside the function through the specific dependent parameter(s), e.g., p_1 and p_2 in this case. If yes, then this caller is kept, otherwise discarded. KUBO then keeps the selected caller functions in a queue and analyzes the possible callchains in breadth-first search order.

Number of hops (callchain length): The incremental call chain upwalk continues until either $R1$ is confirmed or the

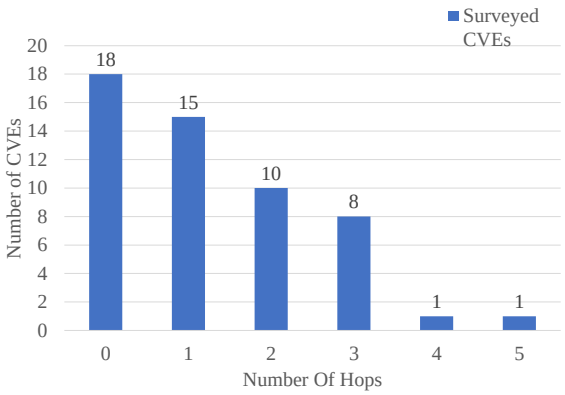

Fig. 4: For each input-related UBs in S_{eval}, how many hops are required to trace to where the input can be identified. Turns out 52 out of 53 UB-related CVEs requires tracing 4 or less hops.

```
1  int blk_ioctl_zeroout(struct block_device *bdev,
     ↪  fmode_t mode,unsigned long arg){
2      uint64_t start = /* from user space */
3      uint64_t len = /* from user space */
4      end = start + len - 1;
5        ...
6      if (end < start)
7          return -EINVAL;
8      truncate_inode_pages_range(mapping, start, end);
```

Fig. 5: A sanity check which eliminate the overflow after it happens

callchain length reaches a limit. We took an empirical approach to find a proper hop limit that allows KUBO to quickly exit from unfruitful callchain upwalks while not missing real UBs. As shown in Figure 4, 98% of the UBs in S_{survey} (excluding the six not triggered by untrusted source) can be triggered on a callchain involving four or fewer calls. As a result, we use four as the max number of hops to trace during incremental call chain upwalk.

G. Post-bug analysis

After finding a code path through which some userspace input controls the UB condition ($R1$), KUBO performs the post-bug analysis, which first checks $R2$ (the path and UB conditions are satisfiable) and then confirms the consequences of the UB before reporting it as a valid bug. The check on $R2$ uses standard path-based symbolic execution and SMT-based constraint solving techniques. The UB consequence check is described below.

Some UBs, such as `divide by zero`, can cause immediate consequences, while other UBs might not exhibit the impact until a later use site of the corrupted value, such as `integer overflow` and `out-of-bound shift`. KUBO keeps analyzing these consequence-delayed UBs beyond the value corruption site. This check is needed because we found many UBs are patched via code inserted after a UB instruction and before any consequence may occur. Figure 5 shows an example of a patched UB, where an integer overflow can happen on Line 4, and a patch is added at Line 6. Without the post-bug analysis, many patched or inconsequential UBs could be falsely reported as bugs.

For each valid UB (*i.e.,* $R1$ and $R2$ are met at the UB

instruction), KUBO checks its consequence by extending the check on R2 from the UB instruction to the subsequent use sites of the corrupted value. This extended check makes sure that value corrupted by the UB instruction can indeed reach a later operation. Our current implementation of the post-bug check does not go beyond function boundaries because most UBs are fixed/checked in the same function where the UB occurs.

Consider again the example in Figure 5. After identifying the overflow on Line 4, KUBO finds the value is used as a parameter passed to the function call on Line 8. It collects the path constraints from Line 4 to Line 8 and adds them to the previously accumulated path constraints and UB conditions. Obviously, the new constraints are no longer satisfiable (due to the check on Line 6) and therefore KUBO concludes that this UB does not have an impact.

IV. Implementation

A. Call graph & taint analysis

KUBO needs the taint summary of each function and the call graph in order to conduct cross-function data flow backtracking. The taint summary is collected with a tool developed based on Dr_Checker [27], a soundy inter-procedural taint analysis for Linux drivers.

As Dr_Checker only considers ioctl function, we added extra userspace input discussed in § III-F into the taint set. 2 notable additions are the $\_\_user$ annotated variables and sysctl variables.

For $\_\_user$ annotated variables, since they would disappear after it is compiled down to LLVM IR, we modify the clang frontend to preserve the annotation. For sysctl variables, we scan through the compiled and linked modules looking for *struct.ctl_table* global variables and arrays. Within each *struct.ctl_table* entry, a global variable and a *proc_handler* are specified. The specified global variable will be recorded as the user-writable global variable, and its *proc_handler* is often used to specify its range. For example, if the associated *proc_handler* is *proc_dointvec*, it means this global variable can be written to whatever value its type can possibly get. However, if the associated *proc_handler* is *proc_dointvec_minmax*, as its name suggests, a constant limit is exerted to this variable. We record this range, as it can improve the accuracy of the later symbolic solving.

The call graph is generated by [42], [24]. In these works, a structured type analysis is used to refine the call target of indirect calls, we directly apply this tool to the tested Linux kernel source code, and reuse the constructed call graph to facilitate the inter-procedural analyses developed in KUBO.

B. Backward tracking & symbolic solving

These are the main components of KUBO which is implemented based on LLVM v9.0 and Z3(C interface) v4.5. It can be applied to the latest Linux kernel (version 5.6.13 at the time of writing).

KUBO takes the generated dataflow summaries and call graphs as input from the above two procedures. We implement the static data flow analysis of backtracing and post-bug analysis based on the LLVM analysis framework. The symbolic execution is built atop DEADLINE [40] where a symbolic modeling and solving for *doule fetch* bug is implemented. We instead modeled the control- and data- dependencies of the instrumented UB instruction and added the inter-procedural analysis for us to fulfill the callchain upwalk analysis and post-bug analysis.

Overall KUBO consists of over 16,000 lines of code written in C++.

V. Evaluation

KUBO is evaluated to to answer the following questions regarding its performance:

- $Q1$: How accurate and complete is KUBO at finding critical UB Bugs? (§ V-A)
- $Q2$: How does each key technique of KUBO contribute to the bug finding process? (§ V-B)
- $Q3$: How does KUBO compare with the state-of-the-art kernel dynamic testing tool like syzkaller[7]? ((§ V-C))
- $Q4$: How much manual effort is needed for triaging KUBO bug reports? (§ V-D)
- $Q5$: What is the performance overhead of KUBO when analyzing the entire Linux kernel? (§ V-E)

Experiment setup: All experiments reported in this section were conducted on a server with a 10-core 2.20 GHz Intel Xeon Silver 4114 CPU and 64 GB of RAM. This server runs Ubuntu 16.04 with gcc 5.4.0 and LLVM 9.0 installed.

For the comparison with dynamic analysis, we used the latest version of syzkaller at the time of writing (commit 3a1f32e). Syzkaller was running in the same environment as KUBO. Both of them were running in one process with a single thread for a fair comparison.

A. Controlled experiments on completeness & accuracy

To study the completeness and accuracy of KUBO's UB detection, we measured the false negative rate (FNR) and false detection rate (FDR) of KUBO via two controlled experiments based on manually established ground truth. In the first experiment, we chose a relatively old Linux kernel (4.1) for testing, which contains 19 CVEs (S_{eval}) that are confirmed UB. Via applying KUBO on this kernel and then examining how many of the 19 CVEs KUBO misses, we derived the false negative rate. We chose this Linux version because it is the oldest that LLVM 9.0 can compile and contains the highest number of known UB vulnerabilities. For the CVEs in S_{survey}, since the LLVM 9.0 cannot generate bitcode for them given that they are not compatible, they are not evaluated by this experiment.

In the second experiment, to measure the false detection rate, we used the most recent Linux kernel (5.6.13) and manually verified all bug reports produced by KUBO. Next, we discuss the details of each experiment and share our findings on when KUBO may miss certain bugs or report false bugs.

1) False negatives: We used S_{eval} which is based on Linux kernel v4.1 released on June 2015, to evaluate the false negative rate of KUBO. KUBO correctly detects 12 of the 19 CVEs in

the tested kernel, a false negative rate of 36.8%. We manually studied the missed cases and present the details below.

Non-user data: As by the design choice, KUBO does not consider UB triggered by non-user input thus does not track such data in the analysis (i.e., only userspace data entering the kernel is considered highly dangerous if it may trigger a UB). Although a minority, we did observe in this experiment five UB CVEs triggered by non-user data or hardware input, including timer (CVE-2018-13053), network (CVE-2019-11477,CVE-2017-7542), DMA (CVE-2016-8636), and kernel global variables (CVE-2020-12826). As expected, KUBO missed these cases. We further examined all the 78 UB CVEs in Linux kernels and found that only 10 (less than 13%) were triggered by non-user input. Table IV shows the breakdown of the sources of inputs triggering the 78 UB CVEs. This result justifies our design choice of focusing on UB triggered by userspace inputs.

Loop unrolling: The other two UBs missed by KUBO in this experiment are due to the insufficient loop unrolling, which iterates each loop only twice in a path-sensitive fashion (§ III). Although simple, this customized strategy is very lightweight and works well in general of UB detection. It may prevent KUBO from detecting a UB if the triggering condition is satisfied only when a loop has to be unrolled three times or more. For example, below the snippet shows the patch for CVE-2018-12896, an integer overflow happens only when i, the loop counter is greater than 32 (i.e., after 32 iterations).

```
    incr = timer->it.cpu.incr;
    ...
    for (i = 0; incr < delta - incr; i++)
      incr = incr << 1;
    for (; i >= 0; incr >>= 1, i--) {
      if (delta < incr)
        continue;
      timer->it.cpu.expires += incr;
-     timer->it_overrun += 1 << i;
+     timer->it_overrun += 1LL << i;
      delta -= incr;
  }
```

It is worth noting that, with the simple loop unrolling strategy, KUBO can detect common loop-induced UB, such as CVE-2017-7294 because the triggering condition for UB can often be statically determined after iterating a loop twice (e.g., summing up an array of integers fetched from userspace from zero). Moreover, we found that only 7 out of 78 UB CVEs (less than 9%) are triggered through loops (Table IV). Therefore, KUBO's simple customized loop unrolling strategy suffices in practice.

2) False alarms: In the second experiment, we measure KUBO's false detection rate based on the most recent version of the Linux kernel (5.6.13). This experiment also aims to demonstrate KUBO's ability to detect previously unknown UB bugs in the latest Linux kernel. Same as the previous experiment, KUBO analyzed the whole kernel with all *KConfig* entries enabled and drivers included. That is 166 kernel modules in total and 320,937 UBSAN instrumentations checking for potential UB bugs. As for the results, KUBO generated 40 UB bug reports. We manually examined each report and confirmed 29 of them to be true UB bugs (11 false bugs), a false detection rate of 27.5%. This rate is significantly lower than that of the existing UB detectors that can work on large codebases [37], whose false detection rates is 91%. This false detection rate was barely achieved by some UB detectors [29], [33], [43] that use much heavier analysis techniques, solely focus on a subset of UB on much smaller codebases.

Through the communications with the kernel developers and our own investigation, we found that KUBO generated the 11 false positives for two reasons:

Const kernel globals: sysctl allows userspace programs to overwrite certain kernel-space global variables. KUBO treats such global variables as a source of userspace inputs. However, these global variables may not always take their values from userspace inputs. In fact, some may be initialized with constant values in the __init functions. Two false positives were caused by KUBO mistakenly treating a variable writable by sysctl as userspace input. KUBO can eliminate this kind of false positives by treating all sysctl variables defined in __init functions as constants.

Limited post-bug analysis: The post-bug analysis performed by KUBO is intra-procedural by design. This is because the use of a UB variable usually remains local (the affected value is used right after the bug). However, in very few cases, the UB affected value is passed to another function where a UB check takes place. Nine detected UB bugs fall into this category where the bug is triggered but its check is performed in a different function which means these bugs have been noticed and handled, so we mark them as false positives.

```
/* Check Source */
if (!access_ok(source + dest_offset, count)) {
        IVTVFB_WARN("Invalid userspace pointer
 ↪  %p\n", source);
        ...
        return -EINVAL;
}
```

As shown in the code snippet above, the parameters source, dest_offset, and count are all directly from userspace, so source+dest_offset could overflow but function access_ok eliminates the possibility of UB by making sure memory from source+dest_offset to source+dest_offset + count is a piece of valid userspace memory.

3) Previously unknown bugs: For the 29 detected true positive UBs which have not been handled or checked, 14 of them have been confirmed as new and critical bugs and patched correspondingly. Seven still await response. Two are confirmed as true positive but will not be fixed. Since the other six of them are later validated to be an intentional violation or benign wrap-around, we did not report it with the developer. The status of each report is shown in Table V.

Intentional UB is referred to as a UB that appears to be undefined but its result is somehow defined and expected. This can be done, for example, by forcing the signed integer overflow which by design is undefined to be well-defined e.g., 2's complement, via specifying certain compiler flag during compilation; Wrap-around means unsigned integer overflow which is essentially well-defined modulo arithmetic for C

attack vector	ioctl	fetch	syscall	sysctl	disk	timer	network	global variable	loop
count	26	20	7	3	3	2	6	4	7

TABLE IV: Attack vectors of the collected benchmark. The first four vectors directly introduce userspace inputs.

#	error	modules	Function	Status	hops	implication
1	u+	block	blk_ioctl_discard	acknowledged	0	OOB Write
2	u×	drivers:infiniband	uverbs_request_next_ptr	will not fix	3	User Pointer Overflow
3	enum	drivers:media	ccdc_data_size_max_bit	patched	2	DoS
4	u+	drivers:rapidio	rio_mport_maint_rd	submitted	0	OOB Read
5	u+	drivers:rapidio	rio_mport_maint_wr	submitted	0	OOB Write
6	array	drivers:staging:gasket	gasket_partition_page_table	submitted	2	OOB Read
7	u+	drivers:misc	genwqe_unpin_mem	submitted	1	OOB Write
8	u+-	drivers:soc	aspeed_p2a_region_acquire	acknowledged	1	Arbitrary Write
9	s-	drivers:message	mptctl_gettargetinfo	submitted	0	OOB Write
10	trunc	drivers:usb	sisusb_setreg	patched	1	Logical Error
11	trunc	drivers:usb	sisusb_setidxregor	patched	3	Logical Error
12	s+	drivers:usb	sisusb_setidxreg	patched	3	OOB Write
13	s+	drivers:usb	sisusb_getidxreg	patched	3	OOB Read
14	s+	drivers:usb	sisusb_setidxregandor	patched	3	OOB Write
15	s+	drivers:usb	sisusb_setidxregmask	patched	3	OOB Write
16	shift	drivers:usb	sisusb_write_mem_bulk	patched	0	DoS
17	u+-	drivers:video	kyro_dev_overlay_viewport_set	will not fix	1	N/A
18	s+	fs	kern_select	submitted	0	Logical Error
19	shift	drivers:video	lcd_cfg_vertical_sync	acknowledged	1	DoS
20	shift	drivers:video	lcd_cfg_horizontal_sync	acknowledged	1	DoS
21	shift	sound	snd_hwdep_dsp_load	patched	0	Data Confusion
22	u+	sound	snd_emux_hwdep_ioctl	submitted	1	OOB Read
23	s+	kernel	panic	acknowledged	0	Zero Delay Panic

TABLE V: New UB detected by KUBO and reported to the kernel developers. Each line is a reported bug. For each bug, we list its error operation with the bug (**s+** stands for signed integer overflow etc.), the corresponding module, the function name where the bug resides, its current status, how many hops it requires to trace the bug and its security implication, specially we shade the bugs that are evaluated to have security implication.

language [6]. However, if not handled with care, it still can lead to security breach such as CVE-2018-5848, bug #1 and #8 in Table V, mainly because such a wrap-around is not anticipated by the developer.

One example of a detected, purely benign wrap-around is shown below, where the `name` array from the userspace is hashed by (ab)using unsigned integer overflows.

```
static void warn_on_bintable(const int *name, int
  ↪  nlen)
{
  int i;
  u32 hash = FNV32_OFFSET;
  for (i = 0; i < nlen; i++)
    hash = (hash ^ name[i]) * FNV32_PRIME;
...
```

Only two out of the 23 bugs we reported were rejected (will not be fixed) by the kernel developers, namely Bug #2 and #17 in Table V. For Bug #2, the developer acknowledged that the overflow can happen and may cause userspace faults but failed to see any harm to the kernel thus chose not to fix it. For Bug #17, the developer stated that although the UB bug is valid, the code is for legacy hardware. Without access to the hardware, it's hard to evaluate the impact of this overflow.

We manually evaluate the security implication by observing how the detected UB is used and impact the system. 17 out of the 23 reported bugs (74%) are considered to be critical with obvious security implications (shaded rows in Table V).

```
if (unlikely(copy_from_user(&maint_io, arg,
↪  sizeof(maint_io))))
        return -EFAULT;
if ((maint_io.offset % 4) ||
    (maint_io.length == 0) || (maint_io.length % 4)
        // a wrap-around can happen here
    ||(maint_io.length + maint_io.offset) >
    ↪  RIO_MAINT_SPACE_SZ)
        return -EINVAL;
buffer = vmalloc(maint_io.length);
```

We use 2 typical examples to demonstrate the exploitability of the found bugs. In one example shown in the figure above, *maint_io.length* and *maint_io.offset* are fetched directly from userspace, and the sum of these 2 variables are compared with a constant *RIO_MAINT_SPACE_SZ* which is *16MB*. However, since this sum can wrap-around, an arbitrary kernel buffer can be allocated. Such a buffer will subsequently be used to communicate and iterate data from/to userspace to/from device, possibly leading to either memory leak or arbitrary write which can be used as primitive for further exploitation.

```
uint64_t range[2], start, len;
if (copy_from_user(range, (void __user *)arg,
↪  sizeof(range)))
        return -EFAULT;
start = range[0];
len = range[1];
...
//start + len can wrap-around and bypass
// this check
if (start + len > i_size_read(bdev->bd_inode))
        return -EINVAL;
...
//after passed into truncate_bdev_range
//wrong memory can be truncated
err = truncate_bdev_range(bdev, mode, start,
    start + len - 1);
```

Another example, as shown in the snippet above, a wrap-around could happen for the addition of the *start* and *length* of a piece of memory, as they are coming from userspace. However, a wrap-around check is missing, thus one can bypass the check at line 9 and the overflowed value is passed into *truncate_bdev_range* which is a wrapper of *truncate_inode_pages_range*, possibly truncating a wrong piece of memory and causes memory corruption issues.

In summary, the results from the two experiments reported in this sub-section (§ V-A) show that KUBO detects critical userspace triggerable UB bugs in Linux kernels with significantly

lower false negative and false detection rates than previous works.

B. Component-wise evaluation

In this section, we evaluate and justify each design choice in KUBO based on the collected statistics of analyzing the latest kernel.

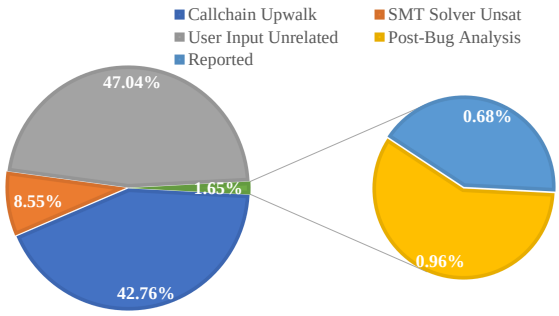

Fig. 6: The percentage of the UBs filtered out by each technique among the all UB instrumentations. If solely using SMT solver, only 8.55% UB instrumentation can be deemed false. In the meantime, a large portion of UB instrumentation can be removed because they are not related to userspace input.

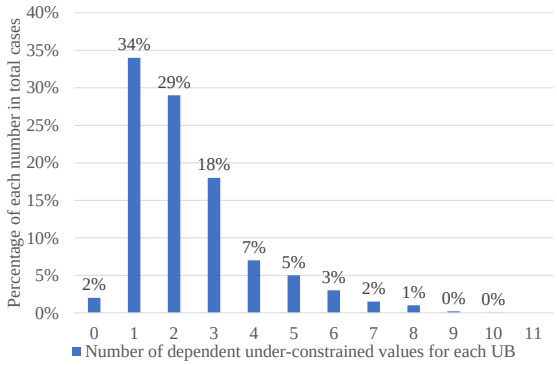

Fig. 7: Distribution of number of dependent under-constrained values for instrumented UB. 98% of the instrumented UB depend on at least one under-constrained values. 83% of them depends on 3 or less under-constrailed values.

1) User-input centric detection: To evaluate the effectiveness of each technique, we summarized how many instrumented UBs are filtered out in each stage in Figure 6. As shown in the figure, SMT solver can only remove 8.55% UB instrumentations i.e., 91.45% of the UBs will be deemed satisfiable thus being kept. This high satisfiability rate is the major evidence of the significant gap between the theoretically satisfiable UB and the real UB in reality which makes previous works struggle with FP. KUBO bridges this gap by searching for userspace input. As Figure 6 shows, generally only 10.2% (8.55%+1.65%) UBs are caused directly by userspace input, yet, according to our survey, they account for the majority of the exploitable real-world vulnerabilities.

2) Incremental call chain upwalk: In this subsection, we evaluate how incremental call chain upwalk affects the scalability and detection result of KUBO. We further present the results based on different hop limits to study how the configuration of hop limit may affect the effectiveness of incremental call chain upwalk.

hop	0	1	2	3	4	5	6	7+
# of UB instrumentation	320937	106240	29256	27451	24794	13176	7003	4557
# of bugs detected	15	27	32	38	40	40	40	NA
FDR	20.0%	22.2%	28.1%	23.7%	27.5%	27.5%	27.5%	NA

(a) Per-hop evaluation for detected bugs and false alarms

hop	0	1	2	3	4	5
# of bugs missed	14	11	9	7	7	7
FNR	73.3%	57.9%	47.4%	36.8%	36.8%	36.8%

(b) Per-hop evaluation for false negatives.

TABLE VI: Per-hop evaluation for FDR and FNR. For FDR (subtable VIa), we report how many UB instrumentations are processed by this hop (row 2). How many bugs are detected up to this hop (row 3) and the corresponding false detection rate (row 4). For FNR (subtable VIb), we report how many bugs are missed out of the 19 bugs ground truth for each hop (row 2) and the corresponding false negative rate (row 3).

Effect on scalability: As shown in Figure 7 where we survey the number of under-constrained values for each instrumented UBs, 98% UBs depends on at least one under-constrained value, this high percentage of dependency on under-constrained values makes it necessary to trace back to the callers to gain more visibility. When we trace back the callers, instead of blindly scaling up to each possible caller, we only select the callers that can taint all UB-dependent under-constrained values. During this selection, 46.07% of callers are filtered out because they cannot taint all under-constrained values, this big reduction is the key factor for KUBO to scale up.

Per-hop evaluation: To better understand how the configuration of different hop limits affects the result for false detection rate (FDR) and false negatives rate (FNR), we first measured how many UB instrumentations are processed by each hop in the lastest kernel experiment. As shown in Table VI, the number of hops dropped dramatically at the first 3 hops. This largely alleviates the path explosion problem that most symbolic execution tools have. As for false positive rates, with the number of hops getting larger, the FDR remains consistent around 25% and plateaued after 5 hops. This suggested that increasing hops does not effectively detect more bugs mainly because a fairly large portion of UB instrumentations e.g., about 91%, has been processed in the first 3 hops.

In the experiment of different hops for false negative evaluation, as shown in Figure VIb. False negatives benefit a lot from the increasing number of hops as it dropped by half from hop 0 to hop 3, and also plateaued afterward. This indicates the intrinsic limitation of KUBO which is being unable to handle other sources of input and loops. And this cannot be simply solved by increasing the hops.

3) Post-bug analysis: As the kernel grows more and more sophisticated over time, an increasing number of UB is being noted and properly handled. Unlike other types of bugs that can be eliminated completely, the UB is usually sanitized after

id	Modules	Testable?	Why (not)?
1	block	y	built-in
2	fs	y	built-in
3	sound	y	built-in
4	kernel	y	built-in
5	drivers:infiniband	y	virtual driver
6	drivers:media	n	AM4x Cortex-A9 specific
7	drivers:rapidio	n	physical device required
8	drivers:staging:gasket	n	physical device required
9	drivers:misc	n	physical device required
10	drivers:soc	n	ASpeed BMC SoC specific
11	drivers:message	n	physical device required
12	drivers:usb	y	usbfuzzer
13	drivers:video	y	virtual driver

TABLE VII: Summarization of testability of syzkaller for all modules where KUBO found bugs. 7 modules are testable while other 6 are not. For testable modules, **built-in** means this module can be tested in the default setting; **virtual driver** means it's a driver for software e.g. *rxe* for infiniband; **usbfuzzer** means syzkaller emulated usb stack, thus making usb driver testable. For untestable modules, it can be either it's unique to certain chip e.g. 6 and 10, or a physical device is required.

Modules	Found UB	buggy function	True bug?	Why not?
block	No	NA	NA	NA
fs	No	NA	NA	NA
sound	shift	snd_timer_user_ccallback	n	post-bug check
kernel	shift	ext4_fill_super	n	post-bug check
	s*	yura_hash	y	NA
	s*	__ntfs_write_inode	n	non-reproducible
drivers:infiniband	No	NA	NA	NA
drivers:usb	No	NA	NA	NA
drivers:video	s+	__v4l2_find_nearest_size	n	post-bug check

TABLE VIII: UB Bugs found by running syzkaller for each testable modules for 48 hours. 5 UB bugs were reported for these 7 modules. For each found bug, we recorded their UB type, the corresponding function, whether if it's a true bug, and the reason why if we validate it to be a false bug.

it actually happens. This challenges all UB detectors to not only ensure the UB is triggerable but also to go beyond the UB instruction to make sure the triggered UB is unhandled and can cause unintended consequences. For discussion of a better post-bug analysis, please see Section III-G. In our experiment, if without post bug analysis, the false detection rate will be increased from 27.5% to 68.3%.

C. Comparing with syzkaller

1) Setting up syzkaller: In this experiment of comparing with syzkaller, the kernel image of the same version where KUBO found unknown bugs was compiled with the default recommended configuration from syzkaller and additional UBSan instrumentation enabled. For each subsystem where KUBO has found bugs, we launched a syzkaller instance where only the relevant syscalls are enabled so that syzkaller can focus on that specific subsystem e.g., for *sound* subsystem, we only enabled syscalls specified in *dev_snd_seq.txt* and other related syscalls description files.

There are two exceptions for the above procedure, one is for the instance launched for *kernel* subsystem where we did not specify any syscall, so every syscall can be tested; Another one is for the device drivers since most of the device drivers are not testable in the syzkaller VM as they need to communicate with a real physical device; Worse still some of the device drivers are unique to certain chip or architecture, thus cannot be compiled into the kernel image on a regular server environment. We summarized the testability situation in Table VII. The untestability problem is solved either by 1) syzkaller's emulation of the hardware stack e.g. usbfuzzer [8] or 2) testing a virtual driver (driver communicating with software).

2) Syzkaller reported bugs: We ran each instance of syzkaller testing individual kernel module for 48 hours and the UBSan reports are summarized in Table VIII. As shown in this table, five UBs were detected in total across 3 different kernel modules. However, after manual validation, four of them were determined to be false bugs. For these four false bugs, three of them are because they are checked right after the UB takes place, another one of them is because it was decided as un-reproducible by syzkaller.

There is one true bug in *yura_hash*, as shown in the figure below, which is a signed overflow. However, according to our previous communication with the developers, all signed-overflow are converted to 2's complement wrap-around in the kernel module, making this overflow purely benign. Question has been submitted to the developer in regard to whether this overflow is harmful or not.

```
1  u32 yura_hash(const signed char *msg, int len)
2  {
3    int j, pow;
4    u32 a, c;
5    int i;
6    for (pow = 1, i = 1; i < len; i++)
7      pow = pow * 10;// to overflow this, this loop
8      // should be unrolled at least 10 times.
9  ...
```

Based on the above reported results, KUBO has two advantages over syzkaller: 1) directly applicable to (any) kernel code despite the lack of customized hardware. It is also worth noting that it takes non-trivial effort to set up syzkaller to test against different drivers, for example, configuring the kernel with driver-specific *KConfig* entries and installing extra user-space libraries for virtual drivers e.g., *RDMA Core Userspace Libraries* for *rxe* driver are usually required. and 2) lower false detection rates mainly thanks to post-bug analysis.

However, we note that there are bugs easily found by syzkaller but difficult for a static analysis tool (*e.g.,* the bug in *yura_hash* requires unrolling the loop for at least 10 times). This makes KUBO and the dynamic testing approach a good complement for each other.

D. Triaging efforts

Since static analysis reports usually take a huge amount of manual efforts to triage, in this section, we measure the time spent on manually verifying the bug reports. We invited a graduate student to validate all the bugs detected by KUBO. In this process, the validator needs to check if the bug can be triggered via user-controlled data, and the triggered UB has a real system impact. Thanks to the modest number of reports, and well-marked attack vector, it only costs the student 5 hours to validate all the reports. Compared to the traditional symbolic execution based static approach like KINT, where two bug review marathons were used to validate only 0.6% of the 125,172 generated reports, we believe our userspace focused detection approach is more actionable and usable in practice.

Out of the 23 submitted reports, despite the fact that we were unable to provide a PoC program to cause any sensible consequence. Only via the description of the data sources and the UB found by KUBO, the developers were able to

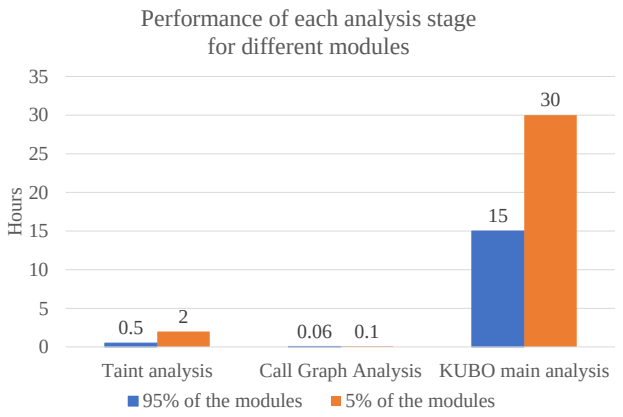

Fig. 8: Total time spent in each analysis phase. Although processing time varies between different modules due to their varying complexity, and all tasks were put into a thread pool, most of the modules can be finished within 16 hours (0.5h+4m+15h) (blue bar), a few extremely complex modules take much longer with the worse case scenario being 33 hours (2h+6m+30h) (red bar) detailed in § V-E.

quickly acknowledged 14 of them, thanks to the inherent security implications that they can be directly triggered by user-controlled data. The average turn-around time is 44 hours.

E. Performance

In this subsection, we report the time cost in each analysis stage, the result is presented in Figure 8. Note that, for each analysis stage, instead of analyzing each module one by one, all tasks were put into a thread pool, and we measured the time of an individual module from the moment it entered the thread pool until it finished.

For taint analysis, most of the subsystems take half an hour to generate the per-function taint summary while drivers/net drivers/infiniband drivers/gpu and drivers/net take about 2 hours. The call graph analysis takes all modules for less than 6 minutes.

As for KUBO, the main analysis, 161 subsystems are finished within the first 15 hours since the analysis starts. Only 5 subsystems take extra 15 hours due to the high volume of annotated sanitizers and complexity of the codebase, these subsystems are fs, drivers:video, drivers:gpu, net and drivers:scsi. In conclusion, KUBO can produce high-quality bug reports without introducing too much overhead, the whole analysis can be finished in a reasonable human time.

VI. Discussion & Future Work

A. Model hardware input

KUBO does not handle hardware input sources at the moment, and this contributes to 5 of the missing bugs out of S_{survey}. Hence, we identified several unique challenges to model hardware inputs.

First of all, hardware interfaces are much more diverse due to various hardware specifications (*e.g.,* DMA, MMIO). Thus when identifying untrusted input coming from hardware, it usually needs to be analyzed case by case. Also, the data from the device are often structurized thus a field-sensitive analysis is generally required. Last but not least, the data sent from the device to the driver can be pre-processed by the device firmware, this requires the analysis tool to understand the full software stack including the firmware, in order to complete the constraints. Given that there has been works studying the driver-device interaction like PeriScope [32] which tries to identify the untrusted input from the devices, it would be interesting to see how to incorporate these two lines of works to facilitate the detection of more bugs in the kernel, we leave this to future exploration.

B. Improve for less false alarms

As we can see from the evaluation § V-A1, 80% of KUBO's false positives are caused by the incomplete post-bug analysis due to its limitation of being intra-procedural. In addition, understanding the semantics at a UB sink site is also important in order to precisely screen the non-impactful UBs, since ultimately it is the developer's intention that needs to be understood by the analysis [33], [29], [10]. And such an intention is expressed in various ways and can be hard to enumerate.

Existing works [29], [31], [33] trying to distinguish between harmful UBs and benign ones generally hinge on the idea that the undefined/overflowed values being used in sensitive functions is definitely not intended by the developer. This poses advantages for false alarms but is theoretically unsound in the sense that sensitive functions are hard to enumerate and the sinks (UB impacting site) apart from the sensitive functions can be also critical. KUBO takes the first step (towards a generic approach) to model the constraints of the UB's use sites, however, the post-bug analysis is still bounded by being intra-procedural for performance concerns. As a result, understanding more about the possible ways of how an undefined value can be used and a (possibly) efficient inter-procedural analysis to locate the dangerous sinks can be interesting for future exploration and warrants a more extensive study.

VII. Related Work

A. Undefined behavior detection/fixing

Previous works reason very well about why undefined behavior exists and the conditions to trigger each of them. Wang et al. [36] study the causes and consequences of all kinds of undefined behaviors. STACK [38] [39] investigates the unstable code which might be optimized out due to the compiler's false assumption of all user's programs being well-defined. D'Silva et al. [18] studied UB, among other issues, introduced by compiler optimizations. Chris et al. [21] systematically study the causes and consequences of UB in C and propose extra semantics to handle the undefinedness in C. Lee et al. [22] tries to address the undefined behavior in the design of LLVM IR instruction set by introducing new instruction.

This line of works, in terms of efforts to detect UB, typically uses limited intra-procedural or local analysis and does not consider UB triggerability by user inputs. As a result, they usually report overwhelmingly high volumes of bug reports and false positives, which are extremely difficult to vet in practice. As for the UB fixing methods, they generally rely on new security features implemented through program rewriting. These techniques are generally not applicable to programs as

large and complex as the OS kernel. In comparison, KUBO focuses on detecting critical UB triggered by userspace input with a much lower false detection rate, thanks to its scalable and accurate data and call chain analysis.

B. Integer overflow (IO) detection

Integer overflow/error is a common type of UB, which attracts a lot of research attention. KINT [37] is a static integer integrity checker applicable to the Linux kernel. It relies on range analysis and user-annotated taint input to infer if an integer can overflow. In comparison, KUBO expands the problem scope to all types of UBs. Moreover, unlike KINT, whose context-insensitive range analysis and intra-procedural symbolic execution together result in extremely high false detection rate, KUBO achieves scalability i.e. finish analysis within 33 hours, to the entire Linux kernel without sacrificing much the accuracy i.e. FP rate is 27.5%.

SIFT [23] generates input filters for programs to prevent integer errors during runtime. However, its approach is not directly applicable to OS kernels or complex software whose input channels are numerous and diverse. DIODE [31] uses a targeted branch enforcement strategy to find a path that can trigger IO at memory allocation sites. Despite being relatively precise, it is narrowly focused on IOs at the memory allocation sites and cannot scale up to big programs like OS kernels.

Another line of work on detecting IOs including IntScope [35], IntFlow [29], IntEq [33], Osiris [34], and IntRepair [28], trying to detect IO using various techniques. IntScope [35] and IntFlow [29] use taint analysis to locate the overflow affected by untrusted input source under the observation that the harmful integer errors often stem from untrusted input and the triggered error must affect certain sensitive functions e.g. memcpy. Osiris [34] targets the IOs in the smart contract. IntRepair [28] aims at automatically detecting and repairing IO through program rewriting. IntEq [33] formally defines what's a benign IO and achieves high accuracy in distinguishing benign IOs from harmful ones. This idea is a great complement to our post-bug analysis to mute more false positives, however since it also requires the overflowed value to flow into a critical sink which might be far from the UB instruction, it still has the problem of not being able to find a meaningful sink. This might not be a big problem for small program, but for code as complex as the kernel, how to locate the sink is an issue.

Idea-wise, our motivation to focus on input from userspace agrees with their observation that user-input affected UB is critical, but KUBO distinguishes from these works in that 1). Prior works [29] use a flow-based method which only utilized coarse-grained data-flow analysis while KUBO strictly distinguishes userspace input from under-constrained memory and assert that only when a UB is solely affected by userspace input can it be regarded as a true bug, for other likely cases, we use BTI to model the likelihood. 2). Their observation in [35], [29], [33] that only the overflows whose value affects sensitive functions can be regarded as meaningful, might be generally true for userspace code. For kernel, however, given the compactness of semantics an integer could represent and how critical certain integers could be, any unintended or unintended undefinedness and wrap-around are worth being paid attention to. As a result, the scope should not be limited to a set of predefined sensitive functions. Lastly, the benchmark adopted in all of these works are generally small to medium sized userspace program which is an indicator that they might not scale to large codebase like the kernel. As stated in IntRepair [28] which neither has false positive nor false negative, *"we expect that in even more complex and large real-world programs, INTREPAIR would report false positives.*

C. Symbolic execution for bug finding

Symbolic execution has been widely used with SMT solvers [16] for finding bugs [19], [12], [30], [26], [14], [11]. These works, along with the aforementioned STACK and KINT, made great progress towards analyzing large codebases. However, there is always a trade-off between scalability and precision. In particular, works like UCKLEE and KINT tried to gain better scalability by performing symbolic execution on partial program paths starting from certain interesting functions within the kernel and carry on forward. As a result, they suffer from high false detection rates. KUBO also adopts the general approach of symbolic execution and path constraint solving. However, KUBO differs from previous works in that it focuses on each instrumented bug and analyzes its triggerability via an efficient backward inter-procedural analysis, enabled by on-demand call chain upwalk and per-function taint summaries. This analysis scales to the whole kernel and combats path explosion through actively pruning potential UBs that are not triggerable by userspace input.

D. Non-UB bug detection in kernel

KMiner [20] is a static analysis tool for detecting memory corruption bugs in the Linux kernel. It is based on value flow analysis and can reason about potential wrong operations on memory, e.g. free on specific memory objects. PEX [42] generates a whole-kernel call graph and uses static call chain analysis to detect privilege checking errors in the kernel. APISan [41] and Unisan [25] aims at finding API misuse. However, their intra-procedural analysis suffers from high false detection rates, especially when used for detecting integer overflows, due to the inability of tracking where and how API parameters flow into the API. Unlike these previous works, KUBO uses a scalable and efficient inter-procedural analysis and is focused on detecting critical UB that can be triggered by userspace inputs.

VIII. Conclusion

This paper presents KUBO, a precise and scalable static analysis framework to detect undefined behavior bugs in OS kernel. KUBO identifies UB bugs that are triggerable by userspace inputs. By using a novel inter-procedural analysis that tracks data and control dependencies across function calls, KUBO can produce highly precise results. By only analyzing paths that are directly affected by userspace inputs, and with the on-demand, incremental call chain analysis, KUBO can significantly reduce the number of paths to analyze. Going beyond triggering the UBs, KUBO also tracks the post-bug triggering paths in order to filter out UBs that have been handled and does not pose security implication to the system. KUBO can finish analyzing 27.8 million lines of code in the latest Linux kernel under 33 hours. In total, KUBO found 23 UBs,

including 17 critical ones, 14 of them are quickly accepted or patched.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments and Meng Xu for his releasing the DEADLINE source code and consultation during our development of KUBO prototype.

This project was supported by the National Science Foundation (Grant#: CNS-1748334) and the Office of Naval Research (Grant#: N00014-18-1-2660). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Common vulnerabilities and exposures about linux kernel. https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cpe_vendor=cpe%3A%2F%3Alinux&cpe_product=cpe%3A%2F%3Alinux%3Alinux_kernel. Accessed: 2020-6-18.

[2] Common vulnerability scoring system calculator cve-2018-13053. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?name=CVE-2018-13053&vector=AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:L&version=3.0&source=NIST.

[3] Cve-2018-8781. https://nvd.nist.gov/vuln/detail/CVE-2018-8781.

[4] Undefined behavior sanitizer - clang documentation. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks.

[5] The undefined behavior sanitizer - ubsan. https://www.kernel.org/doc/html/v4.14/dev-tools/ubsan.html.

[6] specification of integer overflow. https://www.gnu.org/software/autoconf/manual/autoconf-2.64/html_node/Integer-Overflow-Basics.html, 2020.

[7] syzkaller - kernel fuzzer. https://github.com/google/syzkaller, 2020.

[8] syzkaller - usb fuzzer. https://github.com/google/syzkaller/blob/master/docs/linux/external_fuzzing_usb.md, 2020.

[9] ANSI/ISO. Iso/iec 9899:2018. https://www.iso.org/standard/74528.html, 2018.

[10] BRUMLEY, D., SONG, D. X., CHIUEH, T. C., JOHNSON, R., AND LIN, H. Rich: Automatically protecting against integer-based vulnerabilities. In *Network & Distributed System Security Symposium* (2007).

[11] CADAR, C., DUNBAR, D., AND ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (USA, 2008), OSDI'08, USENIX Association, p. 209–224.

[12] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur. 12*, 2 (Dec. 2008).

[13] CHEN, Y., LI, P., XU, J., GUO, S., ZHOU, R., ZHANG, Y., WEI, T., AND LU, L. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, may 2020), IEEE Computer Society, pp. 2–2.

[14] CUI, H., HU, G., WU, J., AND YANG, J. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, Association for Computing Machinery, p. 329–342.

[15] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), C. R. Ramakrishnan and J. Rehof, Eds., Springer Berlin Heidelberg, pp. 337–340.

[16] DE MOURA, L., AND BJØRNER, N. Satisfiability modulo theories: Introduction and applications. *Commun. ACM 54*, 9 (Sept. 2011), 69–77.

[17] DIETZ, W., LI, P., REGEHR, J., AND ADVE, V. Understanding integer overflow in c/c++. *ACM Trans. Softw. Eng. Methodol. 25*, 1 (Dec. 2015).

[18] D'SILVA, V., PAYER, M., AND SONG, D. The correctness-security gap in compiler optimization. In *2015 IEEE Security and Privacy Workshops* (2015), pp. 73–87.

[19] ENGLER, D., AND DUNBAR, D. Under-constrained execution: Making automatic code destruction easy and scalable. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2007), ISSTA '07, Association for Computing Machinery, p. 1–4.

[20] GENS, D., SCHMITT, S., DAVI, L., AND SADEGHI, A.-R. K-miner: Uncovering memory corruption in linux. In *NDSS* (2018).

[21] HATHHORN, C., ELLISON, C., AND ROUNDEFINEDU, G. Defining the undefinedness of c. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, Association for Computing Machinery, p. 336–345.

[22] LEE, J., KIM, Y., SONG, Y., HUR, C.-K., DAS, S., MAJNEMER, D., REGEHR, J., AND LOPES, N. P. Taming undefined behavior in llvm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, Association for Computing Machinery, p. 633–647.

[23] LONG, F., SIDIROGLOU-DOUSKOS, S., KIM, D., AND RINARD, M. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2014), POPL '14, Association for Computing Machinery, p. 439–452.

[24] LU, K., AND HU, H. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2019), CCS '19, Association for Computing Machinery, p. 1867–1881.

[25] LU, K., SONG, C., KIM, T., AND LEE, W. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, Association for Computing Machinery, p. 920–932.

[26] MA, K.-K., YIT PHANG, K., FOSTER, J. S., AND HICKS, M. Directed symbolic execution. In *Static Analysis* (Berlin, Heidelberg, 2011), E. Yahav, Ed., Springer Berlin Heidelberg, pp. 95–111.

[27] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. Dr. checker: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Conference on Security Symposium* (USA, 2017), SEC'17, USENIX Association, p. 1007–1024.

[28] MUNTEAN, P., MONPERRUS, M., SUN, H., GROSSKLAGS, J., AND ECKERT, C. Intrepair: Informed repairing of integer overflows. *IEEE Transactions on Software Engineering* (2019), 1–1.

[29] POMONIS, M., PETSIOS, T., JEE, K., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Intflow: Improving the accuracy of arithmetic error detection using information flow tracking. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New York, NY, USA, 2014), ACSAC '14, Association for Computing Machinery, p. 416–425.

[30] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 49–64.

[31] SIDIROGLOU-DOUSKOS, S., LAHTINEN, E., RITTENHOUSE, N., PISELLI, P., LONG, F., KIM, D., AND RINARD, M. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, Association for Computing Machinery, p. 473–486.

[32] SONG, D., HETZELT, F., DAS, D., SPENSKY, C., NA, Y., VOLCKAERT, S., VIGNA, G., KRUEGEL, C., SEIFERT, J.-P., AND FRANZ, M. PeriScope: An effective probing and fuzzing

framework for the hardware-OS boundary. In *Network and Distributed System Security Symposium (NDSS)* (2019).

[33] SUN, H., ZHANG, X., ZHENG, Y., AND ZENG, Q. Inteq: Recognizing benign integer overflows via equivalence checking across multiple precisions. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, Association for Computing Machinery, p. 1051–1062.

[34] TORRES, C. F., SCHÜTTE, J., AND STATE, R. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference* (New York, NY, USA, 2018), ACSAC '18, Association for Computing Machinery, p. 664–676.

[35] WANG, T., WEI, T., LIN, Z., AND ZOU, W. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution.

[36] WANG, X., CHEN, H., CHEUNG, A., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Undefined behavior: What happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems* (New York, NY, USA, 2012), APSYS '12, Association for Computing Machinery.

[37] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Improving integer security for systems with kint. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (USA, 2012), OSDI'12, USENIX Association, p. 163–177.

[38] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 260–275.

[39] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. A differential approach to undefined behavior detection. *ACM Trans. Comput. Syst. 33*, 1 (Mar. 2015).

[40] XU, M., QIAN, C., LU, K., BACKES, M., AND KIM, T. Precise and scalable detection of double-fetch bugs in os kernels. pp. 661–678.

[41] YUN, I., MIN, C., SI, X., JANG, Y., KIM, T., AND NAIK, M. Apisan: Sanitizing api usages through semantic cross-checking. In *Proceedings of the 25th USENIX Conference on Security Symposium* (USA, 2016), SEC'16, USENIX Association, p. 363–378.

[42] ZHANG, T., SHEN, W., LEE, D., JUNG, C., AZAB, A. M., AND WANG, R. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 1205–1220.

[43] ZHANG, Y., SUN, X., DENG, Y., CHENG, L., ZENG, S., FU, Y., AND FENG, D. Improving accuracy of static integer overflow detection in binary. In *Research in Attacks, Intrusions, and Defenses* (Cham, 2015), H. Bos, F. Monrose, and G. Blanc, Eds., Springer International Publishing, pp. 247–269.