Progressive Scrutiny: Incremental Detection of UBI bugs in the Linux Kernel

Yizhuo Zhai*, Yu Hao*, Zheng Zhang*, Weiteng Chen*, Guoren Li *, Zhiyun Qian *, Chengyu Song *,
Manu Sridharan *, Srikanth V. Krishnamurthy *, Trent Jaeger †, Paul Yu ‡
* University of California, Riverside, †The Pennsylvania State University, ‡U.S. Army Research Laboratory
* {yzhai003, yhao016, zzhan173, wchen130, gli076}@ucr.edu, {csong, zhiyunq, manu, krish}@cs.ucr.edu,
† trj1@psu.edu, ‡ paul.l.yu.civ@mail.mil

***Abstract*—The Linux kernel has a rapid development cycle, with 10 commits every hour, on average. While these updates provide new features and bug fixes, they can also introduce new bugs and security vulnerabilities. Recent techniques showed how to detect some types of vulnerabilities using static analysis, but these tools cannot run quickly enough to keep up with the pace of kernel development. Ideally, an incremental analysis technique could address this problem, by doing a complete analysis once and then only analyzing changed portions of the code subsequently. However, incremental analysis of the Linux kernel poses unique challenges, due to its enormous scale and the high precision required to reduce false positives.**

In this paper, we design and implement INCRELUX, a novel Linux kernel incremental analysis tool. It allows rapid vulnerability detection after each update, via targeted analysis of the new code and affected prior code, and also speeds the tracking of pre-existing bugs to understand how long they have been present, thereby increasing awareness of such bugs. Our approach hinges on a bottom-up, function-summary-based approach, which leverages the benefits of a one-time clean-slate, but expensive analysis of a prior Linux baseline. INCRELUX also uses an effective heuristic to apply symbolic execution to incremental results to improve precision. Via extensive experiments on the challenging problem of finding use-before-initialization (UBI) bugs, we showcase a number of benefits of INCRELUX: (a) we can rapidly check if any new releases introduce UBI bugs and help eliminate them early in the process. (b) we perform a historical analysis to determine when a bug was first introduced and when it was fixed (a critical procedure in bug triage in the Linux kernel). (c) we compare the incremental analysis results with a clean-slate analysis and show our approach yields nearly the exact same results, demonstrating its fidelity in addition to efficiency. While the clean-slate analysis took 106.45 hours, the incremental analysis was often completed within minutes, achieving an average 200× speed-up for the mainline kernel and 440× on average, when analyzing stable branches.

I. INTRODUCTION

The Linux kernel has a fast paced evolution cycle, with 10 new commits on average, every hour. A new stable version is released about every two months [7]. While these updates provide new features and bug fixes, they can also introduce new bugs and security vulnerabilities.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24380
www.ndss-symposium.org

Due to the rapid development cycle, developers usually do not have time to conduct thorough security checks before committing new code. Unfortunately, once a bug is introduced, it can take a long time to catch the bug and fix it, especially for downstream distributions [36], [40], [78], [79]. For example, Cook [36] reported that the average lifetime of kernel bugs in Ubuntu (according to CVE tracker) from 2011 through 2016, is 3.3 years for critical bugs and 6.4 years for high-severity bugs. This provides ample time for adversaries to discover and exploit the vulnerabilities in the wild [3]–[5], [9], [47].

To alleviate the aforementioned security risks in the Linux kernel, both dynamic testing and static analysis have been applied. Dynamic testing, and fuzzing specifically [18], [31], [33], [39], [53], [55], [58], [59], [61], [69], is currently the most popular and effective approach to find bugs in the Linux kernel. With the state-of-the-art kernel fuzzer Syzkaller and the help of various sanitizers [25]–[27], thousands of bugs have been discovered in the past 4 years using the continuous fuzzing platform maintained by Google [28]. With regards to static analysis, many tools have been developed specifically for the Linux kernel, including commercial ones such as Coverity [19] and academic ones as in [13], [21], [23], [41], [43], [46], [60], [63], [64], [67], [70], [72], [75]. In practice, fuzzing is much more popular because it generates no false positives by design. Static analysis tools, on the other hand, often generate too many false positives in the pursue of soundness (i.e., no false negatives). To mitigate this problem, modern kernel static analyzers usually leverage more precise (field-, flow-, and context-sensitive) whole kernel analysis to reduce false positives.

One distinctive advantage of static analysis over dynamic testing is the code coverage—it does not require a concrete input to exercise the code to be analyzed. Therefore, static analysis has better potential to identify bugs in newly introduced code, where a corresponding input to trigger the code is usually missing. Unfortunately, to maintain decent precision, whole-kernel static analysis is often too expensive to be integrated into the rapid Linux kernel development cycle. For example, the state-of-the-art soundy static analysis tool Dr. Checker [46] needs minutes to analyze just a single driver (already with significant simplifications of the analysis). The state-of-the-art summary-based bottom-up analysis tool UBITect [75] needs a week to fully analyze the kernel. This makes them ill-suited for tight integration with the development cycle, as new commits and kernel versions arrive much more quickly than what the analysis can handle.

Given that the Linux kernel is huge and the changes are often localized in small pockets of the codebase, it is an ideal target to apply the kernel static analysis incrementally [56], on the changed portion only. Such an incremental analysis could dramatically reduce the analysis time, but without compromising the precision or the impact of the new changes on the whole kernel. This can bring several benefits for both kernel developers and maintainers. First, it enables a much quicker turnaround time for each analysis (e.g., applied before every minor version release and even in between), allowing (an otherwise infeasible) a precise and expensive static analysis to be integrated into the development cycle. Second, it enables quick validations of newly proposed patches. Currently, after a new patch is proposed, the kernel community heavily relies on manual inspection from peers (e.g., e-mail exchanges with maintainers for feedback) to spot potential bugs, which is both time consuming and error prone (as it is hard to reason about how the patch would affect other kernel components beyond the local scope). With an automated, whole kernel incremental analysis, a much more timely feedback can be provided, even before the patch is officially merged into the development branch. Third, because incremental analysis is based on static analysis, it provides an exhaustive coverage unlike what dynamic testing can offer.

Even though incremental analysis was conceptualized nearly 25 years ago [62] and has been applied recently in industry [19], [32], to the best of our knowledge, there is no publicly available tool that can be applied directly to the Linux kernel. In addition, few technical details are documented regarding the inner workings of [19], [32]. In this project, we develop a whole kernel incremental analysis framework, which we name INCRELUX. INCRELUX is flow-sensitive, field-sensitive, context-sensitive, and partially path-sensitive. Our choice is motivated by the publicly-available repository for "clean-slate" analysis of UBI bugs that was recently made available, and the relative difficulty of discovering such bugs in Linux [75]. Incrementalizing such an analysis poses particular challenges due to the scale and complexity of the kernel and the need for highly-precise analysis to reduce false positives. To facilitate the reproduction of results and further research, we open sourced our framework at https://github.com/seclab-ucr/IncreLux.

In this paper, we make the following contributions.

- **Design of incremental analysis.** We design INCRELUX which is an efficient and scalable tool to incrementally detect and track the evolution of use-before-initialization bugs in the Linux kernel. We document in detail how to turn a bottom-up summary-based static analysis into an incremental version.
- **Path-sensitive analysis of UBI bugs.** Due to the nature of UBI bugs being manifested only along certain paths, path-sensitive analysis is essential for a precise analysis. We show an effective technique for integrating path-sensitive symbolic execution into our incremental analysis that maintains scalability and empirically does not lead to missed warnings.
- **Measurement and evaluation.** Via evaluations of INCRELUX on the Linux kernel, we show that compared to the clean slate analysis which took one week to complete, it takes a significantly shorter time (depending on the situation, hundreds to thousands of times faster), detecting almost all the bugs that the clean slate approach would have found. In addition, we showcase the opportunity to catch bugs as soon as they are introduced, and timely confirmation on correct bug fixes.

II. BACKGROUND

In this section, we provide some brief background relevant to our work. First, we describe the workings of the rapid Linux kernel development cycle, which is the main motivation for our incremental analysis. Then, we describe the general concept of a bottom-up, summary-based static analysis, as well as prior work that does a whole-kernel clean-slate static analysis for detecting UBI bugs; this will lay the foundation for the design of our incremental analysis.

A. Linux Kernel Development

The Linux kernel is composed by tens of thousands of contributors around the world and has been customized for different usage scenarios. Thus, there are various actively-maintained kernel branches. Ubuntu and RedHat are two popular downstream distributions for desktops and servers. The Android Open Source Project (AOSP) also adopts the Linux kernel with some additional kernel features (e.g., the binder inter-process communication mechanism) and customized drivers. All such Linux distributions inherit code from the Linux upstream versions, including the Linux mainline and Linux stable / long-term-support (LTS) branches.

In our work, we focus on the Linux mainline [8] and stable versions. To be clear, there is a single Linux mainline branch where new features and bug fixes are continuously being added, while there are multiple Linux stable versions that are forked from the mainline and maintained separately. Typically, once a Linux stable branch is forked, no new features are added and only the necessary bug fixes are applied, hence the name "stable." Long-term support (LTS) branches are special stable branches that are maintained for much longer times. Mainline and stable versions adhere to the following versioning convention:

Major Versions. Major versions correspond to the Linux mainline. The version numbers are usually represented by $x.y$ (e.g., Linux 4.4). A new version (e.g., 4.5) is released roughly every two months. Compared to the immediately previous version, both new features and bug fixes (typically consisting of at least thousands of commits) could be present in the new version. It is critical to monitor the mainline branch because it contains all the features which are the main source of bug introduction.

Minor Versions. The Linux stable branches inherit the major version from the mainline and add a minor version (e.g., 4.4.12). From one minor version to the next (e.g., 4.4.13), only applicable bug fixes (as opposed to new features) from the mainline will be backported. These minor versions are important because downstream Linux distributions such as Ubuntu follow these stable or LTS branches (porting almost

all patches). It is important to check whether patches applied to stable branches indeed fix a bug.

Release Candidates. Release candidates refer to the candidates for the next major version in Linux mainline; each candidate has a suffix to the major version to indicate which release candidate it is, e.g., 4.4-rc1. The release candidates are released every week, representing intermediate states between major versions, which should also be analyzed.

B. Bottom Up, Summary-based Static Analysis

Scalability is often a challenge in performing static analysis on large codebases, especially the Linux kernel. Many static analysis tools for the kernel like Dr. Checker [46] are top-down. They start the analysis from an entry function (e.g., syscall entry), following its callees level-by-level. This means that many functions would have to be re-analyzed if they are invoked more than once. Bottom-up, summary-based analysis can avoid such redundant analysis of the same function. At a high level, it works by first building some program dependencies such as the call graph. Then the tool starts by analyzing the leaf functions (with no callees) and storing the analysis results for the function into a summary. Summaries are computed once and reused when analyzing all callers.

A few bottom-up static analysis tools have been developed in the literature, e.g., for Java [57], [73] and C [15], [51], [68], [75]. Some have been shown to be successfully applied to the Linux kernel [15], [68], [75]. Typically such analyses need to decide what kind of information to record in the summary, e.g., points-to information [51], locking behaviors [68], and data flow [15], [75].

C. UBITect: Summary-based Analysis for Detecting UBI Bugs

The Use-before-Initialization (UBI) bug is a kind of memory error caused by the use of uninitialized variables [6]. A use of an uninitialized variable is an undefined behavior. Importantly, UBI bugs in the Linux kernel can introduce serious security threats such as opening the door for arbitrary code execution [17] and information leakage [44], [49]. Previous work [45] has shown that UBI bugs are exploitable in an automated way, making their detection critical. To mitigate such threats, the Linux kernel added the `INIT_STACK_ALL` option to set uninitialized variables to a unified value viz., either zero or `0xAA`. However, Zhai et al. [75] argue that this method cannot fully eliminate such threats and since the correct initialization value is hard to infer, the best solution is detection and case-by-case patching. Based on this observation, they developed and open sourced UBITect, a clean-slate bottom-up, summary-based analysis. UBITect combines flow-sensitive static analysis and path-sensitive symbolic execution to perform a precise and scalable analysis for the Linux kernel.

Specifically, it constructs the global call graph for the whole kernel, i.e., the tool not only accounts for direct calls but also resolves UBI bugs that may carry over across indirect call relations. Based on this call graph, UBITect analyzes the leaf functions first; once these functions are analyzed, it summarizes the initialization and use behaviors of each of the arguments and return values of these functions.

The function summary primarily records two types of information, which serves as the contracts between the caller and the callee: 1) Requirements on the inputs (i.e., arguments) for the callee to be invoked safely. For example, in the context of detecting UBI bugs, the requirements of `memcpy` specify that all arguments (two pointers and the size) must be initialized; otherwise a UBI could happen. 2) Updates to outputs (including the return value and output arguments) after the invocation of the callee, with regards to the inputs. For example, in UBI bug detection, the updates of `memset` specify that the memory object point-to by the destination pointer will be initialized after the invocation; and the updates of `memcpy` specify that the memory object point-to by the destination pointer would have the same initialization status as the memory object point-to by the source pointer. These summaries are then provided to the callers of these leaf functions and the process continues, i.e., at each step, after all the callee functions are analyzed, the caller function uses these summaries to obtain the analysis result (instead of re-analyzing the callee functions again). In addition, warnings about potential UBI bugs, and some additional guidance for symbolic execution to assess the warnings, are generated during this process. Symbolic execution is then used to attempt to find a feasible path corresponding to each warning, leveraging the extra guidance to avoid exploration of certain irrelevant paths. A bug is reported if and when such a path is discovered. The symbolic execution step is necessary to filter false positives, because in the kernel, variable initialization and uses are often performed under correlated path conditions that the baseline analysis does not track.

An Example. Now we would like to present some necessary details with a simplified example in Figure 1. This example is taken from a real kernel UBI bug. There are two functions `stm32_dfsdm_irq()` and `regmap_read()` in this example, and `stm32_dfsdm_irq()` calls `regmap_read()`. UBITect will start its analysis from `regmap_read()`, and then generate the summary shown in Table I. The summary contains primarily two types of information for each variable: requirements and updates. The requirements describe what states are expected from the caller in order for the function to ensure no UBI bugs, whereas the updates describe the state updates of the variables after the function finishes executing. Variable `reg` and `val` are used in the if statement and the pointer dereferences respectively; therefore, to be free of UBI bugs, the requirements for these two arguments are `init`, meaning that callers should always pass in initialized variables. Regarding the updates, there are no assignments to `reg` and `val`, so their initialization status after the execution of `regmap_read()` will remain the same. For the object `val_obj` pointed to by `val`, it is initialized in only one branch but is left uninitialized in another branch (i.e., the error branch); therefore, after the caller calls `regmap_read()`, it is possible that the variable keeps the same initialization status as before. Therefore, to be conservative, the summary records there is no update to its initialization status. Finally, since `regmap_read()` returns a constant, either `0` or `-EINVAL`, the update of the return value is `init`. At the same time, since the branch `*var = some_init_number` would make the object of `var` to become initialized, UBITect adds this branch into the avoidlist of `var_obj` so that the symbolic execution later will avoid exploring this branch (when confirming a potential UBI bug). After having the summary for function `regmap_read()`, UBITect would analyze the

```
1  /* A simplified buggy code from
2   * drivers/iio/adc/stm32-dfsdm-adc.c
3   * uninteresting code lines are ommited
4   */
5  static irqreturn_t stm32_dfsdm_irq(int irq, void *arg)
6  {
7    struct stm32_dfsdm_adc *adc = arg;
8    unsigned int status, int_en;
9
10   regmap_read(DFSDM_ISR(adc->fl_id), &status);
11   regmap_read(DFSDM_CR2(adc->fl_id), &int_en);
12
13   if (status & DFSDM_ISR_ROVRF_MASK) {
14     if (int_en & DFSDM_CR2_ROVRIE_MASK)
15       //do sth here.
16   }
17
18   return IRQ_HANDLED;
19 }
20 int regmap_read(unsigned int reg, unsigned int *val)
21 {
22   if (reg)
23     return -EINVAL;
24
25   *val = some_init_number;
26   return 0;
27 }
```

Fig. 1: A piece of buggy code that adapted from a real UBI bug in the Linux kernel.

caller (i.e., `stm32_dfsdm_irq()`), when analyzing the function call at line 10 and line 11, instead of going into `regmap_read()`, the caller would look at the function summary in Table I. Using `status` as an example (`int_en` shares the same analysis step), the input `&status` corresponds to `val`, and `status` corresponds to `val_obj` in the function summary, respectively. The requirement for `val` is `init` and INCRELUX would check this before the function call; `&status` is an initialized variable, as it is the stack address and is therefore deemed to have met the requirements. The variable `status` would share the same status as there is no update for it; therefore, `status` remains uninitialized after the function call. Then, `status` is used in the `if` statement (line 13) after an `and` operation, and so INCRELUX reports a UBI bug here and pinpoints the uninitialized variable `status`. The warning contains the bitcode `drivers/iio/adc/stm32-dfsdm-adc.c`, the function `stm32_dfsdm_irq` that declared `status`, the id of the bug `driversiioadcstm32-dfsdm-adc.bc_stm32_dfsdm_irq_%status$obj$and$2`, the basic block for the use (line 13), and the avoidlist containing the basicblock in line 25.

After the warnings are generated, the symbolic execution would search for a feasible path from the declaration basic block of `status` (the block of line 8) to its (uninitialized) use. During the exploration, it would avoid the basic blocks (line 25) which initialize the variable. In this example, the feasible path exists if `reg` is equal to zero. The symbolic execution would report it as a true bug, and generate a detailed report containing the input and the path to trigger the bug. The bug report retains the information from the original warning (see the end of the previous paragraph). The same process would apply to `int_en` as well.

As shown by the authors of UBITect and verified by us, it takes over a week to do the whole kernel analysis to cover all the functions compiled in an allyes config. Motivated by the observation that function summaries are reusable, we posit that we can avoid analyzing the same function not only within a specific version, but across Linux releases as well. In particular, we can run the whole kernel analysis once as a clean-slate baseline, and then focus on analyzing only those functions that got changed. Based on this idea, we develop an incremental analysis to significantly reduce the analysis time of evolving Linux versions to detect bugs.

TABLE I: Function summary for function `regmap_read()`.

Argument	Requirements	Updates
reg	init	no
val	init	no
val_obj	no	no
regmap_read_ret	n/a	init

III. DESIGN OF INCRELUX

In this section, we present the design of our tool for an incremental analysis of the Linux kernel. We begin with an example to motivate the design. Then, we discuss the challenges that arise in conducting this incremental analysis. Subsequently, we describe specific design choices we make to address these challenges.

A. Motivating example

Prior to delving into the details of INCRELUX, we present a motivating example of a bug that was introduced in Linux v4.16-rc1. Compared with v4.15, v4.16-rc1 added a function `mlx5e_params_calculate_tx_min_inline()`, with 11 lines of code in total. Figure 2 depicts the details of the bug. The variable `min_inline_mode` will be left uninitialized if a query function `mlx5_query_nic_vport_min_inline()` inside the function `mlx5_query_min_inline()` fails. However, both the function `mlx5_query_nic_vport_min_inline()` and the caller function `mlx5e_params_calculate_tx_min_inline()` use this variable directly without any return value check.

To detect this bug, an interprocedual holistic program analysis is necessary as there are three functions involved. For the prior work UBITect to catch this bug in v4.16-rc1 (the major version immediately after v4.15), it would need to construct the global call graph, and beginning with the leaf functions, generate summaries and continue the analysis upwards to the callers. Based on the function summary of `mlx5_query_min_inline()`, the caller could infer that the variable `min_inline_mode` might be uninitialized and upon the use on line 9, generate a warning and begin a check using symbolic execution for verifying the path feasibility. The observation here is that the functions `mlx5_query_min_inline()`, `mlx5_query_nic_vport_min_inline()` and other low level functions are unchanged from version v4.15. Therefore, if we still have the function summaries for these, then we can significantly expedite the analysis by analyzing only the new added function `mlx5e_params_calculate_tx_min_inline()`. This observation motivates us to reuse function summaries not only in analyzing a single kernel version (as what UBITect did), but across Linux versions that have a large overlap in code. After obtaining summaries from the clean-slate whole program analysis (WPA), when analyzing a new version of the Linux kernel code, we reuse summaries aggressively,

```
/* drivers/net/ethernet/mellanox/mlx5/core/en_common.c
 * uninteresting code lines are ommited
 */
+ u8 mlx5e_params_calculate_tx_min_inline(
+                        struct mlx5_core_dev *mdev)
+ {
+       u8 min_inline_mode;
+
+       mlx5_query_min_inline(mdev, &min_inline_mode);
+       if (min_inline_mode == MLX5_INLINE_MODE_NONE)
+               //do something here
+       return min_inline_mode;
+ }

/* drivers/net/ethernet/mellanox/mlx5/core/vport.c
 * uninteresting code lines are ommited
 */
void mlx5_query_min_inline(struct mlx5_core_dev *mdev,
                           u8 *min_inline_mode)
{
        switch (MLX5_CAP_ETH(mdev, wqe_inline_mode)) {
        case MLX5_CAP_INLINE_MODE_VPORT_CONTEXT:
                mlx5_query_nic_vport_min_inline(mdev, 0,
                                                min_inline_mode);
                break;
        }
}
int mlx5_query_nic_vport_min_inline(
                        struct mlx5_core_dev *mdev,
                        u16 vport, u8 *min_inline)
{
        u32 out[MLX5_ST_SZ_DW(
                query_nic_vport_context_out)] = {0};
        int err;

        err = mlx5_query_nic_vport_context(mdev, v
                           port, out, sizeof(out));
        if (!err)
                *min_inline = MLX5_GET(
                    query_nic_vport_context_out, out,
                    nic_vport_context.min_wqe_inline_mode);
        return err;
}
```

Fig. 2: A use before initialization bug introduced in v4.16-rc1. Lines with the + sign indicate those that are added because of the new function that was introduced.

only re-analyzing code that is affected by any modifications. Progressively, as new versions are released, we can in this way, incrementally analyze only code changes in each subsequent version.

B. Considerations

In designing INCRELUX, we need to consider the following three points:

Consideration 1: Correctness compared to the whole-program analysis. Incremental analysis should yield the same result as if each version were analyzed from scratch. Otherwise, the incremental analysis may miss critical bugs. Correctness boils down to how we determine which part of the program can be safely skipped during incremental analysis. First, all modified code and newly added functions should be analyzed. Functions must also be re-analyzed if their analysis results depend on a function whose summary has changed. Finally, symbolic execution should be re-run for warnings that may have been impacted by code changes.

Consideration 2: Function summary design and re-analysis scope. As mentioned above, when a function summary changes, we need to re-analyze any function whose analysis results depend on the summary. Therefore, what to include in a function summary have important implications on scalability and accuracy. In UBITect, function summaries contain information limited to how the caller/callee may directly interact with each other based on arguments and return values; so, a change to the summary only affects callers of the function. Technically, the summary could also include other dependencies, such as indirect interactions through global states (e.g., global variables). For example, the modification of a global variable (e.g., from initialized to uninitialized) in one function can potentially affect many subsequently invoked functions — not only callees but also functions invoked from a different concurrent syscall. Capturing such dependencies can lead to more accurate results; but on the other hand, this can potentially lead to a much larger "radius of changes" and make the incremental analysis less scalable. We note that the goal of our incremental analysis is not to improve the accuracy of the underlying static analysis, but to ensure that the analysis results will be identical to the clean-slate analysis. Therefore, the task for our incremental is not to re-design the summary used by the underlying analysis, but to make sure all code that needs to be re-analyzed will be included. Fortunately, as a summary-based bottom-up analysis already needs to calculate the dependencies between functions, we can just reuse the same dependency graph to calculate the re-analysis scope. For example, our prototype re-uses the same call-graph analysis of UBITect to identify the scope of functions that need to be re-analyzed.

Consideration 3: Extensibility. Even though we focus on UBI bugs, we aim for a design that can be extended with relative ease to catch other types of bugs. Indeed, we make the observation that the UBI bugs are fundamentally bugs that can be discovered by data flow analysis. In fact, it can be viewed as a taint analysis problem, where an uninitialized variable can be considered a taint source, and any use of the tainted variable can be considered the sink (e.g., arithmetic operation or deference of a tainted pointer). Therefore, we note that the summary used in UBITect can be easily adapted to store taint information representing other semantic information (e.g., inputs from userspace [34], [46]). Using Dr. Checker's loop bounds checker [46] as an example. It checks if userspace data is used as loop bounds, which may lead to out-of-bound accesses (i.e., buffer overflows). To do so, Dr. Checker defines all user inputs from syscalls as `tainted` variables; then along with the top-down data-flow analysis, it propagates the taint tag to other variables to identify the use of any tainted variable as loop bound. To fit it in a bottom-up style analysis, we can reuse the same semantics of `tainted` (userspace data) and `untained` (non-userspace data). Then in the function summary, we can require that if an input of the function will affect a loop bound, then the caller must pass an `untainted` argument for safe invocation. For updates, UBITect's current summary (subsection II-C) already includes how inputs would affect the outputs (i.e., the taint propagation from inputs to the outputs). For instance, if the function includes a statement `retvalue = input0;`, then we will record in the function summary as "the tainted status of `retvalue` will be changed to whatever the taint status `input0` is."

C. System Overview

Having described key design considerations, we next provide an overview of INCRELUX with a high-level workflow depicted in Figure 3. In particular, the workflow includes a

few pre-processing steps followed by the main incremental analysis.

We assume that function summaries have already been computed for some previous kernel version via a whole-program analysis. When a new kernel version needs to be analyzed, we first perform a simple diff with the previous version to figure out which functions have been changed. Second, we perform a dependency analysis to calculate the dependency graph between functions. Specific to detecting UBI bugs, we perform a global call graph analysis on the new kernel version (including indirect call resolution). Note that we have not attempted to perform the call graph analysis incrementally as the call graph analysis is relatively inexpensive anyways. We also compute strongly connected components (SCC) of the call graph [75] so that we can be ready for fixed point analysis. Finally, we proceed to the actual incremental analysis which will be described next. This step combines the static analysis and the under-constrained symbolic execution(UCSE), the different bugs set between the previous version and the current version would be generated.

D. Bottom-up, Summary-based Incremental Analysis

We use a typical worklist algorithm to perform the bottom-up summary-based incremental analysis (algorithm 1). Specifically, the inputs include the call graph (`CG`), the old version code (`OC`), the new version code (`NC`) and all the function summaries for the old code (`OldSum`). `OC` and `NC` represent the source codes of the two kernel versions in their entirety. The analysis first initializes the worklist with modified/new functions (`DiffSet`). For each function in the worklist, INCRELUX computes a new summary (`newFSum`) for it. If the new summary differs from that of the previous version (`(OldFSum)`), or the previous summary does not exist, INCRELUX then adds all the callers of the current function to the worklist for further analysis. The process terminates when the worklist is empty. The algorithm produces two sets of outputs viz., the new function summaries `newFSumSet` which replace the old summaries, and the set of warnings `WS`. The rules to generate the warings are the same as in [75] (subsection II-C) for consistency.

In the pseudocode, `get_diff_functions()` is used to automatically extract the set of functions differing in the old and new versions. After extracting these diff functions, INCRELUX computes a bottom-up analysis ordering using `get_order()`, based on a topological sort of the call graph. The results are stored in `SCCs`. Each item in `SCCs` is a set of functions; if there is more than one function in the set, then each of the functions can reach the others in the set via a call chain. If there is only one function in the set, it could be a recursive function or a function that is not involved in any loop in the callgraph. Note that these sets are ordered in a bottom-up style, i.e., the function sets that are close to the leaves appear before functions that are closer to the root.

After INCRELUX computes the order in which the functions are to be analyzed, starting from the modified functions in `DiffSet`, it follows the callgraph and conducts the bottom-up analysis. If the summary of a modified function is changed, then all its callers are reanalyzed, and iteratively INCRELUX checks if their summaries change and so on. Note that we will perform a fixed point analysis for each SCC, which means that sometimes the same function can be analyzed multiple times until their summaries converge. Once no more summary changes are left, we go back and start from the next function in `DiffSet`. After all functions in `DiffSet` are processed, we can be sure that all necessary functions have been re-analyzed, and all the function summary changes should have been collected completely.

Algorithm 1: Bottom-Up, Summary-Based, Incremental Analysis

```
Input: CG: Callgraph;
       OC: Old version code;
       NC: New version code;
       OldSum: Old Function Summary
Output: newFSumSet: New Function Summary;
        WS: Warning Set
1  DiffSet=get_diff_functions(OC, NC);
2  SCCs=get_order(CG);
3  foreach SC ∈ SCCs do
4      worklist ← ∅;
5      foreach func ∈ SC do
6          if func in DiffSet then
7              worklist.push(func);
8      while is_not_empty(worklist) do
9          func = worklist.front() ;
10         worklist.remove(func);
11         oldFSum = get_old_sum(OC, func);
12         (warnings, newFSum) = analyze_funcs(func);
13         WS.append(warnings);
14         newFSumSet.append(newFSum);
15         if not_equal(oldFSum, newFSum) then
16             callerSet ←get_callers(func);
17             foreach caller ∈ callerSet do
18                 if caller ∉ worklist and caller ∈ SC
                   then
19                     worklist.push(caller);
20                     continue;
21                 if caller ∉ worklist then
22                     DiffSet.push(caller);
```

Tracking Warnings Changes. After the above steps, warnings are generated automatically. We retain the rules for generating warnings from [75] for consistency. Furthermore, since we are now analyzing multiple versions of kernels, we now need to track the warnings across versions. We place warnings in three categories:

1) Disappearing – these disappear in the new version.
2) Remaining – these remain across the old and the new versions.
3) New – these are introduced in the new version.

We will describe in §IV, how the categorization is performed.

Under-Constrained Symbolic Execution (UCSE). As mentioned before, UBITect applies under constrained symbolic execution on the reported warnings to confirm whether there

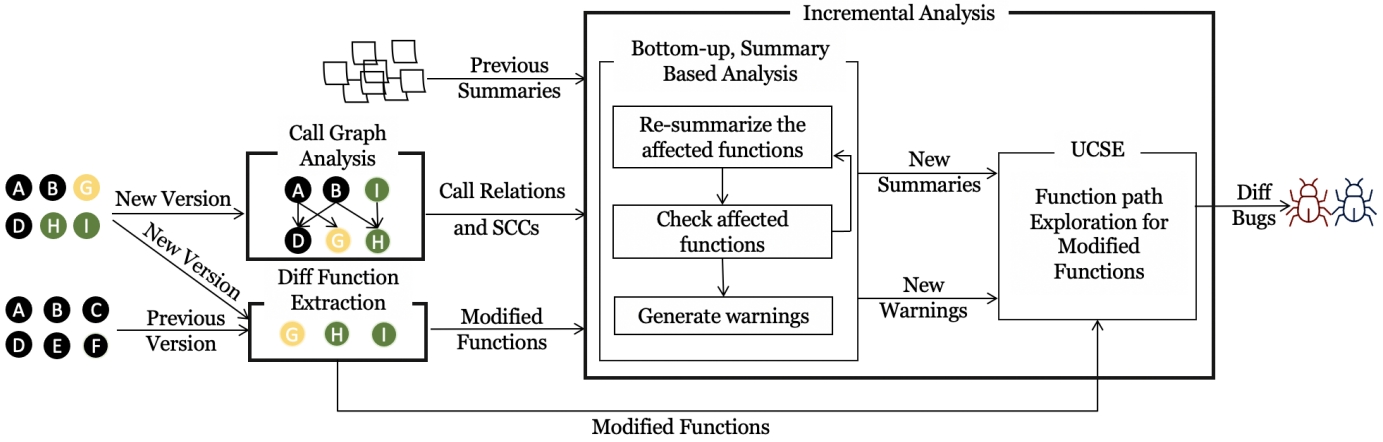

Fig. 3: Upon obtaining a new version, INCRELUX leverages function summaries that were previously computed (in a clean slate version or a prior incremental analysis) to do an expedited incremental analysis. First, warnings relating to potential UBI bugs are generated and then under-constrained symbolic execution is applied to find a potential feasible path to triger the bug.

is a feasible path that leads to the potential bug. The bug is reported only if UCSE finds a feasible path. Otherwise, the warning is filtered.

In the original UBITect, symbolic execution is applied to all the warnings reported. However, INCRELUX applies symbolic execution to only new warnings or those existing warnings whose associated functions are re-analyzed. A function is associated with a warning if it is a transitive caller or callee of the function containing the warning's variable declaration; these are the functions that could possibly affect path feasibility. If all these associated functions are unchanged, then the path feasibility of the warning cannot change. However, if some associated function has been changed (diffed), even if its summary remains the same, we conservatively run symbolic execution for the warning again, in case the path feasibility is influenced.

Note that here we do not consider the influence of global states. In theory, if the newer kernel version gives a global variable a different value, it could affect the warning if the global variable is used somewhere along the path. Nevertheless, since our symbolic execution is under-constrained, by design, we do not capture the constraints for global variables that are modified outside of the scope of our analysis.

There is an additional issue regarding whether INCRELUX's UCSE component yields the same results as what is obtained by running the process from scratch (clean-slate). Due to the non-determinisms in KLEE (e.g., path scheduling), INCRELUX cannot fully guarantee identical results. However, we argue that this is not a fundamental limitation of INCRELUX as the same non-determinisms would also affect two different runs of the same clean-slate analysis. Moreover, we show in our evaluation (section V) that in practice this case is very rare.

IV. IMPLEMENTATION

In this section, we describe the implementation of INCRELUX. We implement INCRELUX on top of UBITect, which was developed based on LLVM-7.0.0. We ported it to LLVM-9.0.0, which has a better support of Linux kernel (e.g., supporting asm goto). The main functionality we added includes the diff function extraction, function summary adaptation, logic to reuse function summaries from previous versions, and logic for under-constrained symbolic execution on warnings. We use KLEE as our symbolic execution engine, and the boost library to serialize the function summaries to the disk. We will describe a few aspects to help explain some of the details left out in the design section.

Compiling kernel source code. To generate the IR bitcode files from the Linux source code, we use -O0 optimization level and enable debug information (-g), to ensure we have source location and variable name information required for identifying bugs. In addition, the unoptimized LLVM IR is generally easier to analyze compared to -O1 and -O2.

Indirect Call Resolution. There are generally three types of indirect call resolution techniques that are widely used for static analysis of the Linux kernel: the pointer analysis from KINT [64], the type-based analysis from Unisan [44], and a hybrid of these two methods used in multi-layer type-based analysis [42]. We chose the points-to based algorithm from [64]. This is because the other two type-based methods lead to a large number indirect call targets, which causes a significant bloating of some strongly connected components, leading to much longer analysis time. The downside of using the points-based algorithm is that it may miss valid indirect call targets. We plan to investigate ways to leverage the type-based methods and yet still be able to break the strongly connected components (we suspect they should have been much smaller).

Diff Function Extraction. There are two possible sources for function changes. The first is from direct modifications in the source code; these changes can be caught easily by the `diff` tool. Another source of changes is addition or deletion of entire files, typically reflected in a change to some `Makefile`. Both types of changes are fully supported by INCRELUX.

UBI warning detector. Given that we currently support only the UBI bugs, we follow the same rules that were used in UBITect [75] to detect UBI bugs, i.e., any use of

variables that are uninitialized. No changes are needed in the incremental analysis because the analysis follows largely the same procedure except that it ignores the vast majority of unchanged functions. However, as mentioned earlier in §III, it is possible to add additional taint-style bug detectors, e.g., integer overflow, which we leave as future work.

Guided Symbolic Execution. We have described how to apply symbolic execution in an incremental fashion in §III, by avoiding re-execution on the cases where functions do not change at all. Nevertheless, symbolic execution is extremely expensive as we can still face the path explosion problem, especially when considering that the number of warnings can be large as the static analysis is flow-sensitive only (instead of path-sensitive). Therefore, we choose to limit the time and memory usage of each warning to 10 minutes and 2GB. The thresholds are decided empirically based on a small-scale experiments (sampled warnings) with much loose limit (12 hours and 4 GB). Basically, the results of small-scale experiments showed that 90% of the warnings finish within 10 minutes, consuming at most 2GB. Note that UBITect used only a 2-min time limit which will yield fewer confirmed bugs according to our analysis.

Bug Identification and Tracking Across Versions. Given that there are often multiple warnings associated with the same uninitialized variable, e.g., multiple uses of the same uninitialized variable, we decide to group warnings that share the same associated variable name (including the field name if the uninitialized variable is a part of a struct) into a bug. Furthermore, given that we are interested in understanding the lifetime of a bug, we simply consider bugs in two different kernel versions that share same variable name to be the same bug. This is a reasonable approach become the exact warnings may look different on different kernel versions, and yet they are highly likely sharing the same root cause of failing to initialize the same variable.

V. Evaluations

To evaluate the efficacy of INCRELUX and demonstrate the benefits of incremental analysis, we perform a large-scale analysis from Linux kernel v4.15-rc1 mainline, progressively to Linux v4.19, covering one year worth of development period. In addition, we analyzed the stable version (a long-term-support version) of v4.14.y that spans over three years, and v4.15.y that spans over three months. In total, we have analyzed 46 mainline and 28 stable versions. To see how well the incremental analysis would work when the amount of changes is significant, we also analyzed v5.4 using the base of v4.19, and v5.9 using the base of v5.4. All kernel versions are analyzed using `allyesconfig`, i.e., most of the non-conflicting configuration options are set to "yes" and the corresponding features will be analyzed. Then, we present the results from applying INCRELUX, including the speed improvement, new bug discovery, patch confirmation, and equivalence analysis (i.e., to check if the results are consistent between incremental and whole-program analysis). We conduct our static analysis on a server with Intel Xeon E5-2697v3 CPU and 157G RAM, 160G swapping, running Ubuntu 20.04. All experiments use the `-O0` optimization level to compile the kernel to LLVM IR. The symbolic execution experiments were run on a machine with Intel Xeon CPU E5-2698v4 CPU cores and 256G RAM.

A. Evaluation Scope

According to the Linux kernel development guide [38], once a stable version (i.e., denoted by a new major version such as v4.14) is released, a two-week window called the "merge window" is open for the next stable version. During these two weeks, all feature changes and bug fixes are allowed. After two weeks, a series of release candidate (with suffix -rc and a number) are published weekly to stabilize the version. Starting from rc1, only regression fixes or entirely new drivers can be added. Once the kernel is sufficiently stable, a new stable version is released and the two-week "merge window" is opened again for the next stable version. Due to this development process, the first release candidate (rc1) usually has significantly more code changes than later release candidates. For example, there are 23,941 functions modified or newly added in v4.15-rc1 compared to v4.14, but only 719 functions in the subsequent v4.15-rc2.

As mentioned in §II, once a stable version (identified by the major version number such as 4.14) is released, it is forked into an independent branch for maintenance (identified by minor version number such as 4.14.1). Each minor version consists of relatively small number of bug fixes only (no feature additions). For example, on average, 102 functions are modified or added to the v4.15 branch between two consecutive minor versions.

B. Speed Improvement Analysis

Incremental Analysis for the Mainline Versions. Table II shows the key experimental results for the mainline analysis. Due to space constraints, we leave the results for v4.17-rc1 to v4.19 to Appendix , which are consistent with the results in earlier versions.

As mentioned, typically, rc1 releases contain a large number of changes, as they include changes from the "merge window" where new features are accepted [1]. Despite the large number of changes we still see an almost 3× speed up in analyzing the rc1 versions as compared to the initial exhaustive analysis. For versions with fewer changes the speed ups are much more dramatic, ranging from 31× to 937×.

For the experiment that "stress tests" our incremental analysis with major changes from v4.19 to v5.4 and from v5.4 to v5.9. Compared to the 106.75 hours baseline clean-slate analysis time, the incremental analysis from v4.19 to v5.4 took 97.9 hours, and from v5.4 to v5.9 took 99.65 hours. The results indicate that incremental analysis does not yield benefits when changes are significant. This is expected because the version gaps represent a whole year worth of development effort, with 90K functions modified (compared to the 625K functions in total in v4.14). Consequently, INCRELUX re-analyzed 205,327 functions for v5.4, and 197,413 functions for v5.9. We suggest doing the clean slate analysis for such big changes.

We note that as the distance between versions increases, the number of functions that are to be analyzed grows, and the benefits of INCRELUX diminish; our expectation is that

TABLE II: Incremental analysis results for mainline versions v4.14 to v4.16. Please refer to the Appendix for the full table from v4.14 to v5.9. **T(h)** : Total analysis time in hours. **SU** : Speedup compared to exhaustive analysis of v4.14. **FM** : Number of functions modified compared to the immediate. predecessor. **FR** : Number of functions (re-)analyzed in this version. **Warn** : Number of Warnings reported in the current version. **Disappearing** : Number of warnings that disappear in the current version (compared with the last analyzed version). **Equal** : Number of warnings that remain in the current version compared to the immediate previous analyzed version. **New** : Number of warnings newly introduced in the current version compared with the last analyzed version. **SE-New** : New bugs confirmed by SE that are introduced in the new version.

versions	T(h)	SU	FM	FR	Warn	Disappearing	Remaining	New	SE-New
v4.14	106.75	N/A	N/A	629862	103616	N/A	N/A	N/A	622
v4.15-rc1	28.26	3.78	23941	42548	21190	4325	99291	4979	69
v4.15-rc2	2.91	36.74	719	2617	2190	422	103848	211	0
v4.15-rc3	2.27	47.04	656	3084	1937	301	103758	238	0
v4.15-rc4	1.13	94.52	332	1268	718	116	103880	70	0
v4.15-rc5	1.28	83.31	329	1793	1339	207	103743	331	0
v4.15-rc6	1.15	92.6	273	1761	1282	96	103978	96	0
v4.15-rc7	0.43	248.74	101	403	263	21	104053	27	0
v4.15-rc8	1.33	80.15	243	1707	1305	114	103966	151	0
v4.15-rc9	0.63	169.82	217	1031	696	49	104068	48	0
v4.15	0.13	800.63	122	262	215	36	104080	48	2
v4.16-rc1	26.77	3.99	21251	52151	26922	4240	99888	4742	55
v4.16-rc2	0.24	453.72	220	521	342	10	104620	25	0
v4.16-rc3	0.7	151.6	434	1012	713	136	104509	77	1
v4.16-rc4	3.39	31.48	278	7398	3446	502	104084	509	4
v4.16-rc5	0.76	141.23	422	979	598	80	104513	76	0
v4.16-rc6	0.26	415.01	144	401	279	15	104574	27	0
v4.16-rc7	0.93	115.2	498	1543	819	107	104494	190	2
v4.16	0.33	318.92	183	535	278	18	104666	15	0
v4.17rc1-v4.19	...	...	...	...	...	...	...	...	...
v5.4	97.9	1.09	99370	205327	158018	43762	69652	88366	N/A
v5.9	99.65	1.07	91741	197413	152746	40720	85425	67321	N/A

INCRELUX will be applied over nearby versions so that a continuous process of analysis and bug finding is viable.

Incremental Analysis for Stable Versions. Table III and Table IV show the incremental analysis results for stable v4.14 and v4.15 kernel versions, respectively. As v4.14 is a long-term-support branch and has more than 200 minor versions released, we sampled and only ran incremental analysis for every 20 minor versions. For v4.15 we analyzed all the minor versions till the end of the branch (i.e., v4.15.18). For these kernel versions, we again see impressive analysis speedups. In fact, for the v4.15 versions, since the number of changes in each version is quite small, we see enormous speedups (up to 2,260×), and we display the analysis time in Table IV in seconds rather than hours.

Relations between the number of functions reanalyzed and the time that the incremental analysis incurs. To confirm the factors which affect the analysis time, we draw the relations between the number of functions analyzed and the analysis time in Figure 4. As we can see, the accumulated analysis time is proportional to the number of analyzed functions.

Incremental Analysis for Each Patch. Our incremental analysis can also be used to perform regression check for individual commits. This can be useful for individual developers who might be submitting commits to get quick feedback on whether their changes would introduce new UBI bugs, or fix existing ones (if they are submitting patches), without having to wait for the much slower feedback from peers or the automated fuzz testing results. Specifically, we extract a few patches that fixes UBI bugs reported in prior work [75]. We performed the incremental analysis for each patch (using its immediate predecessor commit as the baseline), and the results are shown in Table V. We can see incremental analysis quickly finishes checking for each patch, showing that the intended bug was fixed and did not add new UBI bugs. Except for one patch that took 32.46s to finish, other patches were checked within 5.46s, and the average time for checking was ∼5.01s. We believe this instantaneous feedback can be extremely valuable for kernel development.

To summarize, our experiments show that INCRELUX yields substantial speedups in a variety of scenarios. Even for an rc1 release with more than 20k function changes, INCRELUX runs faster than an exhaustive analysis. For changes touching fewer functions, the incremental analysis can run in minutes or even seconds. This efficiency of analysis enables new possibilities, like immediate testing of patches before merging.

C. Time Breakdown

In Figure 5, we took the analysis results from v4.14 to v4.15-rc1 as an example to show the time breakdown of our incremental static analysis. The first step is to construct the call graph, which takes a few minutes — we currently do not attempt to incrementally construct the call graph as it is not a bottleneck. For each function, INCRELUX follows the same analysis step in [75]: points-to analysis, alias set generation, qualifier inference, and the summary generation. We see that the most time costly phase is the alias set generation (14.42 hours), followed by the points-to analysis (6.13 hours), the qualifier analysis and function summary generation take a

TABLE III: The incremental analysis result of v4.14 stable, we sampled every 20 versions.

versions	T(h)	SU	FM	FR	Warn	Disappearing	Remaining	New	SE-New
V4.14	106h45min	N/A	N/A	629862	103616	N/A	N/A	N/A	622
v4.14.20	3.93	27.18	2123	2519	1358	235	103381	257	12
v4.14.40	5.92	18.02	2079	3289	2334	350	103288	230	2
v4.14.60	5.25	20.33	2096	3033	2079	384	103134	351	4
v4.14.80	6.43	16.6	1879	3934	3021	726	102759	470	6
v4.14.100	9.67	11.04	1812	6029	3222	462	102767	340	3
v4.14.120	3.35	31.86	1894	2700	6026	195	102912	235	2
v4.14.140	4.91	21.75	1565	2843	3079	534	107213	1249	0
v4.14.160	7.29	14.65	2354	4824	3042	1091	107371	463	2
v4.14.180	8.05	13.26	2336	3798	3289	1907	105927	641	0
v4.14.200	5.73	18.63	1773	3841	3218	1098	105470	385	0
v4.14.220	3.31	32.29	1221	2252	1343	250	105605	123	1

TABLE IV: The incremental analysis result of stable v4.15.

versions	T(s)	SU	FM	FR	Warn	Disappearing	Remaining	New	SE-New
v4.14	106.75h	N/A	N/A	629862	103616	N/A	N/A	N/A	622
v4.15rc1-v4.15-rc9	...	...	...	...	...	...	...	...	...
v4.15	468	800.63	122	262	215	36	104080	48	2
v4.15.1	559	687.48	56	100	61	6	104122	4	1
v4.15.2	622	617.85	73	93	47	6	104120	24	0
v4.15.3	170	2260.6	29	77	33	2	104142	2	1
v4.15.4	4864	79.01	268	804	492	46	104098	40	1
v4.15.5	2121	181.19	157	301	205	63	104075	23	0
v4.15.6	947	405.81	55	184	179	13	104085	11	0
v4.15.7	1007	381.63	65	119	131	20	104076	59	2
v4.15.8	8151	47.15	146	1136	1056	260	103875	158	0
v4.15.9	613	626.92	22	80	118	15	104018	13	0
v4.15.10	3866	99.41	175	663	247	18	104013	18	3
v4.15.11	11760	32.68	122	4482	2941	168	103863	147	0
v4.15.12	939	409.27	62	130	69	2	104008	1	0
v4.15.13	2592	148.26	86	540	243	11	103998	15	0
v4.15.14	855	449.47	109	253	189	4	104009	5	0
v4.15.15	1787	215.05	52	377	81	11	104003	9	0
v4.15.16	522	736.21	73	98	73	5	104007	8	1
v4.15.17	2934	130.98	180	630	815	146	103869	95	1
v4.15.18	920	417.72	72	150	109	23	103941	16	2

TABLE V: The incremental analysis for patches from UBITect. **T(s)** : Time in seconds. **FM** : Number of functions modified compared to predecessor. **FR** : Number of function (re-)analyzed after the patch.

commit #	T(s)	FM	FR
4674686d6c897	32.46	1	12
0fb68ce02ae73	0.31	1	1
e20bfeb0b7d80	1.01	1	1
4a8191aa9e057	5.46	1	4
8c3590de0a378	0.64	1	3
e33b4325e60e1	2.17	1	3
1252b283141f0	0.85	1	1
53de429f4e88f	0.18	1	2
472b39c3d1bba	2.03	1	1

small portion of the incremental analysis, 4.07 hours and 0.05 hours, respectively. INCRELUX also took an additional 3.48 hours to generate bug reports and serialize function summaries to the disk.

Finally, in addition to the time breakdown for different phases, we also look at the variations across analyzing different functions. We find that 31,926 out of 42,548 functions (75%) are analyzed within 1s (for four phases combined), while 40,888 functions (96%) can be finished within 10s, only 3 functions take more than 1,000 seconds to finish where the most time consuming functions took 1 hour to finish. We did not calculate the time for symbolic execution here as we impose a time budget for each warning.

D. Correctness/Equivalence Analysis

A key requirement of incremental analysis is that it yields the same results as the clean-slate whole-program analysis (WPA). Towards evaluating this requirement, we perform the WPA for Linux v4.14.20 and v4.15 and then compare the bugs reported with those reported by INCRELUX. The results show that the same warning sets are obtained i.e., INCRELUX is able to obtain the same results as the WPA. However, in the warning validation phase, due to KLEE's non-determinisms (e.g., path scheduling), the bug set varies slightly after the symbolic execution. For example, with v4.15, 634 warning from INCRELUX are confirmed as bugs, while 635 warning from WPA are confirmed. In particular, all of the 634 bugs from INCRELUX were present in the results from WPA (leaving 1 bug being different). This is an insignificant difference. Given the speedup that INCRELUX provides, we believe that this is a very compelling result. More importantly, we note that the non-determinisms in KLEE does not only affect INCRELUX—even two difference runs of the WPA could generate different results. To mitigate the non-determinism, one could provision more resources or develop better heuristics for pruning the path exploration.

(a) The incremental results from the v4.15-rc1 to v4.19, it plots the relations between the number of functions reanalyzed and the analysis time in hours.

(b) The incremental results for sampled minor versions of v4.14, it plots the relations beween the number of functions reanalyzed and the analysis time in hour.

(c) The incremental results for minor versions of v4.15, it plots the relations between the number of functions reanalyzed and the analysis time in second.

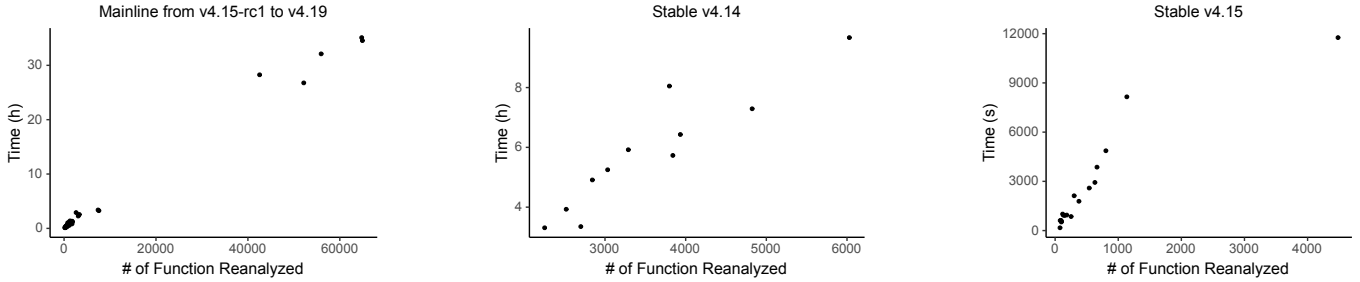

Fig. 4: The incremental results for different versions.

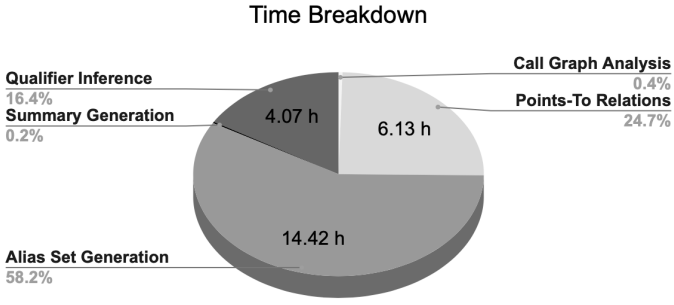

Fig. 5: The time distributions for different analysis phase along the incremental analysis from v4.14 to v4.15-rc1.

E. Bug Finding Results

Reported warnings. We first present the results of new UBI bugs found as we analyze the mainline version from v4.14 to v4.19. Since UBITect has been applied on v.4.14 already, we look at only the new ones found by the incremental analysis. Given that the analysis results can come out extremely fast, we can catch the bugs when they are introduced in candidate release versions and prevent them from slipping through the production versions. In our evaluation, we randomly sample 44 bugs reported by INCRELUX, and find 22 true positives, all of which, turned out to have been introduced in the first release candidates. 5 of the 22 true positives are dismissed by maintainers, as the conditions to trigger the bug cannot be satisfied in reality (e.g., a failure of PCI config read). For the rest 17 cases, we found that 7 are fixed later on in mainline; and 10 bugs are still unpatched by the time of our reporting. We have reported all unpatched bugs to the maintainers; 5 have been confirmed; but the remaining 5 are still awaiting maintainers' responses. The bugs are listed in Table VI.

False Positives and False Negatives. 22 out of 44 reported bugs (50%) turn out to be false positives. Our manual inspection revealed that 14 are caused by the incomplete guidance generated by the static analysis, 7 are caused by imprecise indirect call analysis (missing indirect call), and 1 due to the approximation of array. To evaluate the false negatives, we need to obtain another set of UBI bugs with ground truth. First, we use the keywords "uninit" and "Uninit" to find commits in the mainline that are patches for UBI bugs. Following the fixes tags label in the commit message [8], we locate the bug-introducing commit. We then select bugs that were introduced between v4.15-rc1 to v4.19 and involve stack variables for evaluating false negatives. Overall, we find 12 UBI bugs, among which our detector successfully found 9

TABLE VI: Bugs introduced in the new code, in the column of the **Patch**, **E** means that the patch is not easy to draft; here, we e-mail the bug to the maintainer. **A** means that the patch that we submitted was applied; **C** means that our bug was confirmed by the maintainers. **F** means that the bug has been fixed in the latest version of the kernel by others. **IL** : Information Leakage. **MC** :Memory Corruption. **B** :Benign. **HWCC** :Hardware configuration corruptions.

Sub-System	Module	Variable	Line No.	Intro.	Patch	Impact
Input/hideep	hideep.c	unmask_code	380	v4.15-rc1	A	IL
atomisp	atomisp-mt9m114.c	retvalue	1552	v4.15-rc1	A	MC
drm/nouveau	ioctl.c	type	269	v4.15-rc1	S	B
media/imx274	imx274.c	err	659	v4.15-rc1	F	B
net/mlx25	en_common.c	min_inline_mode	180	v4.15-rc1	F	B
net/mlx5e	en_dcbnl.c	params→ tx_min_inline_mode	989	v4.15-rc1	F	B
xfs	xfs_bmap.c	got.br_startoff	4868	v4.15-rc1	E	MC
xfs	xfs_bmap.c	s	1521	v4.15-rc1	E	MC
iio/adc	stm32-dfsdm-adc.c	status	866	v4.16-rc1	C	MC
iio/adc	stm32-dfsdm-adc.c	int_en	873	v4.16-rc1	C	MC
iio/adc	qcom-pm8xxx-xoadc.c	ch	599	v4.16-rc1	F	MC
net: mscc	ocelot.c	val	365	v4.18-rc1	F	MC
media: davinci_vpfe	dm365_isif.bc	format.pixelformat	234	v4.18-rc1	F	HWCC
display	dc_link.c	old_downspread.raw	1259	v4.18-rc1	A	HWCC
display	dc_link_dp.c	training_rd_interval	61	v4.18-rc1	F	HWCC
net:mscc	ocelot.c	val	34	v4.18-rc1	E	MC
scsi: sd	sd.c	sshdr.asc	2390	v4.19-rc1	E	B

TABLE VII: INCRELUX detected 2 bug fixes in our data set.

Sub-System	Module	Variable	Line No.	Fixed
media/imx274	imx274.c	err	659	v4.16-rc1
iommu/amd	amd_iommu.c	unmap_size	1524	v4.19-rc1

TABLE VIII: The false negatives for bug finding and bug fixes for mainline

	GroundTruth	TP	FN	FN-iCall	FN-Heap	FN-Padding
Bug Finding	12	9	3	2	1	0
Bug Fixes	2	2	0	0	0	0

TABLE IX: The false positives for bug finding and bug fixes for mainline

	Total	TP	FP	TN	FP-Guidance	FN-iCall	FN-Array
Bug Finding	44	22	22	N/A	14	7	1
Bug Fixes	2	2	0	72	N/A	N/A	N/A

bugs. 2 false negatives are caused by the imprecision of indirect call analysis, and 1 needs heap modeling.

Security Impact. We attempted to understand the security impacts for the 17 bugs that are not dismissed by Linux kernel maintainers. This turns out to be a non-trivial task as the danger of uninitialized uses heavily depends on the semantics of the variable. We consider a few conditions: (1) whether the uninitialized variable can cause control flow to diverge (e.g., used in an if condition); if so, does it cause additional memory operations such as free() to occur, which can likely lead to memory corruption. (2) whether the variable represents the size of objects; if so, it may cause out-of-bounds access. (3) whether the uninitialized variable will propagate to userspace, e.g., through `copy_to_user()` or logging to userspace-accessible places. Note that we did not confirm the impact through end-to-end verification (e.g., fuzzing), which we believe would be beyond the scope of the paper. Rather, we aim to obtain a rough estimate on how dangerous these bugs might be. Overall, we find 8 of these could potentially lead to memory corruption, 1 could cause the information leakage, 3 could cause the hardware configuration corruption, and 5 are benign.

F. Patch Identification Results

Reported Patches. INCRELUX can help developers reason about whether their patches are indeed working as intended. Specifically, in this evaluation, we choose to evaluate whether INCRELUX is capable of finding patches for the confirmed UBI bugs discovered by INCRELUX. This includes bugs discovered from the baseline analysis on v4.14, as well as the incremental analysis up to v4.19. This leaves us 74 confirmed UBI bugs. Note that their patches may or may not appear in the range of v4.14 to v4.19. It turns out only 2 are patched within this range, and INCRELUX correctly identified exactly the change that fixed the problem.

False Positives and False Negatives. Note that we do not claim the analysis for patch identification is sound or complete. Therefore, in principle, INCRELUX could report a commit earlier than the actual bug-fixing commit as the patch, due to false positives in the analysis. In the evaluation, we do not find any such case. Similarly, INCRELUX could miss real bug-fixing commits, due to false negatives. In this evaluation, we do not find this case either. We believe that INCRELUX will typically be quite accurate in identifying the correct patch for UBI bugs that it was originally able to detect, due to its use of precise symbolic execution to reason about path feasibility before and after the patch.

Bug Lifetime and Case Study. As mentioned, there are two bugs whose corresponding patches are correctly identified by INCRELUX. Out of the two, one bug was introduced before v4.14 (but captured by our baseline analysis of v4.14). For the other one, it was introduced in v4.15-rc1 and fixed in v4.16-rc1 with a lifetime of about ten weeks. The bug had unfortunately slipped through the stable release of v4.15. Below we use this bug as a case study to demonstrate how INCRELUX identifies the bug-introducing commit and the corresponding bug-fixing commit.

```
/* drivers/media/i2c/imx274.c
* uninteresting code lines are omitted */
static int imx274_regmap_util_write_table_8 ()
{
    int err;
    if (range_count == 1)
                err = regmap_write(regmap,
                        range_start, range_vals[0]);
    else if (range_count > 1)
                err = regmap_bulk_write(regmap, range_start,
                                &range_vals[0],
                                range_count);
+    else
+         err = 0;
    if (err) {
        return err;
    }
}
```

Fig. 6: The patch that fixed the previous bug; this bug was introduced in v4.15-rc1 and the patch was applied in v4.16-rc1. By continuously tracking the bug, INCRELUX could find both the bug upon introduction, and the time of the bug disapperance. If we use this patch as the input for the incremental analysis, the disapperance of the bug indicates that this commit was related to a bug fix.

This bug was introduced in v4.15-rc1 with the addition of the imx274 module, which is a `V4L2` driver for the Sony imx274 CMOS sensor. Function `imx274_regmap_util_write_table_8()` is supposed to write some values to some hardware resister; if the write fails, it should notify the caller by assigning the return value to an otherwise uninitialized local variable `err`. In Figure 6, we show that clearly without the patch, there is one branch where `err` will not be initialized and yet used in a conditional statement at line 15, which can potentially lead to logic errors in the kernel. This UBI bug is relatively easy to capture, as there exists one feasible path that triggers the uninitialized use. Furthermore, INCRELUX detects this bug easily through incremental analysis because the function `regmap_write()` and `regmap_bulk_write()` are already defined and analyzed before the v4.15-rc1. INCRELUX simply reuses their summaries. Similarly, when the patch is applied, we can also reuse the summaries of `regmap_write()` and `regmap_bulk_write()`, and simply re-analyze `iimx274_regmap_util_write_table_8()`. Clearly, the patch has caused `err` to become initialized in all branches before the use at line 15. We can therefore quickly confirm that the patch is indeed effective.

VI. Related Work

A. Bug Detection Tools for the Linux Kernel

Static Analysis Tools for the Linux Kernel. A variety of tools have been proposed to unearth bugs in the Linux kernel, some target specific types of bug, others are more general or extendable. KINT [64] is a static analysis tool designed for detecting integer overflow bugs in the Linux kernel. K-Miner [23] performs an inter-procedural analysis to detect memory-corruption vulnerabilities. UniSan [44] adopts byte-level data-flow analysis to detect information leakage caused by uninitialized variables. UBITect [75] uses type-qualifier inference to detect use-before-initialization (UBI) bugs on the stack. It mitigate the false positive problem by using under-constrained symbolic execution to find a feasible bug-triggering path for human inspection. Similar strategy (i.e., using symbolic execution to reduce false positives) has also been adopted by DEADLINE [70], which detects double-fetch bugs, a type of time-of-check-to-time-of-use (TOCTTOU) bug that is caused by fetching the same data from the user-space twice; and KUBO [41], which detects undefined behavior bugs. LRSan [63] aims to find a broader spectrum of TOCTTOU bugs that can bypass kernel security checks. CRIX [43] detects insufficient handling of erroneous states in the Linux kernel (e.g., forgetting to check if `kmalloc` returns `NULL`). K-MELD [21] detects kernel memory leak bugs through precise ownership analysis.

Because error handling paths are usually less tested, several tools have been developed to find bugs in error handling paths. Juxta [50] finds semantic bugs in Linux file systems by cross-checking paths that handle the same type of errors. RID [48] and [60] detect reference count bugs using consistency checks across error handling paths. EeCatch [54] aims to detect error handling code that causes the kernel to enter a state that is even worse than the error itself. HERO [67] finds bugs in error return paths that perform cleanup operations in an incorrect order, redundantly, or inadequately.

Coverity [19] is a commercial product (and thus incurs cost) that is able to perform incremental static analysis. Beyond its inner working being opaque, it seems to have the following limitations: a) It seems that it does not use underconstrained SE to automatically filter warnings and thus, is not able to precisely confirm whether a warning leads to a true positive in an automated way [19]. b) It also appears that it leaves some expectations on customers [16] to annotate the code to mark false positives – these warnings are later suppressed. The danger of this is that, these warnings may turn into true positives when other code is altered, and suppressing them could potentially hide the bug. More importantly, if developers do not annotate the code, the warnings related to false positives could reappear and may need to be re-analyzed. In addition to being fully transparent, IncreLux does not have these possible limitations.

Bug detection frameworks for the Linux Kernel. Dr.Checker [46] is a framework for detecting bugs in Linux drivers; it is extendable to detect different taint-style bugs such as integer overflow and out-of-bound memory access caused by untrusted user-space input. SUTURE [77] summarizes the taint results for each syscall interface (i.e., entry points) where it still employs a top-down analysis within each entry. The goal is to efficiently enumerate cross-entry taint flows, and identify bugs that only manifest across syscall invocations. CQUAL [22] is a type qualifier framework which is able to detect kernel bugs following user customized type system. Several papers also leverage type qualifier inference to find various bugs in the Linux kernel [75], [76]. Some frameworks [10], [12], [37], [52], [74] also use intra-procedural analysis to analyze Linux.

Dynamic Analysis Tools for the Linux Kernel. Dynamic analysis is widely applied to catch bugs in the Linux kernel at run time. Such techniques include hypervisor-based detection and fuzzing. Since the hypervisor can monitor a guest OS kernel, it can be used to dynamically catch kernel bugs. For example, bochspwn-reloaded [35] exposes memory disclosure bugs in the Linux kernel by tracking the flow of sensitive locations using a dynamic taint analysis and shadow memory.

Fuzzing finds bugs in the Linux kernel by repeatedly feeding system calls and other input dimensions like files and devices with mutated inputs. The state-of-the-art off-the-shelf kernel fuzzer is Syzkaller [61], which has been used by Google to perform continuous fuzzing for all versions of the Linux kernel [28]. Many research prototypes have also been developed to improve kernel fuzzing. IMF [31] tries to infer syscall dependencies from real-world applications. MoonShine [53] tries to improve the quality of initial seeds by "distilling" seeds from syscall traces of real-world applications. HFL [39] combines kernel fuzzing with symbolic execution. Difuze [18] uses static analysis to help construct well-formatted inputs to fuzz kernel drivers. Razzer [33] also uses static analysis to guide the discovery of kernel data-races. KRace [69] uses dynamic data-flow in addition to code coverage to fuzz data race bugs in the file systems. Janus [71] improve file system fuzzing by mutating both the syscalls and the on-disk file system. PeriScope [59] focuses on fuzzing the hardware interface of the kernel.

The fundamental limitation of all dynamic analysis tools is that they cannot find bugs in code that is not exercised during testing. Unfortunately, even with all recent advances in fuzzing, the code coverage of fuzzers is still very limited. As a static analysis tool, IncreLux can find bugs in all the compiled code.

B. Incremental Analysis and Regression Test

Relatively little attention has been paid to incremental analysis of the Linux kernel. Facebook Infer [32] is a static analysis framework that supports incremental analysis for various bug types. Interestingly, according to our testing of the UBI bugs at the time of writing, it appears the support is not very robust. One problem is that it is overly aggressive in reporting errors under the default mode: `PULSE_UNINITIALIZED_VALUE`, leading to potential false positives. For example, it disallows uninitialized arguments passed to a function call, as well as uninitialized return values. It is possible that an uninitialized argument becomes initialized in a callee (it is quite common in Linux kernel). Unfortunately, under the latent mode: `PULSE_UNINITIALIZED_VALUE_LATENT` where INFER tries to mitigate such false positives, we find that it may miss UBI bugs, leading to false negatives. We investigated the reason and it appears that INFER's requirement on function

parameters is incompletely. We document such examples in our project repository [11]. One other interesting feature of INFER is that its summary is built per path in a function, which is precise in nature but also challenging to scale up to complex programs. This is in contrast with IncreLux's design where the summary is built per function, which is much more scalable; and we achieve precision with a follow-up symbolic execution instead.

Furthermore, Infer does not use fixed point analysis for recursive functions, and it does the bottom-up analysis with a random starting point in the SCC, which can lead to non-determinism in the results [2]. Finally, applying it to the kernel may face other challenges like complex pointer arithmetic in the kernel (e.g., the `container_of` macro). And its incremental analysis inherits all those limitations. Conc-iSE [30] designs a symbolic execution algorithm to help generate new test cases for concurrent code affected by new changes. Due to the path exploration problems, inter-procedural symbolic execution cannot scale to the whole Linux kernel. Based on our experience, even with under-constrained symbolic execution, our tool has to frequently tradeoff precision (i.e., by making a variable under-constrained) for scalability. Regression verification [24] is another related concept that focuses on code changes. It aims is to check for software regression and make sure that an old function still works in the new version. To efficiently verify the absence of regressions, partition-based regression verification [14] divides the program input space into units of verification (differential partitions), allowing for gradual checking. None of these approaches perform incremental analysis to discover bugs in the Linux kernel, which poses new challenges because of scale, its rapid evolution, indirect calls, and complex function relationships (e.g., recursion).

C. Security patches

Maintainers of large software receive numerous bug reports and proposed patches everyday. They need to manually inspect these proposed commits and prioritize the security patches to be applied. This process is time-consuming and thus, there is a lot of work which targets differentiating security patches from normal bug fixes. The first approach towards this is based on leveraging commit messages; for example, supervised and unsupervised learning techniques [20], [29], [80] classify security vulnerabilities and general bugs based on commit message text. There exist other statistical approaches [65] that also heavily rely on bug messages. However, not all bug reports are properly written with necessary annotations relating to security information. We point out here that in fact, the kernel actually requires developers to add a fix tag to indicate if the patch is a bug fix. However, many independent developers are still not aware of this requirement. The second approach is leveraging static analysis and symbolic execution [66] to automatically differentiate security patches from the normal code commits. This approach compares the constraints from the unpatched version and the patched version; then, with some bug modeling, [66] can automatically identify security patches and even infer the type of the security vulnerability associated with a patch. As shown in section V, INCRELUX can very quickly verify if a patch adds or removes UBI bugs.

VII. DISCUSSION

INCRELUX uses a principled way to conduct an incremental analysis for the Linux kernel. To achieve scalability and reduce the turnaround time, it uses function summaries to avoid analyzing functions that are not affected by new code changes. Our evaluation over individual patches suggests that IncreLux might be useful in checking for individual commits before merging them to the mainline. Although our evaluations are on the allyes configuration, the approach can be used with other configurations as well.

While extremely effective in achieving its goals, INCRELUX does have some limitations on UBI bug detection. However, we note that all these limitation are inherited from the underlying analysis UBITect [75]. First, it only tracks uninitialized stack variables, as opposed to uninitialized global state variables (i.e., heap or global variables). This turns out to be a minor issue as the majority of UBI bugs we find are indeed due to uninitialized stack variables. We have verified this by sampling 51 UBI bugs through keyword search, i.e., "uninit" and "Uninit" from minor versions of v4.14.y and v4.15.y. Only 9 of them are not uninitialized stack variables. Second, we also do not track how an uninitialized stack variable may propagate to global states which then encounter uses. From analyzing the above 42 uninitialized stack variables, we find that only 5 did propagate to global states. Third, it only detects UBI bugs in a single thread and does not yet handle bugs that span multiple threads. In summary, we believe that INCRELUX is a significant step in enabling the timely analysis of bugs in the Linux kernel and leave these open problems to future research.

VIII. CONCLUSIONS

In this paper, we design and implement INCRELUX, a framework for principled incremental analysis of the Linux kernel. INCRELUX is effective across both mainline and stable versions, and provides an effective progressive way to detect bugs with dramatic speed ups compared to today's expensive whole-kernel analysis that needs to be performed each time a new Linux kernel version is released. This speed up aids developers in quickly identifying bugs before merges can happen, thereby enabling much safer Linux kernel version releases. By tracking bug lifetimes across Linux versions, INCRELUX is able to identify bugs that potentially are hard to exploit (since they have remained for long) and newer bugs that need more immediate attention. The same feature also allows INCRELUX to effectively disambiguate bug fixes from other normal commits. Our evaluations of INCRELUX over a fairly large set of Linux versions show that INCRELUX dramatically reduces the analysis time towards detecting bugs (a factor of nearly a $1000\times$ speed up at times). Furthermore, we show that it is able to achieve almost perfect accuracy in terms of conclusions that it draws via its incremental analysis of a new version, in comparison to a holistic clean slate analysis of the same version. We also point out some future directions that can further expand and improve the capabilities of INCRELUX.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality

of the paper. This research was partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. It was also partially supported by NSF award CNS-1718997, NSF project CNS-1801534, NSF grant # 1652954 and ONR under grant N00014-17-1-2893.

REFERENCES

[1] "How the development process works," https://www.kernel.org/doc/html/latest/process/2.Process.html. V-B

[2] "Infer is not deterministic," https://github.com/facebook/infer/issues/1110. VI-B

[3] "CVE-2015-3636," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3636, 2015. I

[4] "CVE-2019-2215 - Bad Binder: Android In-The-Wild Exploit," https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html, 2019. I

[5] "CVE-2021-22555: Turning
x00
x00 into 10000$," https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html, 2021. I

[6] "CWE-457: Use of Uninitialized Variable," https://cwe.mitre.org/data/definitions/457.html, 2021. II-C

[7] "Kernel.org git repositories," git.kernel.org/pub/scm/linux/kernel, 2021. I

[8] "Linux," https://github.com/torvalds/linux, 2021. II-A, V-E

[9] "Sequoia: A Local Privilege Escalation Vulnerability in Linux's Filesystem Layer (CVE-2021-33909)," https://blog.qualys.com/vulnerabilities-threat-research/2021/07/20/sequoia-a-local-privilege-escalation-vulnerability-in-linuxs-filesystem-layer-cve-2021-33909, 2021. I

[10] "Smatch," https://repo.or.cz/w/smatch.git, 2021. VI-A

[11] "IncreLux," https://github.com/seclab-ucr/IncreLux, 2022. VI-B

[12] I. Abal, C. Brabrand, and A. Wasowski, "Effective bug finding in c programs with shape and effect abstractions," in International Conference on Verification, Model Checking, and Abstract Interpretation. Springer, 2017, pp. 34–54. VI-A

[13] J.-J. Bai, T. Li, K. Lu, and S.-M. Hu, "Static detection of unsafe DMA accesses in device drivers," in USENIX Security Symposium, 2021. I

[14] M. Böhme, B. C. Oliveira, and A. Roychoudhury, "Partition-based regression verification," in 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 302–311. VI-B

[15] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao, "Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 388–400. II-B, II-B, II-B

[16] A. Chou. False positives over time: A problem in deploying static analysis tools. https://www.cs.umd.edu/~pugh/BugWorkshop05/papers/34-chou.pdf. VI-A

[17] K. Cook., "Kernel exploitation via uninitialized stack." https://www.defcon.org/images/defcon-19/dc-19-presentations/Cook/DEFCON-19-Cook-Kernel-Exploitation.pdf., 2011. II-C

[18] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017. I, VI-A

[19] Coverity, "COVERITY SCAN STATIC ANALYSIS," https://scan.coverity.com/, 2021. I, VI-A, VI-A

[20] D. C. Das and M. R. Rahman, "Security and performance bug reports identification with class-imbalance sampling and feature selection," in 2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR). IEEE, 2018, pp. 316–321. VI-C

[21] N. Emamdoost, Q. Wu, K. Lu, and S. McCamant, "Detecting kernel memory leaks in specialized modules with ownership reasoning," in Network and Distributed System Security Symposium (NDSS), 2021. I, VI-A

[22] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," in ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), 2002, pp. 1–12. VI-A

[23] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux." in Network and Distributed System Security Symposium (NDSS), 2018. I, VI-A

[24] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similar programs," Software Testing, Verification and Reliability, vol. 23, no. 3, pp. 241–258, 2013. VI-B

[25] Google. (2021) Kernel addresssanitizer. https://github.com/google/kasan. I

[26] ——. (2021) KMSAN. https://github.com/google/kmsan. I

[27] ——. (2021) KTSAN. https://github.com/google/ktsan. I

[28] Google, "syzbot," https://syzkaller.appspot.com/upstream/, 2021. I, VI-A

[29] K. Goseva-Popstojanova and J. Tyo, "Identification of security related bug reports via text mining using supervised and unsupervised classification," in 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2018, pp. 344–355. VI-C

[30] S. Guo, M. Kusano, and C. Wang, "Conc-ise: Incremental symbolic execution of concurrent software," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 531–542. VI-B

[31] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017. I, VI-A

[32] D. Harmim, V. Marcin, and O. Pavela, "Scalable static analysis using facebook infer." I, VI-B

[33] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in IEEE Symposium on Security and Privacy (SP). IEEE, 2019. I, VI-A

[34] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference." in USENIX Security Symposium, 2004. III-B

[35] M. Jurczyk and E. Ghuliev, "bochspwn-reloaded." https://github.com/googleprojectzero/bochspwn-reloaded, 2019. VI-A

[36] Kees Cook, "Status of the Kernel Self Protection Project," in Linux Security Summit, 2016. I

[37] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: Toward type checking# ifdef variability in c," in Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, 2010, pp. 25–32. VI-A

[38] T. kernel development community, "How the development process works," https://www.kernel.org/doc/html/latest/process/2.Process.html, 2021. V-A

[39] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in Network and Distributed System Security Symposium (NDSS), 2020. I, VI-A

[40] F. Li and V. Paxson, "A large-scale empirical study of security patches," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2201–2215. [Online]. Available: https://doi.org/10.1145/3133956.3134072 I

[41] C. Liu, Y. Chen, and L. Lu, "Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel," in Network and Distributed System Security Symposium (NDSS), 2021. I, VI-A

[42] K. Lu and H. Hu, "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, 2019, pp. 1867–1881. IV

[43] K. Lu, A. Pakki, and Q. Wu, "Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences," in USENIX Security Symposium, 2019. I, VI-A

[44] K. Lu, C. Song, T. Kim, and W. Lee, "Unisan: Proactive kernel memory initialization to eliminate data leakages," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 920–932. II-C, IV, VI-A

[45] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying." in Network and Distributed System Security Symposium (NDSS), 2017. II-C

[46] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "{DR}.{CHECKER}: A soundy analysis for linux kernel drivers," in 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 1007–1024. I, II-B, III-B, III-B, VI-A

[47] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, "Dns cache poisoning attack reloaded: Revolutions with side channels," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 1337–1350. I

[48] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "Rid: finding reference count bugs with inconsistent path pair checking," in International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016. VI-A

[49] A. Milburn, H. Bos, and C. Giuffrida, "Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities." in Network and Distributed System Security Symposium (NDSS), vol. 17, 2017, pp. 1–15. II-C

[50] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in ACM Symposium on Operating Systems Principles (SOSP), 2015. VI-A

[51] E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in Static Analysis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 165–180. II-B, II-B

[52] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," Acm sigops operating systems review, vol. 42, no. 4, pp. 247–260, 2008. VI-A

[53] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing os fuzzer seed selection with trace distillation," in USENIX Security Symposium, 2018. I, VI-A

[54] A. Pakki and K. Lu, "Exaggerated error handling hurts! an in-depth study and context-aware detection," in ACM SIGSAC Conference on Computer and Communications Security (CCS), 2020. VI-A

[55] J. Pan, G. Yan, and X. Fan, "Digtool: A virtualization-based framework for detecting kernel vulnerabilities," in USENIX Security Symposium, 2017. I

[56] L. Pollock and M. Soffa, "An incremental version of iterative data flow analysis," IEEE Transactions on Software Engineering, vol. 15, no. 12, pp. 1537–1549, 1989. I

[57] A. Rountev, M. Sharp, and G. Xu, "Ide dataflow analysis in the presence of large object-oriented libraries," in Compiler Construction, L. Hendren, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 53–68. II-B

[58] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for OS kernels," in USENIX Security Symposium, 2017. I

[59] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in Network and Distributed System Security Symposium (NDSS), 2019. I, VI-A

[60] X. Tan, Y. Zhang, X. Yang, K. Lu, and M. Yang, "Detecting kernel refcount bugs with two-dimensional consistency checking," in USENIX Security Symposium, 2021. I, VI-A

[61] D. Vyukov, "Syzkaller," https://github.com/google/syzkaller, 2021. I, VI-A

[62] T. A. Wagner and S. L. Graham, "Incremental analysis of real programming languages," in Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, ser. PLDI '97, 1997, p. 31–43. I

[63] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in ACM SIGSAC Conference on Computer and Communications Security (CCS. I, VI-A

[64] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with {KINT}," in 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12), 2012, pp. 163–177. I, IV, VI-A

[65] D. Wijayasekara, M. Manic, and M. McQueen, "Vulnerability identification and classification via text mining bug databases," in IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society. IEEE, 2014, pp. 3612–3618. VI-C

[66] Q. Wu, Y. He, S. McCamant, and K. Lu, "Precisely characterizing security impact in a flood of patches via symbolic rule comparison," in Network and Distributed System Security Symposium (NDSS), 2020. VI-C, VI-C

[67] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in USENIX Security Symposium, 2021. I, VI-A

[68] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," ACM Trans. Program. Lang. Syst., vol. 29, no. 3, p. 16–es, May 2007. [Online]. Available: https://doi.org/10.1145/1232420.1232423 II-B, II-B, II-B

[69] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in IEEE Symposium on Security and Privacy (SP). IEEE, 2020. I, VI-A

[70] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in IEEE Symposium on Security and Privacy (SP). IEEE, 2018. I, VI-A

[71] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in IEEE Symposium on Security and Privacy (Oakland), 2019. VI-A

[72] Xu, Wen and Moon, Hyungon and Kashyap, Sanidhya and Tseng, Po-Ning and Kim, Taesoo, "Fuzzing file systems via two-dimensional input space exploration," in 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019, pp. 818–834. I

[73] D. Yan, G. Xu, and A. Rountev, "Rethinking soot for summary-based whole-program analysis," in Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, 2012, pp. 9–14. II-B

[74] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing {API} usages through semantic cross-checking," in 25th {USENIX} Security Symposium ({USENIX} Security 16), 2016, pp. 363–378. VI-A

[75] Y. Zhai, Y. Hao, H. Zhang, D. Wang, C. Song, Z. Qian, M. Lesani, S. V. Krishnamurthy, and P. Yu, "Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 221–232. I, II-B, II-B, II-B, II-C, III-C, III-D, 22, IV, V-B, V-C, VI-A, VI-A, VII

[76] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in European Symposium on Research in Computer Security. Springer, 2010, pp. 71–86. VI-A

[77] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 811–824. VI-A

[78] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in 27th USENIX Security Symposium (USENIX Security 18). Baltimore, MD: USENIX Association, Aug. 2018, pp. 887–902. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-hang I

[79] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, "An investigation of the android kernel patch ecosystem," in 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng I

[80] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in Proceedings of the 2017

11th joint meeting on foundations of software engineering, 2017, pp. 914–919. VI-C

TABLE X: Incremental analysis results for mainline versions v4.14 to v5.4. **T(h)** : Total analysis time in hours. **CG Analysis(s)** : Call graph analysis time. **SU** : Speedup compared to exhaustive analysis of v4.14. **FM** : Number of functions modified compared to the immediate. predecessor. **FR** : Number of functions (re-)analyzed in this version. **Warn** : Number of Warnings reported in the current version. **Disappearing** : Number of warnings that disappear in the current version (compared with the last analyzed version). **Equal** : Number of warnings that remain in the current version compared to the immediate previous analyzed version. **New** : Number of warnings newly introduced in the current version compared with the last analyzed version. **SE-New** : New bugs confirmed by SE that are introduced in the new version.

versions	T(h)	SU	FM	FR	Warn	Disappearing	Remaining	New	SE-New
v4.14	106h45min	N/A	N/A	629862	103616	N/A	N/A	N/A	622
v4.15-rc1	28.26	3.78	23941	42548	21190	4325	99291	4979	69
v4.15-rc2	2.91	36.74	719	2617	2190	422	103848	211	0
v4.15-rc3	2.27	47.04	656	3084	1937	301	103758	238	0
v4.15-rc4	1.13	94.52	332	1268	718	116	103880	70	0
v4.15-rc5	1.28	83.31	329	1793	1339	207	103743	331	0
v4.15-rc6	1.15	92.6	273	1761	1282	96	103978	96	0
v4.15-rc7	0.43	248.74	101	403	263	21	104053	27	0
v4.15-rc8	1.33	80.15	243	1707	1305	114	103966	151	0
v4.15-rc9	0.63	169.82	217	1031	696	49	104068	48	0
v4.15	0.13	800.63	122	262	215	36	104080	48	2
v4.16-rc1	26.77	3.99	21251	52151	26922	4240	99888	4742	55
v4.16-rc2	0.24	453.72	220	521	342	10	104620	25	0
v4.16-rc3	0.7	151.6	434	1012	713	136	104509	77	1
v4.16-rc4	3.39	31.48	278	7398	3446	502	104084	509	4
v4.16-rc5	0.76	141.23	422	979	598	80	104513	76	0
v4.16-rc6	0.26	415.01	144	401	279	15	104574	27	0
v4.16-rc7	0.93	115.2	498	1543	819	107	104494	190	2
v4.16	0.33	318.92	183	535	278	18	104666	15	0
v4.17-rc1	35.1	3.04	23807	64758	33619	4493	100188	6620	28
v4.17-rc2	1.38	77.36	310	1290	1133	90	106718	89	0
v4.17-rc3	0.38	281.13	387	671	466	133	106674	72	1
v4.17-rc4	0.53	203.23	302	546	335	49	106697	30	4
v4.17-rc5	0.56	192.05	267	686	499	36	106691	31	0
v4.17-rc6	0.18	590.32	151	213	128	10	106712	15	0
v4.17-rc7	0.69	154.21	367	729	610	64	106663	56	0
v4.17	0.22	490.8	133	274	183	40	106679	17	0
v4.18-rc1	34.55	3.09	26607	64907	34669	3736	102960	7696	50
v4.18-rc2	3.25	32.85	597	7573	2060	159	110497	221	0
v4.18-rc3	0.32	336.81	251	557	281	14	110704	22	0
v4.18-rc4	1.1	97.41	587	1070	909	53	110673	183	1
v4.18-rc5	0.11	937.32	123	210	301	11	110845	143	0
v4.18-rc6	0.52	203.87	374	667	481	107	110881	136	0
v4.18-rc7	1.03	103.81	266	796	586	41	110976	53	0
v4.18-rc8	0.23	470.38	181	294	143	18	111011	2	0
v4.18	0.14	771.69	81	194	87	11	111002	13	0
v4.19-rc1	32.11	3.32	21896	55931	28780	4182	106833	6109	51
v4.19-rc2	0.55	195.37	271	1121	789	88	112854	126	0
v4.19-rc3	0.6	178.08	304	824	618	38	112942	39	0
v4.19-rc4	0.82	129.87	713	1717	1121	157	112824	218	1
v4.19-rc5	1.06	100.89	252	1015	867	157	112885	128	0
v4.19-rc6	0.53	201.73	231	693	686	45	112968	31	0
v4.19-rc7	2.56	41.72	384	3317	4634	70	112929	486	1
v4.19-rc8	0.39	271.21	129	491	411	6	113409	27	0
v4.19	0.43	247.3	116	579	484	62	113374	40	0
v5.4	97.9	1.09	99370	205327	158018	43762	69652	88366	N/A
v5.9	99.65	1.07	91741	197413	152746	40720	85425	67321	N/A