UNIVERSITY OF CALIFORNIA
RIVERSIDE

Preventing the Memory Errors in the Large-Scale C/C++ Software

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yizhuo Zhai

December 2023

Dissertation Committee:

Professor Zhiyun Qian, Co-Chairperson
Professor Srikanth V. Krishnamurthy, Co-Chairperson
Professor Chengyu Song
Professor Manu Sridharan

The Dissertation of Yizhuo Zhai is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

I am fortunate to have Dr. Zhiyun Qian and Dr. Srikanth V. Krishnamurthy as my co-advisors, who have provided invaluable support throughout my complex system security research journey. When faced with challenges in my research, Dr. Zhiyun Qian's diverse background knowledge and strong research ability enable him to approach problems from multiple angles, turning obstacles into new opportunities. His adeptness in research and problem-solving has been a profound source of enlightenment for me. In addition to the research mentorship, Dr. Srikanth V. Krishnamurthy has also embraced the role of a life mentor. He is able to provide me with insights from the perspective of someone who has gone through similar experiences, offering me information I wasn't aware of. This helps me gain a better understanding of the issues at hand and find solutions more effectively. He has been a constant source of encouragement during moments of research bottlenecks and has generously shared his experiences to guide my decision-making process. Through close interaction with them, I have found inspiration in our discussions and daily conversations. This engagement has gradually liberated me from the constraints of a learner's perspective, fostering the growth of a researcher's mindset. It has always been my great pleasure to work with them.

I would also like to express my sincere gratitude to the members of the committee: Dr. Chengyu Song and Dr. Manu Sridharan. Our collaborative efforts have been integral to the majority of projects undertaken during my PhD journey. In particular, I would like to extend my heartfelt appreciation to Dr. Chengyu Song. Our collaboration spanned all three projects, and his guidance has been invaluable to me. He was my initial introduction

to the realm of system security, guiding me through the gateway of knowledge during the inception of my PhD program. His unwavering support has saved me from reinventing the wheel and significantly expedited my progress. Upon joining the university, Dr. Manu Sridharan also became a vital collaborator in my research endeavors. He possesses strong project management experience and the ability to effectively integrate research with practical applications, significantly advancing the progress of the project. Both Dr. Chengyu Song and Dr. Manu Sridharan have been actively engaged in our weekly discussions and the advancement of our projects. Their contributions have played an indispensable role in propelling these projects forward. I hold in high regard the references they graciously provided, as these references have been pivotal in achieving critical research milestones. Their insights and expert guidance have been instrumental in cultivating and refining my research mindset. I also want to thank Prof. Mohsen Lesani for his support for my project, and Prof. Marko Princevac from ME department for being the external committee member on my qualification exam.

I extend my gratitude to all my exceptional and significant lab mates for their inspiring discussions and substantial contributions to my projects. Many of them have become collaborators on these projects, and our collaborative efforts have enriched the refinement of both projects and papers. It has been a rewarding experience working together.

I greatly thank my family for all the support. They are all living life earnestly and diligently working in their respective fields. Among them, some have experienced life's lows, but through silent efforts, they have reclaimed success. Whenever I feel down, they always offer me encouragement, serving as my strong support and the role models for how I live

and work. Additionally, I am especially grateful to my grandparents, who have given me irreplaceable love in this world.

My research received partial sponsorship from the U.S. Army Combat Capabilities Development Command Army Research Laboratory, conducted under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). Additionally, support came from NSF awards CNS-1718997 and CNS-1801534, NSF grant #1652954, ONR grant N00014-17-1-2893, and the UCR Dean's Distinguished Fellowship. My heartfelt gratitude to all these providers!

Publications.This thesis includes materials from several previously published papers:

1. Zhai, Yizhuo, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul 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, pp. 221-232. 2020.

2. Zhai, Yizhuo, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song et al. "Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel." In 2022 Network and Distributed System Security Symposium. 2022.

To all those who love me, treat me sincerely, and have helped me.

ABSTRACT OF THE DISSERTATION

Preventing the Memory Errors in the Large-Scale C/C++ Software

by

Yizhuo Zhai

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2023
Professor Zhiyun Qian, Co-Chairperson
Professor Srikanth V. Krishnamurthy, Co-Chairperson

The C and C++ programming languages are highly valued for their flexibility in low-level memory management and exceptional performance. They are widely used in various applications, including Linux Kernel, Google Chrome, Microsoft Windows, and Firefox. However, this emphasis on performance comes at the cost of memory safety, resulting in various memory errors, such as use-after-free, use-before-initialization (UBI) and type confusion, etc. These bugs not only impact system reliability but also pose significant security risks, potentially allowing attackers to gain control over the entire system. While the security impacts they bring are harmful, there lacks the efficient and effective approaches to detect them in the large scale software. In this dissertation, we would explains the challenges towards this topic and try to explore new approaches to tackle them. More specifically, we focus on two types of memory errors, use before initialization(UBI) and type confusion. In this dissertation, we (1) address that precise UBI analysis needs path-sensitive analysis and the current approaches either generate too many false positives or cannot scale to the large scale software. (2) studied the long presence of the bug once it merged into the Linux

kernel, and proposed the incremental approaches called IncreLux to detect UBI bugs for new kernel commits. IncreLux is able to analyzing individual commit within minutes and thus avoid the buggy code merging into the kernel upstream. (3) analyzed some type confusion vulnerabilities and patches for Chome, as well as the popular sanitizers' approach to mitigate type confusion bugs. One insight is that the developers encoded the type information to some fields of the data structure. Before type castings, developers already added some type checks by looking at those fields. Therefore, by collecting developers' check, we could reduce sanitizer's instrumentation if such type check is already performed. This aids in enhancing the efficiency of preventing type confusion bugs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software written in C/C++ is prone to memory errors, which in turn, pose security threats to the system and expose a significant attack surface to external threats. Importantly, even the trusted computing base, such as the OS kernel, may also contain memory errors that could allow attackers to gain control of the entire system and render all protection schemes futile.

To better mitigate the threats, the prevailing approach is to identify memory errors before the software release. Current state-of-art approaches include compiler time static analysis and run time dynamic analysis. Static analysis seeks to validate the correctness of memory usage with out executing the program. While dynamic approaches aim to place monitors within the program and ensure certain security property during execution. However, when these approaches are employed to large scale software, their limitations become apparent.

Static analysis validates every potential path from the program's entry point. Considering the millions of lines of code in a single software, path explosion renders the approach impractical during the development cycle. Dynamic analysis places the monitors into the software, and check the certain properties when program execute of the security critical operations. However, introducing these additional checks unavoidably burdens the software.

In summary, directly apply these state-of-art approaches to large software remains a challenge, and there is a pressing need to enhance their efficiency and effectiveness.

1.1 Problem Statement and Challenges

This thesis seeks to overcome the limitations of previous static and dynamic approaches in preventing memory errors in the large scale software by designing and implementing novel frameworks.

We first targeted at the use-before-initialization(UBI) bugs, a new emerged attack vector in software. Previously, the uninitialized use of a variable was considered undefined behaviors. However the exploit of CVE-2010-2963 demonstrated that these types of memory errors pose security threats to the system. Therefore, it becomes crucial to ensure that variables are appropriately initialized before each use. Simply zeroing out memory is not a viable solution, as the default value assigned to the variable may not be correct. Therefore, the most effective approach remains the detection of uninitialized use, reporting it to developers, and asking them to determine the appropriate value. Since UBI errors occur in specific pathways, the adoption of path-sensitive analysis becomes necessary. However,

applying path-sensitive analysis to every variable proves impractical for software with extensive code bases. In response, we developed and implemented UbiTect, a tool that employed a two-phase analysis. First, static analysis is applied to generate warnings for potential uninitialized variables. Subsequently, symbolic execution is employed to conduct a more precise analysis. To assess the efficacy of UbiTect, we utilized Linux kernel v4.14 as our evaluation benchmark. It takes one week to finish the whole kernel. Remarkably, we identified 118 previously undisclosed bugs through this evaluation process. Following email correspondence with the maintainers, we successfully verified 52 of these bugs.

During the evaluation of the UbiTect system, we recognized that one-week analysis time posed challenges in integrating the tool into the software development cycle of the target, i.e. linux kernel which releases new versions weekly. Moreover, when buggy code is introduced, it remains in the system for an extended period, creating a long vulnerability time window. It is advisable to subject new code to comprehensive checks before merging it into the code base. We observed that between different versions, only a small portion of the code is modified. Therefore, we could reuse the previous results, and conduct delta analysis to the new version. Based on this observation, we designed and implemented IncreLux, which is an incremental framework to detect the UBI bugs across versions. It conducts the one time whole program analysis and stores the intermediate information for each functions. Then it could analyze the new code and maintaining the same security guarantees as the whole kernel analysis. We evaluate IncreLux from v4.14 to v4.16, and performs per patch analysis, the results show the significant speedup and we have new bugs confirmed by kernel community.

The second type of bug we address is type confusion, which occurs when a variable is allocated as one type but subsequently used as a different, incompatible type, resulting in memory errors. As the compiler based static check is not sufficient, dynamic sanitizer approaches are more popular and widely used. These approaches involve instrumenting extra type-checking functions into the program, albeit at the expense of introducing considerable overhead. However, an insightful observation is that, developers already embed information into classes and validate these fields before casting, thereby ensuring the correctness of the casting process. This proactive measure by developers diminishes the necessity for certain security check instrumentation typically employed by sanitizers. Building upon this observation, we conceived and implemented T-Prunify. T-Prunify employs static analysis to gather the type information that developers have encoded into the code, subsequently pinpointing their checks. Once these checks have been identified, and if deemed sufficient, we can then judiciously remove redundant type checks that developers had inserted. To assess the efficacy of T-Prunify, we conducted evaluations on Chromium and achieved notable speedup improvements.

1.2 Thesis Contributions

In summary, this thesis contributes the following technical advancements to the field of security research:

1. **Necessity of Detecting UBI Bugs** We emphasize that for comprehensive mitigation of UBI bugs in the system, zeroing the variable is not the primary solution; rather, a more secure approach involves detecting and reporting these issues to developers.

2. **New Two-Phase Approach** We tackle the precision-versus-scalability dilemma in detecting UBI bugs, which often hinders the use of static analysis in large-scale software. To resolve this issue, we introduce a two-phase approach combining static analysis and symbolic execution.

3. **New Delta Analysis Framework** We design and implement an innovative summary-based incremental static bug detection framework for efficient delta analysis.

4. **New Overhead Reduction Approach** In our quest to mitigate type confusion, we explore various methods and conclude that developers' checks are the most efficient. Building on this insight, we develop an approach to extract this information, ultimately reducing redundant runtime monitors in dynamic analysis.

5. **Open Source** We will open source all analysis tool developed in this thesis for further adoption.

Chapter 2

UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel

2.1 Introduction

Linux kernels provide a secure foundation upon which services for user applications can be built. However, security vulnerabilities existing inside kernel code violate the security guarantees that it intends to provide. Among such vulnerabilities, use-before-initialization (UBI) is an emerging threat. A recent report from a Microsoft security team shows that the number of patched UBI bugs is similar to the number of patched use-after-free bugs

[117]. UBI bugs open up significant security threats against the operating system: they could enable attackers to take control over the entire system [7, 44, 108, 185], leak sensitive information [105, 115], and can be exploited using automated means [108].

Both static analysis and dynamic analysis have been applied to detect UBI bugs. Modern compilers provide the `-Wuninitialized` option to facilitate the detection of UBI bugs at compile time. Unfortunately, due to its limited analysis scope (i.e., intra-procedural), this cannot detect UBI bugs that involve multiple functions. In practice, many UBI bugs do occur inter-procedurally. For example, objects can be allocated in one function, initialized in another function, and used in a third function. Static symbolic execution like that in Clang static analyzer (CSA) [148], can perform more accurate analysis, but due to the path explosion, its ability to perform inter-module holistic program analysis is limited. Dynamic analysis used in MemorySanitizer [139] and kmemcheck [151] can also detect UBI bugs, but their limited code coverage means that they will miss many bugs.

Zeroing the allocated object is a popular mitigation strategy for UBI bugs. For example, PaX's `STACKLEAK` plugin [129] forces the initialization of kernel stacks during context switches between the kernel and user space. UniSan [105] forces the initialization of memory objects that may be uninitialized and may leave the kernel space (e.g., `copy-to-user`). SafeInit [115] does so for all stack and heap variables. However, we point out that **forced initialization can only be used to mitigate information leaks, but not other types of UBI bugs**. The reason is that, the value `0` used for initialization may violate a program's semantics and lead to undefined behaviors. For instance, initializing a pointer to `NULL` is sufficient towards preventing information leaks, but dereferencing a `NULL`-pointer results in a

different type of vulnerability viz., CWE-476 [11] (which is not desirable in OS kernels). For normal data, a few patches we submitted were also rejected due to incorrect initialization values. Based on this observation, we conclude that a better way to mitigate UBI bugs is to warn developers and let them decide upon the correct initialization values.

There are two particular challenges for reporting UBI bugs to developers. First, the Linux kernel has about 27.8 million lines of code and so, the analysis must be scalable. Second, most UBI bugs are **path-sensitive**, meaning that they can only be triggered if there is a feasible path between the allocation site and the use site, along which the involved variable will not be initialized. Because of these, UBI bugs are uniquely challenging to comprehensively discover and require inter-procedural path-sensitive analysis. We are not aware of any such analysis scaling to the whole kernel.

Flow-sensitive static analysis and symbolic execution are two state-of-art solutions that can help towards discovering UBI bugs. Our evaluations show that the former method scales well but generates too many warnings to inspect manually. Moreover, there are lots of false positives in those warnings. Symbolic execution reports fewer false positives but suffers from path explosion.

In this work, we seek to address the aforementioned two challenges, and design a tool suitable for reporting UBI bugs for manual inspection and fixing. To this end, we have developed UbiTect, a tool that combines flow-sensitive type qualifier inference and symbolic execution to find UBI bugs in the Linux kernel. In the first stage, UbiTect uses a soundy [109] flow-sensitive, field-sensitive and context-sensitive inter-procedural analysis to find potential UBI bugs. For each potential bug, this step also generates a guidance

for path exploration, so as to avoid paths that will never reach the use site or paths that will initialize the involved variable. In the second stage, UbiTect uses under-constrained symbolic execution [132] to find a feasible path according to the guidance. If a path is found, UbiTect will report the bug together with the corresponding path to make the manual inspection and fix easier.

We perform a thorough evaluation of UbiTect on Linux v4.14 under `allyesconfig`, which includes 16,163 files with 616,893 functions. UbiTect reported 190 bugs, among which 78 bugs were deemed by us as true positives, yielding a false positive rate of 59%. Among true positives, we found that the corresponding code of 9 bugs have been removed from the mainline kernel due to feature updates and 11 bugs were already fixed in the mainline. We submitted patches for the remaining 58 bugs and 37 were confirmed and applied by kernel maintainers. In addition, based on these bugs, we apply some intuitive heuristics and uncover 15 more bugs, thereby confirming 52 bugs in total. Details are provided in the evaluation of this chapter.

Contributions In this paper, our contributions are as follows:

- **Design.** We design UbiTect, which combines scalable type qualifier inference with symbolic execution to perform scalable and precise detection of Use-before-Initialization bugs in the Linux kernel.
- **Implementation.** We implement UbiTect on the LLVM 7.0.0 compiler toolchain and KLEE with 13,446 LoC. The tool is open sourced [13].
- **Results.** UbiTect found 78 bugs in the v4.14 Linux kernel, where 11 were already fixed.

We report the rest of the bugs to the Linux community and 37 were confirmed by Linux maintainers.

2.2 Use-before-Initialization Bugs

In this section, we highlight the severity of UBI bugs and the challenges in detection.

2.2.1 From UBI to Arbitrary Code Execution

The first example is a bug that was found in the `queue_manag` function (simplified in Figure 2.1) and patched in revision `1a92b2b`. The root cause for this bug is that the pointer `backlog` (line 14) is only initialized (line 16) when `(cpg->eng_st == ENGINE_IDLE)`.

Although this case is simple, it highlights the severity of the security impact of UBI bugs. The variable `backlog` belongs to the type structure `crypto_async_request`, which contains a function pointer `complete` (line 8). When `backlog` is left uninitialized, it could point to an arbitrary memory location depending on what value was stored at that address (`&backlog`) before, and `backlog->complete` could also point to arbitrary code. Since `backlog` is allocated on the kernel stack, by utilizing stack spray [108], an attacker can control `backlog` and thus, the function pointer (`backlog->complete`). Consequently, when this function is invoked at line 19, the attacker can achieve arbitrary code execution.

In addition to control-flow hijacking attacks, an attacker can also launch arbitrary reads and writes by overlapping attacker-controlled data with uninitialized pointers (e.g., CVE-2010-2963 [44]). Moreover, if a critical decision variable (e.g., `authenticated`) is uninitialized, an attacker can bypass security checks and induce other unexpected control

```c
/* file: drivers/crypto/mv_cesa.c
 * uninteresting code lines are omitted
 */
typedef void (*crypto_completion_t)(
  struct crypto_async_request *req, int err);

struct crypto_async_request {
  crypto_completion_t complete;
};

static int queue_manag(void *data)
{
  /* backlog is defined without initialization */
  struct crypto_async_request *backlog;
  if (cpg->eng_st == ENGINE_IDLE)
    backlog = crypto_get_backlog(&cpg->queue);
  if (backlog)
    /* uninitialized pointer dereferenced! */
    backlog->complete(backlog, -EINPROGRESS);
  return 0;
}
```

Figure 2.1: This code snippet shows a UBI bug in the Linux kernel. The variable `backlog` is not initialized if `(cpg->eng_st != ENGINE_IDLE)`. It allows arbitrary code execution once an attacker exploits the bug to control the value left on the kernel stack.

flows. A subsequent research effort has shown that such attacks are practical and can be constructed in an automated manner [108].

2.2.2 Challenges in Detecting UBI Bugs

The key challenge in detecting UBI bugs is the need for high-precision analysis (to reduce false positives), which can conflict with our goal of scaling up the analysis to the entire Linux kernel. Figure 2.2 depicts a good example: function `vmw_translate_mob_ptr` takes three input arguments and an output argument `*vmw_bo_p`, which is supposed to be initialized at line 16. Under normal circumstances (i.e., the lookup succeeds), `*vmw_bo_p` will be initialized. However, when the callee enters an error related return path (line 15), `*vmw_bo_p` is left unchanged.

Need for Inter-procedural Analysis A conservative intra-procedural analysis can require that all the variables must be initialized at all levels (e.g., both the pointer and the data the pointer points to), when passed to a callee. However, since the callee may not access all input arguments (e.g., when an error is returned at line 15), this requirement is too restrictive and will generate too many false positives. Therefore, an inter-procedural analysis is necessary. Moreover, since `*vmw_bo_p` is left unchanged upon an error return, whether the actual argument is uninitialized or initialized depends on the calling context (i.e., whether the caller has already initialized it). Hence, a context-sensitive inter-procedural analysis is preferable. Similarly, since the callee may not access all the fields of an argument (e.g., `sw_context`), performing a field-sensitive analysis is preferable.

```c
/* file: drivers/gpu/drm/vmwgfx/vmwgfx_execbuf.c
 * uninteresting code lines are omitted
 */
static int vmw_translate_mob_ptr(
    struct vmw_private *dev_priv,
    struct vmw_sw_context *sw_context,
    SVGAMobId *id,
    struct vmw_dma_buffer **vmw_bo_p)
{
  struct vmw_dma_buffer *vmw_bo;// = NULL;
  uint32_t handle = *id;
  int ret = vmw_user_dmabuf_lookup(
    sw_context->fp->tfile, handle, &vmw_bo);
  if (unlikely(ret != 0))
    return -EINVAL;
  *vmw_bo_p = vmw_bo;
  return 0;
}
```

Figure 2.2: An inter-procedural UBI bug in the Linux kernel. Argument `vmw_bo_p` may remain uninitialized during error return.

Needs for Path-Sensitive Analysis Another interesting part of this example is that the local variable (`vmw_bo`) is not initialized at first (line 10), and may not be initialized if the call to the function `vmw_user_dmabuf_lookup` fails (line 12). However, since `vmw_translate_mob_ptr()` checks the return value to detect the error (line 14-15), the uninitialized value will not reach a use (line 16). Thus, in essence, having a data-flow between where the variable is uninitialized and used, is a necessary condition for UBI bugs but is not sufficient (i.e., , the corresponding execution path must be feasible). Unfortunately, no path-sensitive analysis (e.g., dynamic analysis) can scale to cover all the paths in the kernel. As a practical compromise, UbiTect uses under-constrained symbolic execution to verify the feasibility of a potential buggy path.

2.3 Overview

In this section, we show how UbiTect combines type qualifier inference and symbolic execution to detect UBI bugs. Figure 2.3 illustrates the workflow of UbiTect and we will explain each component in the following content. The design of the type inference will be presented more formally in subsection 2.4.2.

2.3.1 Pre-processing

To make the analysis easier, UbiTect first compiles Linux source code to its LLVM Intermediate representation (IR). To improve the scalability of the type inference, UbiTect adopts the bottom-up style inter-procedural analysis. To support the bottom-up style analysis, the second step is to build the call graph of the whole code base so as to (1)

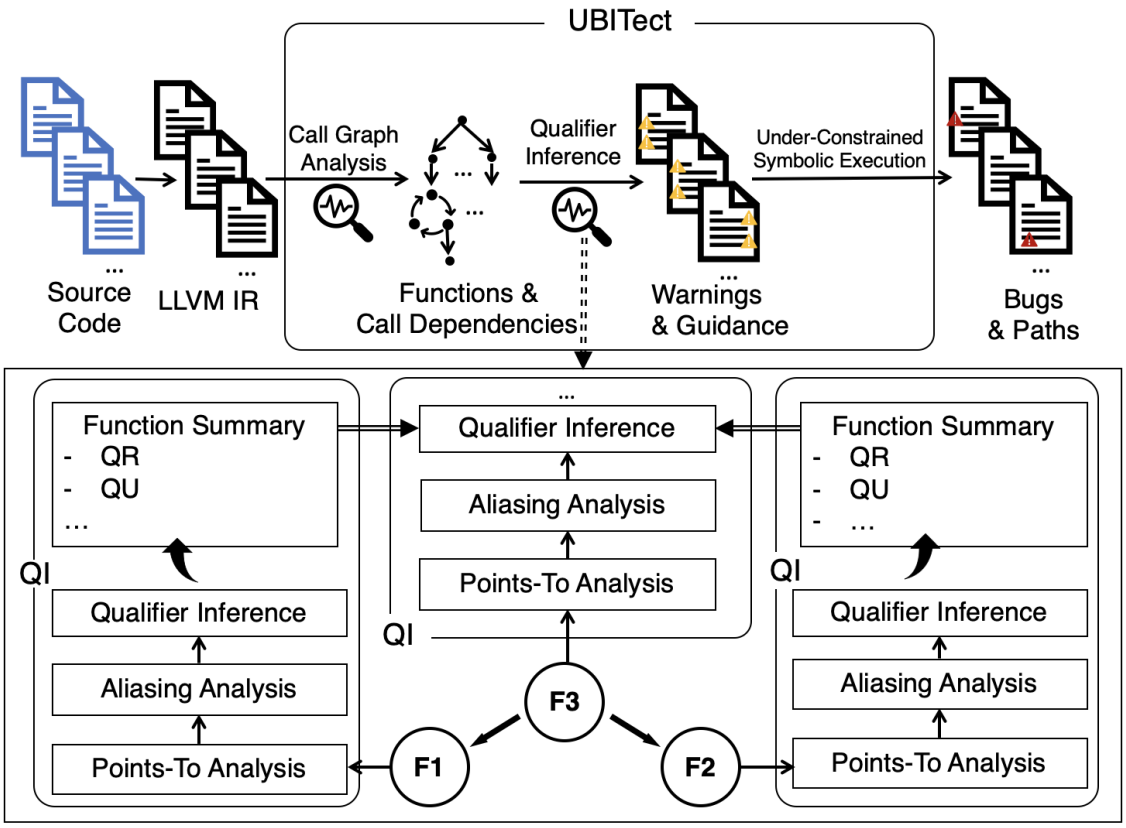

Figure 2.3: The workflow of UBITECT, "QI":Qualifier Inference, "QR":qualifier requirements, "QU": qualifier updates

resolve indirect call targets, (2) build the dependency tree between caller and callee(s), and (3) find potential recursive chains.

2.3.2 Type Qualifier Inference

Type qualifiers have been used in previous works to detect security bugs. For example, Johnson and Wagner [87] introduced two qualifiers `kernel` and `user` to track the provenance of pointers (i.e., whether their values are controlled by user space) and find unsafe dereferences of user-supplied pointers. In this work, we adopt the flow-sensitive type qualifier inference [65] to detect UBI bugs.

From a high level, we introduce two new qualifiers: *init* and *uninit*, where *init* $\preceq$ *uninit* (i.e., *init* is a subtype of *uninit*); and defines the subtype relations between qualified types (e.g., *init int* $\preceq$ *uninit int*). Besides the trivial check that an expression of `uninit` cannot be assigned to a location of `init`, UbiTect adds additional checks/assertions to detect use of initialized variables:

- Only expressions of `init` type can be dereferenced; and
- Only expressions of `init` type can be used in conditional branches.

UbiTect only considers those two assertions that capture UBI bugs with security implications here and ignore other types of uses of such variables. For example, adding two uninitialized variables reflects an uninitialized usage, but is not security-critical.

Since the IR generated by the compiler does not contain any qualifier, UbiTect performs automated **inference** to assign a qualifier for every variable at every program point within a function, including its argument(s) and return value(s). If UbiTect can successfully infer all the qualifiers, then the analyzed function is free of UBI bugs. Otherwise we find potential UBI bug(s) and the corresponding guidance will be generated and passed to UbiTect's symbolic execution engine. We will first explain how UbiTect infers qualifiers within a function and generates function summaries; then we will describe how inter-procedural qualifier inference works.

Intra-procedural Qualifier Inference. The intra-procedural qualifier inference is done as follows. (1) UbiTect assigns each expression (LLVM value) with a symbolic type κ. (2) Along different types of expressions, UbiTect generates subtyping constraints according to rules in subsection 2.4.2. (3) When encountering the security critical operations listed above,

UbiTect enforces that the corresponding expression has the concrete qualifier *init*. (4) UbiTect resolves the symbolic types into concrete qualified types by solving the constraints.

Take `aa_splitn_fqname` in Figure 2.4 as an example. At the entry of the function (line 6), `ns_name` and `ns_len` are assigned with two symbolic types κ_1 *const char* κ_2* κ_3* and κ_4 *size_t* κ_5*. Because `ns_name` (`%2`) and `ns_len` (`%3`) in basic block (BB) `%7` are dereferenced as pointers, the qualifier of the pointer should be *init*. UbiTect can then resolve their qualified types at least to be *uninit const char uninit* $*$ *init*$*$ (initialized pointer to uninitialized pointer to uninitialized constant char) and *uninit size_t init*$*$ (initialized pointer to uninitialized integer).

Function Summaries Generations. After intra-procedural qualifier inference, UbiTect generates function summaries (FS) for every function. Each function summary includes (1) qualifier requirements (QR) over the input arguments for the target function to be invoked without triggering UBI bugs, (2) qualifier updates (QU) for in and out parameters, and (3) qualifier of the return value.

Here, we continue using `aa_splitn_fqname` as an example and focus on how we generate qualifier requirement and qualifier updates for the input arguments `ns_name` and `ns_len`. Let us assume that the actual argument types are κ_1 *const char* $\kappa_2 *$ *init*$*$ and κ_4 *size_t init*$*$, where κ_i is symbolic (i.e., either *init* or *uninit*). By assigning the constant integer to `*ns_name` (line 10) and `*ns_len` (line 11), their qualified types will be updated to κ_1 *const char init* $*$ *init*$*$ and *init size_t init*$*$. However, when the control flow merges at basic block `%8` before returning, because these two variables are not written-to in the other branch (when `name == NULL`), the updates to the qualifier when `aa_splitn_fqname` returns

will be decided by the least-upper bound of κ_2 and *init* (i.e., $\kappa_2 \vee init$), as well as κ_4 and *init*.

To enable context-sensitive inter-procedural analysis, we keep κ_2 and κ_4 as symbolic as "updates to the parameters" in the function summary, and calculate the actual updates according to the calling context.

Inter-procedural Qualifier Inference. After we derive the summary of `aa_splitn_fqname`, we can proceed to analyze `aa_fqlookupn_profile`. The arguments `&ns_name` (`%4`) and `&ns_len` (`%5`) point to memory objects allocated on the stack and thus, the qualified types are *uninit char uninit* $*$ *init*$*$ and *uninit size_t init*$*$. Their qualified types are compatible with the QR generated above. After invocation, according to the QU, their types **remain the same** because when $\kappa_2 = uninit$, $uninit \vee init = uninit$.

When processing the `if` statement on line 22, UbiTect enforces that the expression used as the branch condition has a qualifier *init*. However, in `aa_fqlookupn_profile`, this subtyping constraint cannot be satisfied because the qualified type of `ns_name` (`%7`) is *uninit char uninit*$*$. Due to this conflict, the inference module outputs a potential UBI bug on line 22 (BB `%3`) of `aa_fqlookupn_profile`.

Guidance for Symbolic Execution. To mitigate the path explosion problem, UbiTect generates a guidance for the symbolic execution engine (SE). The guidance includes an avoidlist and a mustlist of basic blocks. A basic block is inserted into the avoidlist when (1) the involved variable is initialized or (2) the basic blocks can never lead to the use site. A basic block is inserted into the mustlist when (1) the involved variable becomes uninitialized

```
1  /* file: security/apparmor/policy.c
2   * uninteresting code lines are omitted
3   */
4  const char *aa_splitn_fqname( const char *fqname, size_t n,
5              const char **ns_name, size_t *ns_len) {
6      const char *name = skipn_spaces(fqname, n);
7      if (!name)
8          return NULL; //*ns_name is not initialized
9      *ns_name = NULL;
10     *ns_len = 0;
11     /* populate *ns_name */
12     return name;
13 }
14 int aa_fqlookupn_profile(struct aa_label *base,const char *fqname, size_t n) {
15   const char *name, *ns_name;
16   size_t ns_len;
17   name = aa_splitn_fqname(fqname, n,
18                 &ns_name, &ns_len);
19   if (ns_name) { // UBI!
20     //ns = aa_lookupn_ns(labels_ns(base),
21                 //ns_name, ns_len);
22   }
23   return 0;
24 }
```

Figure 2.4: An inter-procedural UBI bug in the apparmor module.

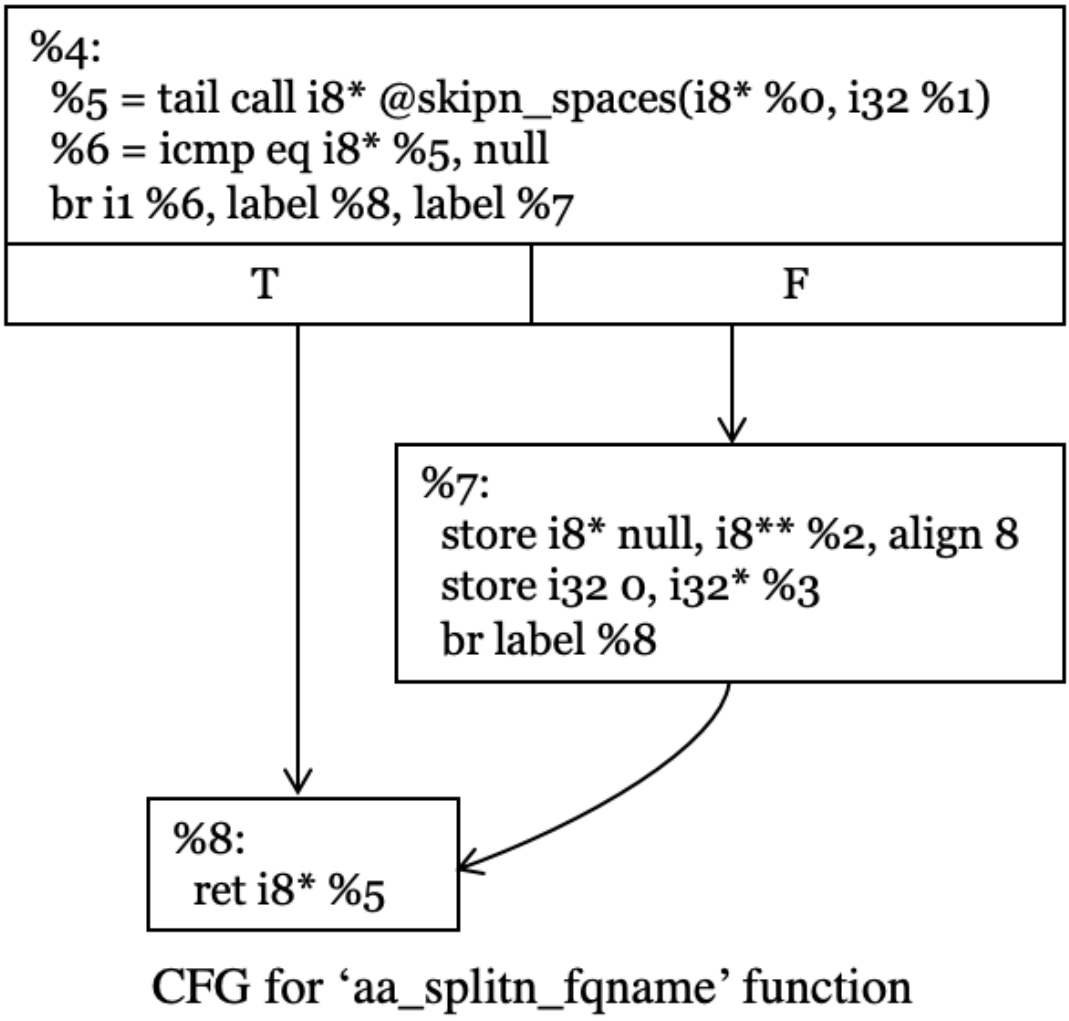

Figure 2.5: LLVM IR for function `aa_splitn_fqname` with control-flow graph for Figure 2.4.

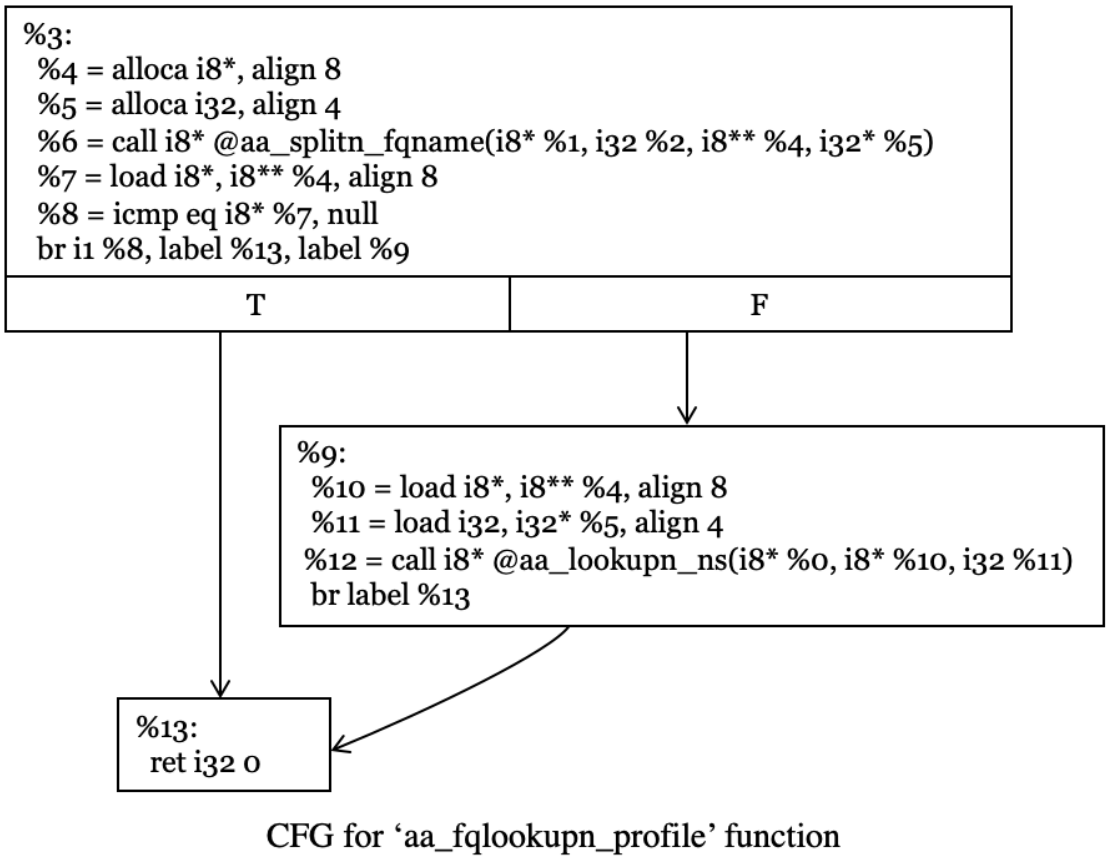

Figure 2.6: LLVM IR for function `aa_fqlookupn_profile` with control-flow graph for Figure 2.4.

or (2) the uninitialized variable is used. For the UBI bug detected above, UbiTect passes SE a avoidlist containing `%7` where the variable is initialized and a mustlist containing `%3` where UBI happens.

2.3.3 Symbolic Execution

After getting the guidance, UbiTect uses under constrained symbolic execution to search for a feasible path (i.e., whose symbolic path constraints can be satisfied) from the allocation site (i.e., the entry of `aa_fqlookupn_profile`) to the problematic use site `%3`, while avoiding `%7`. If a feasible path is found (e.g., BB `%3,%4,%8,%3`), UbiTect outputs a report for manual inspection, together with the path.

2.4 UbiTect Design

This section describes the design details of UbiTect, including points-to and aliasing analysis, the formalization of the type inference, and the symbolic execution engine.

2.4.1 Points-to and Aliasing Analysis

As a precursor to flow-sensitive qualifier inference [65], UbiTect performs a flow-sensitive and field-sensitive intra-procedural points-to analysis; specifically, towards this it applies standard data-flow analysis. For each statement, a points-to map is maintained and updated according to the control-flow. This allows UbiTect to have different points-to sets for the same pointer at different program points (i.e., flow-sensitive).

Because type casting is common in the Linux kernel, the points-to map tracks all variables and (field-extended) objects regardless of whether their types are pointers or not. This allows UbiTect to handle (i) casting between pointers and integers and (ii) integer-based pointer arithmetic. UbiTect also handles two types of castings that are especially troublesome for points-to analysis: `container_of` and casting from a `void` pointer. When handling such cases, UbiTect dynamically extends the allocated object size (i.e., number of fields in a `struct` type), if the destination type contains more fields than the original object. Since such castings usually happen on function arguments, this procedure enables more precise function summaries which will be explained in subsection 2.4.3.

2.4.2 Qualifier Inference

Our qualifier inference component is an extension of the flow-sensitive analysis by Foster et al. [65], and the inference rules for basic expressions are the same. In addition, we consider **pair types** which model the fields inside a `C struct` type and present their corresponding type inference rules. Providing separate qualifiers for elements of pairs (i.e., `struct` fields) is important as `struct` is used extensively in the Linux kernel. More importantly, pointers to `struct` are often passed between kernel functions, and whether a field of a `struct` is or is not initialized is independent of the states of the other fields in the `struct`.

Given a program in LLVM IR, we present a type qualifier inference system to infer a qualifier (either *init* or *uninit*) for each register variable (i.e., LLVM expression) and each field that belongs to an allocated memory object. We perform the inference function-by-function in a bottom-up fashion. If we can successfully infer all the qualifiers

along with the data flow, then the analyzed function is correct; otherwise we find potential UBI bug(s).

While we neither elaborate nor contribute to the sophisticated theory behind type qualifiers here, we try to keep the narrative self-contained by describing the notations and concepts applied in the reference rules. Interested readers can refer to [65] for further details. We retain the standard qualifier notation from Foster et al. [65], and only present the type inference rules for pair expressions; the full set of inference rules is available to the interested reader in the appendix and supplementary material [12].

The subtyping relation between the two qualifiers is straightforward: $init \preceq uninit$ (i.e., *init* is a subtype of *uninit*), meaning that a variable of $init\ t$ could be valid wherever $uninit\ t$ is expected, but not vice versa. Defining the subtyping relations for qualified types, and in particular qualified reference types, is subtle. Considering the primitive type *int*, its subtyping relation of qualified *int* is:

$$\frac{Q \preceq Q'}{Q\ int \preceq Q'\ int}$$

This means that if qualifier $Q \preceq Q'$, then $Q\ int$ is a subtype of $Q'\ int$, For instance, $init\ int$ is a subtype of $uninit\ int$. When it comes to references, the rule is more complicated. The following rule defines the subtyping relation between qualified references.

$$\frac{Q \preceq Q'}{Q\ ref(\tau) \preceq Q'\ ref(\tau)}$$

Specifically, it requires that the type of the (τ) to which the references point, be **the same**.

Syntax

Our qualifier inference is performed on LLVM IR after the alias analysis. For simplicity of the discussion, we use the following abstract syntax following the one used in Foster et al. [65], instead of the full LLVM IR syntax.

$$
\begin{aligned}
e &:= x \mid n \mid \lambda^L x\colon t.\ e \mid e_1\ e_2 \mid \\
&\mid \mathsf{ref}^\rho\, e \mid\ !e \mid e_1 := e_2 \\
&\mid \langle e_1, e_2\rangle \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
&\mid \mathsf{fst}(e_1) := e_2 \mid \mathsf{snd}(e_1) := e_2 \mid \\
&\mid \mathsf{assert}(e, Q) \mid \mathsf{check}(e, Q) \\
t &:= \alpha \mid \mathit{int} \mid \mathit{ref}(\rho) \mid t \to^L t' \mid \langle t_1, t_2\rangle \\
L &:= \{\rho, .., \rho\}
\end{aligned}
$$

In the grammar defined above, an expression e can take a form of a variable x, a constant integer n, a function $\lambda^L x\colon t.\ e$ with argument x of type t, effect set L and body e. The effect set, L, stands for the set of abstract locations ρ that the function accesses, which is calculated as part of our alias analysis. A type t is either a type variable α, an integer type int, a reference $\mathit{ref}(\rho)$ (to the abstract location ρ), a function type $t \to^L t'$ (that is decorated with its effects L) or a pair type $\langle t_1, t_2\rangle$. The expression $e_1\ e_2$ is the application of function e_1 to argument e_2. The reference creation expression $\mathsf{ref}^\rho\, e$ (decorated with the abstract location ρ) allocates memory to store the value e. The expression $!e$ dereferences the reference e. The expression $e_1 := e_2$ assigns the value of e_2 to the location e_1 points to. The expression $\langle e_1, e_2\rangle$ is the pair of e_1 and e_2. The expressions $\mathsf{fst}(e)$ and $\mathsf{snd}(e)$ are the first and second elements of the pair e, respectively. The expressions $\mathsf{fst}(e_1) := e_2$ and

$\mathsf{snd}(e_1) := e_2$ assign the value of e_2 to the first and the second elements of the location that e_1 points to, respectively.

Note that, following the style of Foster et al. [65], we use **explicit** qualifiers to both annotate and check the initialization status of expressions. The expression $\mathsf{assert}(e, Q)$ annotates the expression e with the qualifier Q, which is used to **manually** annotate types (e.g., the `from` argument of `copy_to_user`). The expression $\mathsf{check}(e, Q)$ requires the top-level qualifier of e to be at most Q. We automatically insert the $\mathsf{check}(e, init)$ expressions by a simple program transformation before every security critical use to enforce the safety of the operations. Specifically, we consider a pointer dereference ($!e$) to be security critical; a similar connotation applies when e is used as the predicate of a conditional branch.

Qualified Types and Type Stores

Given the subtyping relations, we now define the qualified types.

$$\tau := Q\ \sigma$$

$$Q := \kappa \mid init \mid uninit$$

$$\sigma := int \mid ref(\rho) \mid (C, \tau) \to (C', \tau') \mid \langle \tau_1, \tau_2 \rangle$$

$$C := \epsilon \mid Assign(C, \rho \colon \tau) \mid ...$$

$$\eta := 0 \mid 1 \mid \omega$$

The qualified types τ can have qualifiers at different levels. Q can be a qualifier variable κ or a constant qualifier *init* or *uninit*. The flow-sensitive analysis associates a ground store C to each program point that is a vector that associates abstract locations to qualified types. Thus, function types are now extended to $(C, \tau) \to (C', \tau')$ where C is the store that the function is invoked in and C' is the store when the function returns.

To track when strong/weak updates should be performed, each location in a store C also has an associated linearity η that can take three values: 0 for unallocated locations, 1 for linear locations (i.e., only point-to a single abstract location and thus, admits strong updates), and ω for non-linear locations (i.e., can point-to multiple different abstract locations and thus, only admits weak updates). An abstract location is **linear** if the type system finds that it corresponds to a single concrete location in every execution. An update that changes the qualifier of a location is called a strong update; otherwise, it is called a weak update. Strong updates can be applied to only linear locations. The three linearities form a lattice $0 < 1 < \omega$. Addition on linearities is as follows: $0 + x = x$, $1 + 1 = \omega$, and $\omega + x = \omega$. The type inference system tracks the linearity of locations to allow strong updates for only the linear locations.

Since a store C maps from each abstract location ρ_i to a type τ_i and a linearity η_i, we write $C(\rho)$ as the type of ρ in C and $C_{lin}(\rho)$ as the linearity of ρ in C. Store variables are denoted as ϵ. We use the following store constructor to represent the store after an assignment expression as a function of the store before it.

$$Assign(C, \rho' \colon \tau)(\rho) \;=\; \begin{cases} \tau' \text{ where } \tau \preceq \tau' & \text{if } \rho = \rho' \wedge C_{lin}(\rho) \neq \omega \\ \tau \sqcup C(\rho) & \text{if } \rho = \rho' \wedge C_{lin}(\rho) = \omega \\ C(\rho) & \text{otherwise} \end{cases}$$

$$Assign(C, \rho' : \tau)_{lin}(\rho) \;=\; C_{lin}(\rho)$$

$Assign(C, \rho \colon \tau)$ overrides C by mapping ρ to a type τ' such that $\tau \preceq \tau'$. (in our approach, τ' can be any super-type of τ.) The condition $\tau \preceq \tau'$ permits the assignment of a subtype τ of resulting type τ' to ρ. If ρ is linear, then its type in $Assign(C, \rho : \tau)$ results in τ'; otherwise

$$\textsc{Int}_{\preceq}\quad \frac{Q \preceq Q'}{Q\ int \preceq Q'\ int} \qquad \textsc{Ref}_{\preceq}\quad \frac{Q \preceq Q'}{Q\ ref(\rho) \preceq Q'\ ref(\rho)}$$

$$\textsc{Fun}_{\preceq}\quad \frac{Q \preceq Q' \qquad \tau_2 \preceq \tau_1 \qquad \tau_1' \preceq \tau_2' \qquad C_2 \preceq C_1 \qquad C_1' \preceq C_2'}{Q\ (C_1, \tau_1) \rightarrow^L (C_1', \tau_1') \preceq Q'\ (C_2, \tau_2) \rightarrow^L (C_2', \tau_2')}$$

$$\textsc{Store}_{\preceq}\quad \frac{\tau_i \preceq \tau_i' \qquad \eta_i \preceq \eta_i' \qquad i = 1..n}{\{\rho_1^{\eta_1} : \tau_1, ..., \rho_n^{\eta_n} : \tau_1\} \preceq \{\rho_1^{\eta_1'} : \tau_1', ..., \rho_1^{\eta_n'} : \tau_n'\}}$$

$$\textsc{Pair}_{\preceq}\quad \frac{Q \preceq Q' \qquad \tau_1 \preceq \tau_1' \qquad \tau_1 \preceq \tau_2'}{Q\ \langle \tau_1, \tau_2 \rangle \preceq Q'\ \langle \tau_1', \tau_2' \rangle}$$

Figure 2.7: Store subtyping.

its type is conservatively determined as the least-upper bound of τ and its previous type $C(\rho)$.

The type inference system generates subtyping constraints between stores. We define store subtyping in Figure 2.7 and would explain them one by one. These constraints among stores subsequently give rise to constraints between linearities and types, which in turn lead to constraints between qualifiers and linearities. The rule $\textsc{Int}_{\preceq}$ requires a corresponding subtyping relation for the qualifiers associated with the type *int*. Similarly, the rule $\textsc{Ref}_{\preceq}$ requires the same subtyping relation between qualifiers and additionally requires the equality of the two locations. For the rule $\textsc{Fun}_{\preceq}$, we necessitate the subtyping relation between top-level qualifiers and implement contra-variance for the argument and input store, as well as co-variance for the return value and output store. Finally, the rule $\textsc{Store}_{\preceq}$ requires both subtyping and stronger linearity for corresponding locations. The rule $\textsc{Pair}_{\preceq}$

$$\textsc{Pair}\quad \frac{\Gamma, C \vdash e_1 \colon \tau_1, C' \qquad \Gamma, C' \vdash e_2 \colon \tau_2, C'' \qquad \kappa \text{ fresh}}{\Gamma, C \vdash \langle e_1, e_2 \rangle \colon \mathit{init}\ \langle \tau_1, \tau_2 \rangle, C''}$$

$$\textsc{Fst}\quad \frac{\Gamma, C \vdash e \colon Q\ \langle \tau_1, \tau_2 \rangle, C'}{\Gamma, C \vdash \mathsf{fst}(e) \colon \tau_1, C'}$$

$$\textsc{FstAssign}\quad \frac{\begin{array}{c}\Gamma, C \vdash e_1 \colon Q\ \mathit{ref}(\rho), C' \\ \Gamma, C' \vdash e_2 \colon \tau_1, C'' \qquad \kappa\ \langle \alpha_1, \alpha_2 \rangle \preceq C''(\rho) \qquad \tau_1 \preceq \alpha_1 \qquad \kappa, \alpha_1, \alpha_2 \text{ fresh}\end{array}}{\Gamma, C \vdash \mathsf{fst}(e_1) := e_2 \colon \tau_1,\ \mathit{Assign}(C'', \rho \colon \langle \tau_1, \mathsf{snd}(C''(\rho)) \rangle)}$$

Figure 2.8: Type inference rules for the pair expressions (`C struct` fields).

requires subtyping between the top-level qualifiers, and also subtyping for corresponding elements of the two pair type.

Type Inference System

A type inference system consists a set of rules which define the preconditions for each expression (with the analyzed function) to be executed safely without UBI. Such preconditions will impose subtyping constraints between each expression. Anchored by the (automatically inserted) $\mathsf{check}(e, \mathit{init})$ and (manually inserted) $\mathsf{assert}(e, \mathit{init})$ expressions, we can infer the qualifiers of the remaining expressions. Again, if the constraints are satisfiable, the analyzed function is free from UBI bugs and the inference can succeed; otherwise there may exist UBI bug(s) and the conflicting constraint(s) will reveal the reason.

Because the main difference between our system and the one described in Foster et al. [65], is the field-sensitivity, we only present the rules for the pair expressions in this Section (as shown in Figure 2.8). The complete set of rules can be found in the appendix and supplementary material [12]. The judgments are of the form $\Gamma, C \vdash e \colon \tau, C'$ that is read

as: in the type environment Γ and store C, evaluating e yields a result of type τ and a new store C'.

The rule PAIR type-checks the expressions e_1 and e_2 in order and results in an initialized pair type. The rule FST checks that the expression e is of a pair type and types $\mathsf{fst}(e)$ as the first element of the pair type. The qualifier Q of the pair type is unconstrained; qualifiers are only checked by the check expressions discussed above. The rule FSTASSIGN checks that the expression e_1 is of a reference type $ref(\rho)$, the post-store C'' (after checking e_1 and e_2) maps the reference ρ to a supertype of a pair type $\kappa\ \langle\alpha_1, \alpha_2\rangle$, and the type τ_1 of e_2 is a subtype of α_1. The resulting store remaps ρ to a new pair type where the first element is the type of τ_1 and the second element is unchanged. We elide the rules for snd that are similar to the rules for fst. The constraints generated by the new rules PAIR, FST and FSTASSIGN are type and store subtyping constraints that were also generated by the basic rules. Further, by the rule $\text{PAIR}_{\preceq}$, subtyping constraints between pair types are decomposed into subtyping constraints between qualifier and simpler types that are inductively decomposed into constraints between qualifiers and linearities. Thus, the added inference rules do not increase the complexity of the generated constraints.

2.4.3 Inter-Procedural Analysis

Given a function F in the call graph, after applying the type inference to each callee function separately, the summaries generated for all of these are used in the analysis of the caller function F. The function summary is represented as (1) the qualifier requirements for the input arguments (of the function), (2) the qualifier of returned value, and (3) the updates to in and out arguments. The requirements specify the weakest qualifiers for the

formal arguments that are necessary for the function to be invoked safely without triggering any UBI bug. This means that if the actual arguments have weaker qualifiers, UBI bug(s) may occur. The updates record the qualifiers of outputs, which in the C language, are output pointer arguments. To support context-sensitive inter-procedural analysis, the updates and return value are polymorphic, i.e., based on the qualifiers of the actual arguments from the callers, the qualifiers of the outputs may change.

As shown in subsection 2.4.2, a qualified function could be represented in the format of $Q\ (C, \tau) \rightarrow^{L} (C', \tau')$ where Q is the qualifier of the function object itself, C maps locations ρ to their types τ before the function is called, τ is the parameter type, C' maps locations ρ to their (possibly) updated types τ after the function is called, τ' is the return type, and L is the set of locations accessed by the function. The concept is further exemplified by the following example:

$$\begin{aligned} &init\ ([\rho \mapsto uninit\ int, \rho' \mapsto init\ int], ref(\rho)) \\ &\quad \rightarrow^{\{\rho,\rho'\}} \\ &\quad ([\rho \mapsto init\ int, \rho' \mapsto init\ int], init\ int) \end{aligned}$$

It represents an (initialized) function that starts with a pre-store where ρ is uninitialized and ρ' is initialized. The input is the reference for ρ, and the function accesses both ρ and ρ'. The function initializes ρ and leaves ρ' initialized. This function is summarized as follows – no initialization **requirements** for its parameter and one update: **update** parameter ρ to initialized.

Calculating and Using Summaries

Requirements over input arguments can be directly fetched from the inference result. While updates are a little complicated, they are calculated as follows. For any pointer argument, UbiTect maintains a copy of the alias set of its abstract location at both the entry and exit of the function. If the alias set changes, then the corresponding argument is updated during the execution, and the output qualifier is the least-upper bound of the qualifiers of all variables from the alias set at the exit of the function. If the points-to set still contains the initial value from the alias set at the entry of the function, then its qualifier is kept as symbolic, so as to support polymorphism. For a concrete example, please refer to section 2.3.

The qualifier of the return value is handled similarly: if it depends on the qualifier of the input value(s), UbiTect keeps them as symbolic so that the return value can have the appropriate qualifier based on the calling context.

Using function summaries, the implementation of context-sensitive inter-procedural analysis is straightforward.

- Inference constraints: Each actual argument must be a subtype of the corresponding formal argument (i.e., requirements). Adding this constraint allows us to (1) check if the callee can be safely invoked (if not, type inference over the current function will fail). and (2) automatically propagate the requirements from the callee to the caller, in case the caller passes its argument(s) to the callee.
- Apply updates: After the invocation of a function, the qualifiers of values inside the points-to set of pointer type argument(s) are updated according to the updates. Further,

the qualifier of the value used to receive the return value is the same as the qualifier of the return value.

- Indirect calls: For indirect calls, the actual arguments have to satisfy the requirements of all possible call targets, and the updates are conservatively calculated as the least-upper bound of all updates.

Special Cases

There are some nuances that are associated with summary-based inter-procedural analysis; here, we describe two that we belive are important.

Heap Objects Because our points-to analysis is intra-procedural, it cannot track aliases created or removed outside the current function. More importantly, the concurrent nature of the kernel also makes it hard to precisely reason about the qualifier for heap data. For example, thread A stores an initialized data to heap address $addr_h$; however, when A tries to load from the same address, the data may no longer be initialized because a concurrent thread B could have written an uninitialized data to the same address. To handle this, we (1) track the provenance of memory objects; any object that is not allocated in the current scope is conservatively considered to be a heap object (i.e., globally visible); and (2) enforce a conservative rule for writing to heap objects: the variable has to be fully initialized (i.e., with qualifier `init`); if the variable is of pointer type, we also require that the data it points to are initialized. By doing so, we can safely assume all data loaded from heap are also initialized but false positives are introduced because of this strategy.

Recursion After building the call graph, we observed recursions among functions calls. Fixed point analysis is adopted to handle such recursions. Specifically, a function in the circular dependency graph is randomly picked to start the qualifier analysis. For callees whose summaries are not available, the subtyping constraints are temporarily ignored. As a result, an imprecise summary of the associated function is constructed by the first-time analysis. Then UbiTect moves on to analyze its callers using this imprecise summary. Following the dependency circle, the function is analyzed again. Because this time the summaries of its callees will be available, despite being imprecise, a new summary would be generated. This process is repeated until there are no changes to the summaries.

2.4.4 Symbolic Execution

Up to this point, the type qualifier inference reported all the suspicious UBI locations. Next, UbiTect uses under-constrained symbolic execution to find true positives.

For each potential bug output by the static analysis module, the symbolic execution (SE) module first links all the bitcode files related to the bug. It then starts searching for a feasible path from the beginning of the function where the involved variable is allocated. During the exploration, the SE module will prune paths that include any basic block in the avoidlist or paths that do not include all basic blocks in the mustlist. In this way, type qualifier inference reduces the searching space for SE and makes it more scalable.

Because we explore a partial path, covering only the portion between initialization sits and use sites, rather than the entire execution path from entry of the kernel (e.g., system call) to the point of use, some false positives may still pass through the filter. Similarly, false positives caused by an imprecise call graph (i.e., indirect call targets) will not be filtered.

However, in our desgin, we take precautions to prevent the incorrect exclusion of any true positives.

Finally, despite the use of under-constrained symbolic execution and guided path exploration, due to path explosion and complex path constraints, the tool may still take a long time and/or a large amount of memory to verify a warning. To handle the large volume of warnings from the static analysis, we rank the remaining warnings by "the time taken to find a feasible path between the uninitialization site and the use site". Our observations are (1) bug reports with a feasible path are much easier for developers to verify and (2) the less complex the path is, the sooner symbolic execution will find it.

2.5 Implementation

UbiTect is built upon the LLVM compiler infrastructure. We adopt the whole kernel analysis infrastructure from KINT [158] and modify it to match the bottom-up analysis. Points-to analysis is based on the structure analysis code from [5] while under-constrained symbolic execution stands on KLEE [39]. Overall, 13,446 LoC are added, the distributions of which are shown in Table 2.1.

We manually summarize 26 functions from three major categories. (the reasons for doing so are provided within the discussion pertaining to each category):

- LLVM intrinsics and assembly functions: We do not have access to intrinsic functions such as `memset` and `memcpy`, as well as functions implemented entirely in assembly. Consequently, in these instances, we are not able to construct summaries through automatic inference and rely on manual effort.

Table 2.1: LoC for different analysis of UbiTect.

Analysis	**Line of Code**
Call Graph	708
Points-To	1,652
Alias	375
Qualifier Inference	4,460
Utility Functions	3,412
Symbolic Execution	2,839
Total	13, 446

- Heap allocation functions: For reasons discussed earlier, we manually summarize typical kernel heap allocation functions, including the `kmalloc` series and the `kmem_cache_alloc` series. Since these functions accept flag `GFP_ZERO`, which will initialize the allocated memory, we set the initial qualifier for the allocated object according to this flag. Because our points-to analysis is field-sensitive (instead of byte-sensitive) and the allocation size to these functions are in bytes, to determine the type of allocated object, we will follow the def-use chain of the returned address and check for a `bitcast` operation. If we cannot find one, we treat the object as having a single field (i.e., void type).

- Security related functions: As mentioned in section 2.2, we can use qualifiers to explicitly express security policies we want to enforce. For example, `copy_to_user()` copies the kernel data to the user space. To avoid information leakage because of uninitialized data, we manually create a summary for this function, requiring the source object to be fully initialized.

2.6 Evaluation

Our experiments are systematically performed with the objective of answering the following research questions:

- **RQ1:** Can UbiTect detect previously known bugs?
- **RQ2:** Can UbiTect detect new bugs?
- **RQ3:** Compared with UbiTect, how do other open sourced static analyzers perform for finding UBI bugs in the Linux kernel?

Experimental Setup. To answer these three questions, we first gathered eight previously patched Linux kernel UBI bugs studied in [108] and validate our tool. Then, we apply UbiTect to the x86_64 Linux kernel, version 4.14, with `allyesconfig`. It was tested on machines with Intel(R) Xeon(R) E5-2695v4 processors and 256GB RAM. The operating system is the 64 bit Ubuntu 16.04 LTS.

Data Availability. Linux kernel is an open sourced project. We will also open source UbiTect for aiding the reproducibility of the experimental results.

2.6.1 Detecting Known UBI Bugs

To answer RQ1, we evaluate UbiTect in terms of finding eight previously patched Linux kernel UBI bugs studied in [108]. Table 2.2 shows the results i.e., UbiTect can detect all of them. Two of these bugs can be detected by intra-procedural analysis but the rest require inter-procedural analysis.

Table 2.2: Evaluation I: UBI bugs patched since 2013. All of the uninitialized variables are located on stack. UbiTect can successfully detect all of them.

Commit or CVE No	Type	UbiTect
bde6f9d	intra-procedural	Yes
1a92b2b	intra-procedural	Yes
8134233	inter-procedural	Yes
c94a3d4	inter-procedural	Yes
da5efff	inter-procedural	Yes
CVE 2010-2963	inter-procedural	Yes
7814657	inter-procedural	Yes
6fd4b15	inter-procedural	Yes

2.6.2 Detecting New UBI Bugs

It took UbiTect about a week to fully analyze the entire Linux kernel with 616,893 functions. Specifically, it took 7 and 205 days of CPU time for qualifier inference and symbolic execution (SE), respectively. After qualifier inference, for each function, generated warnings were immediately fed into SE, which ran on more than 30 CPU cores, on average (and was complete in a week of real time). The qualifier inference component generated 147,643 potential uninitialized use of stack variables. Each warning represents a unique use of an uninitialized variable, meaning that **repeated accesses to the same uninitialized variable in different statements and accesses to different fields of the same object** are considered as different warnings. Since our modeling of heap variables is very conservative and the number of warnings for stack variables is already large enough, we exclude the warnings relating to writing uninitialized values to heap variables.

UbiTect's under-constrained symbolic execution (SE) components filtered 4,150 warnings as false positives because it was unable to find a feasible path based on the guidance. 1,190 cases could not be handled by our SE component due to a mixture of 32-bit and 64-bit pointers. We then manually inspected 190 bugs where our SE component can find a feasible path within 2 minutes. 6 of the 190 bugs are due to the use of uninitialized function pointers, 125 are due to use of uninitialized data pointers, and 59 are related to use of uninitialized data (that affect control flow). Our manual analysis confirmed 78 of them as true positives, yielding a false positive rate of 59%. We interpret a reported bug as a false positive if the path returned by SE is infeasible, or if the variable is actually already initialized along the path.

To confirm our manual inspection results with kernel maintainers, we tried to create patches for the 78 true positive cases. During this process, we found that the buggy code of 9 cases have been removed in the mainline due to feature updates and 11 are already fixed in the mainline. We also found that many bugs were related to missing checks over the return value [103] of the function `regmap_read()`. Further (manual) checks over the remaining callers led to an additional 60 bugs. We submitted patches for all the unfixed 118 cases to Linux developers. 52 bugs have been confirmed, 35 cases were categorized as "will not happen in reality," and the remaining 31 are still in process (we are awaiting feedback). The detailed list of the confirmed bugs is shown in Table 2.3. We point out here that among the 52 bugs, 37 of them were reported automatically while 15 are identified from the additional manual check. In fact, if we extend the time and memory limitations for symbolic execution, we expect that these cases can be reported automatically as well.

For 112 warnings we deemed as false positives, we also analyzed the root causes.The major ones include (1) Incomplete black and whitelist (39 cases): this happens when the path crosses multiple functions. (2) Imprecise indirect call resolution (26 cases): this happens the indirect call target is infeasible or incorrect. (3) LLVM optimization (16 cases): this happens wherein LLVM converts a struct with two $u32$ types, directly to a $u64$ type; this optimization makes certain function summaries inaccurate. (4) The limitations of under-constrained symbolic execution: we treat input arguments as unconstrained symbolic values; however, in reality, such unconstrained inputs are impossible according to the program logic (e.g., constraints incurred outside the scope of the symbolic execution which requires additional domain knowledge). and (5) Assembly code (10 cases).

2.6.3 Sensitivity and Precision

We showcase how different sensitivity levels affect UbiTect's qualifier inference. First, we use a simple syntax analysis as the baseline, where we check for stack variables that are **not** initialized immediately after their declaration. This baseline flagged 1,373,174 abstract locations (expanded to be field-sensitive) out of 2,179,399 as not being initialized when declared. If we add flow-sensitive analysis (but without inter-procedural analysis), the number of warnings was 10,267,357.

This number is higher than the baseline in line with what one might expect, because this is on the basis of uses (i.e., different uses will be considered as different warnings) instead of declarations. If we add inter-procedural analysis but without context-sensitivity, the number of warnings was 242,934. After adding context-sensitivity to the analysis, UbiTect's static analysis component reported 147,644 warnings. Again, because each warning from

static analysis is based on a unique use instead of per variable, the reduction rate is actually higher than 90%.

2.6.4 Comparison with other Static Analyzers

To answer RQ3, we compare UbiTect with two open sourced tools which are able to detect UBI bugs. We first compare the performance of UbiTect with that of cppcheck [113]. Both UbiTect and cppcheck need the access to the source code and do not need manual annotations. While UbiTect's static analysis is inter-procedural and reports the warnings at the use site, cppcheck's analysis is only intra-procedural and reports the warning when the uninitialized variable is read. We ran the cppcheck on the whole Linux kernel, version 4.14. It reported 191 UBI bugs, from which 164 bugs were within our analysis scope (i.e., code enabled by `allyesconfig`). Among the overlapped 164 bugs, only 2 are true positives (i.e., a much higher false positive rate of 98%). From these 2 true positives, UbiTect catches only one via its static analysis component; the other is missed by UbiTect because the use site is not explicitly marked by us. Specifically, the uninitialized value is leaked through the network layer but we only explicitly marked `copy_to_user()` to detect potential leaks. 29 false positives are shared between UbiTect's static analysis and cppcheck. The remaining 131 false positives were correctly filtered by UbiTect's inter-procedure static analysis.

Opposite to the cppcheck's lightweight and imprecise analysis, the Clang Static Analyzer (CSA) is another open source tool which applies the expensive and precise symbolic execution to catch UBI bugs. As with any symbolic execution, it is hard to scale to large

Table 2.3: Evaluation II: New bugs detected by UbiTect. The `Line No.` column gives the place where uninitialized uses happens. The last column: A-Patch Applied; C-Confirmed by developers

No.	Sub-System	Module	Variable	Line No.	Patch
1	iommu/amd	iommu.c	unmap_size	1523	A
2	asoc/rt565	rt5651.c	ret	1759	A
3	asoc/rt274	rt274.c	buf	364	A
4	asoc/rt275	rt274.c	val	1133	A
5	net/stmmac	dwmac-sun8i.c	val	646	A
6	clk/gemini	clk-gemini.c	val	320	C
7	iio/adc	meson_saradc.c	regval	286	C
8	iio/adc	meson_saradc.c	regval	313	C
9	iio/adc	meson_saradc.c	val	454	C
10	iio/adc	meson_saradc.c	regval	631	C
11	iio/adc	meson_saradc.c	regval	789	C
12	regulator	pfuze100-regulator.c	val	635	A
13	drm/bridge	sii902x.c	status	122	C
14	iio/trigger	stm32-timer-trigger.c	ccer	136	C
15	iio/trigger	stm32-timer-trigger.c	cr1	140	C
16	iio/trigger	stm32-timer-trigger.c	ccer	168	C
17	iio/trigger	stm32-timer-trigger.c	cr1	173	C
18	iio/trigger	stm32-timer-trigger.c	cr1	222	C
19	iio/trigger	stm32-timer-trigger.c	psc	224	C
20	iio/trigger	stm32-timer-trigger.c	arr	225	C
21	iio/trigger	stm32-timer-trigger.c	dat	411	C
22	iio/trigger	stm32-timer-trigger.c	dat	454	C
23	media	atmel-isc.c	isc_intsr	1255	C
24	media	atmel-isc.c	isc_intmask	1255	C
25	mfd	fsl-imx25-tsadc.c	status	40	C
26	mfd	ti_am335x_tscadc.c	reg	58	C
27	net/ethernet	hns_mdio.c	reg_value	165	A
28	clk/axi-clkgen	clk-axi-clkgen.c	d	314	C
29	power/supply	max17042_battery.c	read_value	485	C
30	power/supply	max17042_battery.c	vfSoc	667	C
31	power/supply	max17042_battery.c	vfSoc	682	C
32	pwm	pwm-stm32-lp.c	val	163	C
33	pwm	pwm-stm32-lp.c	prd	163	C
34	power/supply	max17042_battery.c	full_cap0	681	C
35	power/supply	max17042_battery.c	val	1082	C
36	power/supply	rt5033_battery.c	val	33	C
37	iio/adc	bcm_iproc_adc.c	intr_status	161	C
38	iio/adc	bcm_iproc_adc.c	intr_mask	162	C
39	iio/adc	bcm_iproc_adc.c	intr_status	187	C
40	iio/adc	bcm_iproc_adc.c	ch_intr_status	194	C
41	iio/adc	bcm_iproc_adc.c	channel_status	201	C
42	iio/adc	bcm_iproc_adc.c	val_check	299	C
43	pwm	pwm-stm32.c	psc	100	C
44	pwm	pwm-stm32.c	arr	100	C
45	pwm	pwm-stm32.c	ccer	295	C
46	pwm	pwm-stm32.c	ccer	312	C
47	regulator	ltc3589.c	irqstat	419	C
48	regulator	max8907-regulator.c	val	303	A
49	media	pvrusb2-hdw.c	qctrl.flags	793	A
50	x86/hpet	hpet.c	msg.f2	503	C
51	staging/ddk750	ddk750_chip.c	pll.OD	58	C
52	power/supply	max17042_battery.c	val	837	C

programs due to the path explosion problem. Therefore, CSA only performs inter-procedural analysis within a module. Unfortunately, even without inter-module whole program analysis, it is difficult to scale CSA to all the source code files in Linux kernel. Alternatively, we ran CSA over the 78 files in which our true positives were located. CSA took about 1.5 hours (96m 8.171s) to finish (had it performed inter-module analysis, the time is likely to blow up much more). Because our analysis was performed over 16,163 files in total, at this speed, CSA will run for $\approx$ 13 days to finish analyzing the entire kernel. Within the 78 files, CSA reported only 22 uninitialized variables. 3 were false positives that were filtered by UbiTect. 2 were not reported by UbiTect due to complex assembly which are hard to verify. For the remaining 17 true positives, 12 were within the 78 bugs UbiTect reported in subsection 2.6.2, while the remaining 5 can be verified by UbiTect's SE component with longer times (more than 2 minutes). The majority of the true positives found by UbiTect were not found by CSA; the main reason is that these bugs fundamentally require analysis across multiple modules.

In UbiTect, we take the best of both qualifier inference and symbolic execution. We apply the expensive and precise symbolic execution only selectively under the guidance of qualifier inference, e.g., to go across the boundary of modules (files) and to focus on a subset of all the program paths. This allowed us to discover more vulnerabilities than pure symbolic execution (i.e., more scalable) with better accuracy than pure static analysis (i.e., less false positives).

2.6.5 Threats to Validity

There are three major threats to the validity of our evaluation. First, although the theoretical foundation of type inference is sound, compromises made during the design could affect the soundness of our analysis results and hence, our static analysis component may miss some bugs. Such compromises include imprecise modeling of assembly code, undefined behaviors (e.g., out-of-bound memory access), and data structure padding. The second threat is potential bugs in our prototype implementation. We have used previously known UBI bugs to test our prototype, but the test set is small and thus, could not cover all corner cases. Finally, classifying bugs reported by UbiTect was done by the authors. As we are not Linux kernel maintainers, we could have made mistakes on whether a reported bug is a true positive or false positive. We tried to mitigate this threat by reporting the bugs that we believe were true positives to the kernel maintainers, but we did not hear back for all the cases.

2.7 Related Work

Mitigating UBI Bugs. Automated mitigation of UBI bugs is pioneered by PaX's `STACKLEAK` plugins [129], which forces the initialization of kernel stacks during context switches between the kernel and user space; `STRUCTLEAK` optimizes `STACKLEAK` by only initializing objects that may be exposed to user space. Two recent related works are SafeInit [115] and UniSan [105]. SafeInit [115] is a compiler extension that initializes all allocated memory to zero. However, this blind initialization strategy is often undesired and can mask the real bug. According to our interaction with kernel developers, it is actually believed that in many cases the right

approach is to leave a variable uninitialized when it is first created. The reasoning is that the real initial value will be computed dynamically later anyway; assigning zero or some arbitrary value is not only unnecessary but can also mask a real bug where the desired (correct) initialization procedure fails and the variable gets used subsequently. The correct way to fix such bugs is to make sure that the use-before-initialization path is eliminated (e.g., by properly checking for the absence of initialization and returning). UniSan [105] detects and zeros uninitialized data that can leak from the kernel space. So, it only eliminates information leakage resulting from uninitialized reads. This work attempts to detect all use-before-initialization bugs. For instance, an uninitialized function pointer may be dereferenced in the kernel to cause arbitrary code execution as discussed earlier. At this stage, UBI bugs still need to be patched manually case by case, and we believe that the identification of such bugs with UbiTect is a necessary first step.

Static Detection of Kernel Bugs. With the increasing popularity of LLVM, many LLVM-based static analysis tools have been developed to find bugs in the Linux kernel source. KINT [158] put together a number of static analysis techniques such as taint and range analysis to discover integer overflow vulnerabilities in the Linux kernel. Juxta [118] detects semantic bugs in Linux file systems by finding deviant behaviors in different file system implementations [60]. Dr. Checker [109] is a static taint analysis engine that can be used to find taint-style vulnerabilities in the Linux kernel. K-Miner [67] performs context-sensitive value-flow analysis to identify memory-corruption vulnerabilities. Deadline [167] and Check-it-Again [157] detect a special type of time-of-check-to-time-of-use (TOCTTOU) bugs due to lack of re-checks. CRIX [103] detects missing security checks in the Linux kernel.

PeX [183] detects missing permission checks. To our knowledge, no analysis has attempted to discover the increasing number of UBI bugs.

Type Qualifiers. Type qualifiers have been shown to be a powerful way to represent invariants in programs. A type qualifier is general and expressive enough to conduct a variety of security analysis and bug finding tasks, including the popular taint analysis [83]. Some examples of applying type systems for bug finding include finding user/kernel pointer bugs [87], format string vulnerabilities [137], integer-overflow-to-buffer-overflow [179], null pointer deference bugs [80], lock/unlock bugs and file descriptor bugs [65].

Chapter 3

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

3.1 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 [19]. While these updates provide new features and bug fixes, they can also introduce new bugs and security vulnerabilities.

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 [90, 98, 181, 184]. For example, Cook [90] 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 [6, 9, 15, 21, 111].

To alleviate the aforementioned security risks in the Linux kernel, both dynamic testing and static analysis have been applied. Dynamic testing, and fuzzing specifically [45, 76, 85, 94, 126, 128, 134, 138, 152, 166], 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 [69–71], thousands of bugs have been discovered in the past 4 years using the continuous fuzzing platform maintained by Google [72]. With regards to static analysis, many tools have been developed specifically for the Linux kernel, including commercial ones such as Coverity [46] and academic ones as in [35, 59, 67, 99, 104, 110, 143, 156, 158, 163, 167, 170, 177]. 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 [110] 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 [177] 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 [131], 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 [154] and has been applied recently in industry [46, 77], 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 [46, 77]. 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 [177]. 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.

3.2 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.

3.2.1 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 [20] 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.

3.2.2 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 [110] 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 [133, 174] and C [41, 124, 165, 177]. Some have been shown to be successfully applied to the Linux kernel [41, 165, 177]. Typically such analyses need to decide what kind of information to record in the summary, e.g., points-to information [124], locking behaviors [165], and data flow [41, 177].

3.2.3 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 [16]. 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 [44] and information leakage [106, 116]. Previous work [107] 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. [177] 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 3.1. 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 3.1. 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 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 3.1. 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`

Table 3.1: 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

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. After finding the feasible path, 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.

```c
/* A simplified buggy code from
 * drivers/iio/adc/stm32-dfsdm-adc.c
 * uninteresting code lines are ommited
 */
static irqreturn_t stm32_dfsdm_irq(int irq, void *arg)
{
  struct stm32_dfsdm_adc *adc = arg;
  unsigned int status, int_en;

  regmap_read(DFSDM_ISR(adc->fl_id), &status);
  regmap_read(DFSDM_CR2(adc->fl_id), &int_en);

  if (status & DFSDM_ISR_ROVRF_MASK) {
    if (int_en & DFSDM_CR2_ROVRIE_MASK)
      //do sth here.
  }

  return IRQ_HANDLED;
}
int regmap_read(unsigned int reg, unsigned int *val)
{
  if (reg)
    return -EINVAL;

  *val = some_init_number;
  return 0;
}
```

Figure 3.1: A piece of buggy code that adapted from a real UBI bug in the Linux kernel.

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.

3.3 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.

3.3.1 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 3.2 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, 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.

```
/* 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*/
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;
}
```

Figure 3.2: 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.

3.3.2 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 framework 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 as a taint source, and any use of the tainted variable can be considered the sink (e.g., arithmetic operation, loop bound 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 [87, 110]). Using Dr. Checker's loop bounds checker [110] 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 3.2.3) 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."

3.3.3 System Overview

Having described key design considerations, we next provide an overview of IncreLux with a high-level workflow depicted in Figure 4.4. In particular, the workflow includes a few pre-processing steps followed by the main incremental analysis.

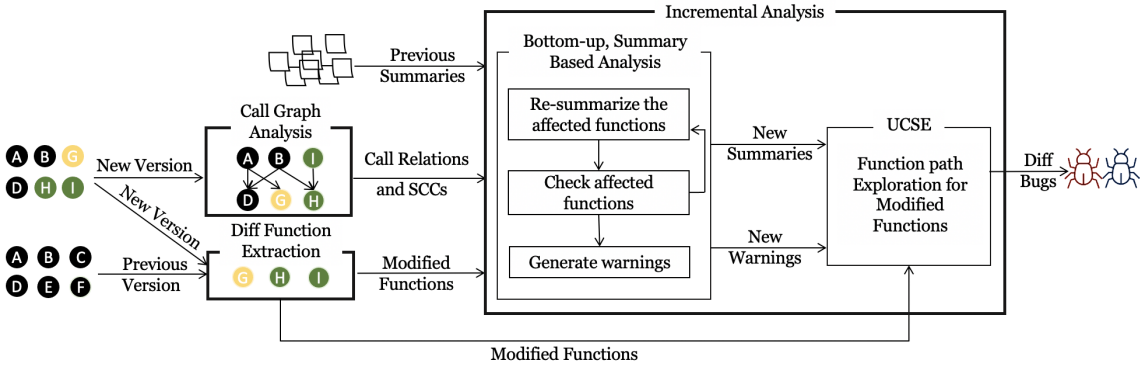

Figure 3.3: Upon obtaining a new version, IncreLux leverages function summaries that were previously computed (in a clean slate version or a prior incremental analysis) to do an expedited incremental analysis. First, warnings relating to potential UBI bugs are generated and then under-constrained symbolic execution is applied to find a potential feasible path to triger the bug.

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 [177] 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.

3.3.4 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 [177] (subsection 3.2.3) 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 *Input data : CG: Callgraph*

Input data : OC: Old version code

Input data : NC: New version code

Input data : OldSum: Old Function Summary

Output data : newFSumSet: New Function Summary

Output data : WS: Warning Set

DiffSet=get_diff_functions(OC, NC)

SCCs=get_order(CG)

for *SC* $\in$ *SCCs* ***do*** *worklist* $\leftarrow \emptyset$

for *func* $\in$ *SC* ***do***

if *func in DiffSet* ***then***

worklist.push(func)

end if

end for

while *is_not_empty(worklist)* ***do***

func = worklist.front()

worklist.remove(func)

oldFSum = get_old_sum(OC, func)

(warnings, newFSum) = analyze_funcs(func)

WS.append(warnings)

newFSumSet.append(newFSum)

if *not_equal(oldFSum, newFSum)* ***then***

```
            callerSet ←get_callers(func)
            for caller ∈ callerSet do
                if caller ∉ worklist and caller ∈ SC then
                    worklist.push(caller)
                    continue
                end if
                if caller ∉ worklist then
                    DiffSet.push(caller)
                end if
            end for
        end if
    end while
end for
```

Tracking Warnings Changes. After the above steps, warnings are generated automatically. We retain the rules for generating warnings from [177] 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 §section 3.4, 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 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 3.5) that in practice this case is very rare.

3.4 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 while enabling debug information (-g). This stemp ensures that we have the necessary 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 [158], the type-based analysis from Unisan [106], and a hybrid of these two methods used in multi-layer type-based analysis [102]. We chose the points-to based

algorithm from [158]. 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 the potential to overlook valid indirect call targets. We plan to investigate approaches that allow us to leverage the type-based methods and yet still maintaining the ability 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 [177] 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 §section 3.3, 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 §section 3.3, 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.

3.5 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.

3.5.1 Evaluation Scope

In accordance with the Linux kernel development guide [93], 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 to be submitted to the code base. Once this two-week window elapses, 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 §section 3.2, 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.

3.5.2 Speed Improvement Analysis

Incremental Analysis for the Mainline Versions.. Table 3.2 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 chapter 5, 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 [2]. 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 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 3.3 and Table 3.4 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 3.4 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 3.4. As we can see, the accumulated analysis time is proportional to the number of analyzed functions.

(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.

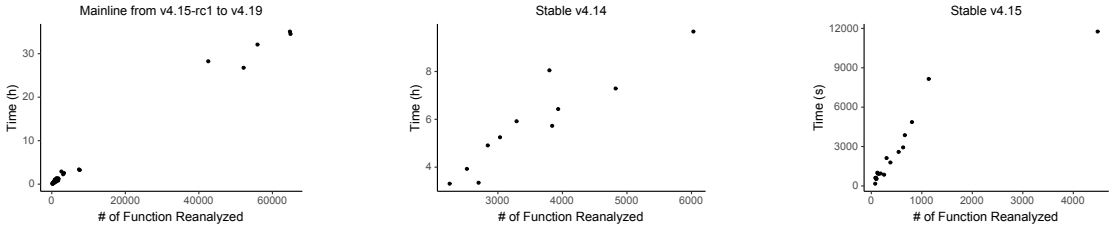

Figure 3.4: The incremental results for different versions.

Incremental Analysis for Each Patch. Our incremental analysis can serve as a valuable tool to perform regression check for individual commits. This functionality proves particularly beneficial for individual developers who might be submitting commits to get quick feedback on whether their changes might 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 [177]. We performed the incremental analysis for each patch (using its immediate predecessor commit as the baseline), and the results are shown in Table 3.5. Incremental analysis demonstrates its ability to rapidly complete the verification process for each patch, showing the successful resolution of the targeted bug without introducing new UBI (Unintended Behavioral Issues) concerns. 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 posit that this near-instantaneous feedback holds significant potential for enhancing the efficiency of kernel development.

To summarize, our experiments show that IncreLux yields substantial speedups in a variety of scenarios. Even in the case of an rc1 release with more than 20k function changes, IncreLux runs faster than an exhaustive analysis. When dealing with changes that affect a smaller number of 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.

3.5.3 Time Breakdown

In Figure 3.5, 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 [177]: 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 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.

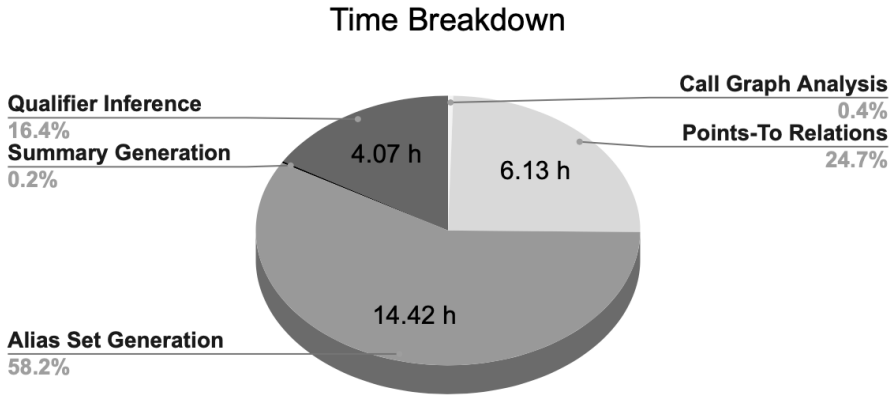

Figure 3.5: The time distributions for different analysis phase along the incremental analysis from v4.14 to v4.15-rc1.

3.5.4 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.

3.5.5 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 3.6.

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 [20], 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 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.

3.5.6 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

```
1  /* drivers/media/i2c/imx274.c
2  * uninteresting code lines are omitted */
3  static int imx274_regmap_util_write_table_8 ()
4  {
5      int err;
6      if (range_count == 1)
7                  err = regmap_write(regmap,
8                          range_start, range_vals[0]);
9      else if (range_count > 1)
10                 err = regmap_bulk_write(regmap, range_start,
11                                 &range_vals[0],
12                                 range_count);
13 +    else
14 +        err = 0;
15     if (err) {
16         return err;
17     }
18 }
```

Figure 3.6: 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.

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.

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 3.6, 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.

3.6 Related Work

3.6.1 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 [158] is a static analysis tool designed for detecting integer overflow bugs in the Linux kernel. K-Miner [67] performs an inter-procedural analysis to detect memory-corruption vulnerabilities. UniSan [106] adopts byte-level data-flow analysis to detect information leakage caused by uninitialized variables. UBITect [177] 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 [167], 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 [99], which detects undefined behavior bugs. LRSan [156] aims to find a broader spectrum of TOCTTOU bugs that can bypass kernel security checks. CRIX [104] detects insufficient handling of erroneous states in the Linux kernel (e.g., forgetting to check if `kmalloc` returns `NULL`). K-MELD [59] 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 [119] finds semantic bugs in Linux file systems by cross-checking paths that handle the same type of errors. RID [112] and [143] detect reference count bugs using consistency checks across error handling paths. EeCatch

[127] aims to detect error handling code that causes the kernel to enter a state that is even worse than the error itself. HERO [163] finds bugs in error return paths that perform cleanup operations in an incorrect order, redundantly, or inadequately.

Coverity [46] 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 [46]. b) It also appears that it leaves some expectations on customers [43] 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 [110] 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 [180] 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 [66] 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 [177, 179]. Some frameworks [22, 32, 91, 125, 176] also use intra-procedural analysis to analyze Linux.

Dynamic Analysis Tools for the Linux Kernel. Dynamic analysis is a widely used approach to catch bugs in the Linux kernel at runtime. This encompasses various techniques, including hypervisor-based detection and fuzzing. Since the hypervisor can effectively monitor the guest OS kernel, it can be used to dynamically catch bugs in the kernel. For example, bochspwn-reloaded [88] is capable of uncovering memory disclosure bugs in the Linux kernel through the tracking of sensitive locations, which uses 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 [152], which has been used by Google to perform continuous fuzzing for all versions of the Linux kernel [72]. Many research prototypes have also been developed to improve kernel fuzzing. IMF [76] tries to infer syscall dependencies from real-world applications. MoonShine [126] tries to improve the quality of initial seeds by "distilling" seeds from syscall traces of real-world applications. HFL [94] combines kernel fuzzing with symbolic execution. Difuze [45] uses static analysis to help construct well-formatted inputs to fuzz kernel drivers. Razzer [85] also uses static analysis to guide the discovery of kernel data-races. KRace [166] uses dynamic data-flow in addition to code coverage to fuzz data race bugs in the file systems. Janus [169] improve file system fuzzing by mutating both the syscalls and the on-disk file system. PeriScope [138] 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.

3.6.2 Incremental Analysis and Regression Test

Relatively little attention has been paid to incremental analysis of the Linux kernel. Facebook Infer [77] 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 [26]. 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 [3]. 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 [74] 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 [68] 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 [37] 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).

3.6.3 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 [48, 73, 186] classify security vulnerabilities and general bugs based on commit message text. There exist other statistical approaches [160] 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 [161] 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, [161] can automatically identify security patches and even infer the type of the security vulnerability associated with a patch. As shown in section 3.5, IncreLux can very quickly verify if a patch adds or removes UBI bugs.

3.7 Discussion

IncreLux uses a principled way to conduct an incremental analysis for the Linux kernel. In pursuit of enhanced scalability and reduced turnaround time, IncreLux 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 have been conducted on the allyes configuration, it's worth noting that this approach remains adaptable for use 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 [177]. 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.

3.8 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× 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.

Table 3.2: 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

Table 3.3: 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 3.4: 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 3.5: 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

Table 3.6: 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.** stands for Information Leakage. **MC.** stands for Memory Corruption. **B.** stands for Benign. **HWCC.** stands for 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 3.7: 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 3.8: 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 3.9: 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

Chapter 4

Don't Waste My Efforts: Pruning Redundant Sanitizer Checks of Developer-Implemented Type Checks

4.1 Introduction

Type confusion [25] represent a critical category of software vulnerabilities, especially prevalent in weakly-typed programming languages such as C/C++. This kind of vulnerabilities occur when the program allocates or initializes a resource, such as a pointer or object using one type, but subsequently accesses that resource using a type that is incompatible with the original type. When a program experiences type confusion, it can

result in undesirable behaviors such as a system crash [49], information leakage [52], and even dangerous arbitrary code execution [50].

To mitigate the threats introduced by type confusion vulnerabilities, both static and dynamic analyses have been applied. For example, static type inference and pointer analysis have been developed to determine if a type cast is safe [40]. However, static approaches can generate a large number of false positives because of the inherent imprecision and over-approximation in static analysis.

Besides static analysis, sanitizer-style runtime checks have also been proposed [57, 75, 84, 97, 145, 147]. The basic idea is simple. Instead of checking for type compatibility at compile time, the sanitizer instruments the program and tracks the type of objects at runtime in order to perform precise type checks. Clearly, such approaches are more precise than static approaches, but are also more expensive as they incur runtime overhead. Although these type confusion sanitizers are faster than C++'s built in `dynamic_cast`, their runtime overhead is still too high for deployment in production systems.

One promising direction to reduce runtime overhead is to statically recognize safe casts and avoid instrumentation selectively. For example, CaVer [97] and HexType [84] track data flow via pointers from object allocation sites to cast sites; if the cast type is a supertype of the allocation site type, then the corresponding runtime check can be eliminated. However, as discussed in [40], static analyses are in general imprecise and will often make conservative inferences, i.e., treating a safe type cast as potentially unsafe. For example, pointer analysis, relied upon by CaVer [97], is a known hard problem in static analysis. As a result, many unnecessary runtime checks cannot be eliminated.

In this paper, we make the observation that **pointer analysis is not the only way to show a cast is safe**. In particular, we find that developers often encode custom runtime type information (RTTI) directly into a structure or class, especially in complex C++ class hierarchies, to facilitate their own type checks. For example, Chrome defines a class `BasicShape` and many classes inherit from this class, such as `BasicShapeCircle`, `BasicShapeEllipse` and `BasicShapePolygon`. Those subclasses override a virtual function `GetType()`, which returns enumeration constants `kBasicShapeCircleType`, `kBasicShapeEllipseType`, and `kBasicShapePolygonType` respectively. In addition, we find that before casting a pointer `BasicShape *basic_shape` to a subclass, developers usually insert a type check using `GetType()`, e.g., checking that `basic_shape->GetType() == kBasicShapeCircle` before casting to `BasicShapeCircle`. Such a check ensures the downcast is safe – often voluntarily inserted to avoid type confusion bugs [28, 29, 31].

Our key insight is to leverage these developer-implemented type checks to discover opportunities to remove redundant sanitizer checks for type casts. Achieving this goal requires addressing two key challenges. First, we must **identify** developer-encoded RTTI and the corresponding type checks, and **validate** the correspondence between these checks and the class hierarchy. Developer-written type checks may involve arbitrary logic and can vary from one class hierarchy to another, and the source code does not explicitly identify which logic serves as a type check. To this end, we conduct an exploratory investigation of real-world programming conventions in Chrome, distilling them into several general patterns. The most-common patterns encoded custom RTTI as a set of predefined values (i.e., constants) in class definitions. By performing a class-hierarchy-wide static analysis,

we could track the definitions and uses of such values and thereby automatically deduce developer-inserted custom RTTI and type checks. Note that our analysis **verifies** the correspondence between the custom RTTI and the type hierarchy – each class needs to take a unique value, allowing it to be distinguished from others in the same hierarchy.

The second key challenge is to discover where a type cast is correctly guarded by a custom type check, thereby proving the cast cannot fail and enabling the removal of redundant sanitizer checks. We tackle this challenge by using a flow- and field-sensitive intra-procedural analysis to track the refined type suggested by developer-implemented type checks, validating if a downcast is always safe under all execution paths.

Based on the above insights, we implemented our techniques in an automated solution called T-Prunify, that conducts static analyses for C++ programs to (1) extract custom RTTI based on understanding and surveying diverse class hierarchies in popular applications, (2) identify developer-implemented type checks based on the identified type information, and (3) further validate the developer-implemented type checks. In addition, T-Prunify also uses the results to prune unnecessary sanitizer checks. Our analysis is designed to be sound with multiple verifications throughout the process and will conservatively prune sanitizer checks only when the developer-implemented checks are deemed safe. We evaluated our solution on the Chromium browser (the open-sourced version of Google Chrome), one of the largest C++ programs, which is also known to be prone to type-confusion vulnerabilities [50–55]. The results showed T-Prunify can prune a large number of sanitizer checks safely and reduce the runtime performance overhead by 1.34× - 3.98× compared to the state-of-the-art sanitizer-based solution with other forms of check pruning.

In summary, our main contributions of this paper are:

- We identify developer-inserted custom runtime type checks as a previously-overlooked source of opportunity to reduce the performance overhead of type confusion mitigation techniques.
- We develop a custom solution to (1) automatically identify developer-implemented custom runtime type checks and (2) leverage them to prove the safety of type casting in C++ programs.
- We develop T-Prunify that packages the insight into a fully-automated system, which we plan to open source to facilitate further research in this direction.
- We evaluate T-Prunify against a state-of-the-art type sanitizer Hextype [84], and showed that the relative runtime overhead reductions range from 25% to 75% when evaluating Chrome under standard benchmarks.

4.2 Background and Motivation

In this section, we start with some basic background relevant to type confusion. First, we describe the C++ type hierarchy, casting operations and type confusion vulnerabilities. Then we illustrate, with examples, how previous type confusion checks are performed statically or dynamically (including sanitizer checks executed at runtime), and why they are inefficient. This will then motivate the design and implementation of T-Prunify.

4.2.1 Type Casting in C++

C++ is an object-oriented programming language, which allows programmers to define new types as classes. A class can inherit from multiple ancestor classes. Descendant classes inherit members (methods and variables) from their ancestor(s) and can optionally define additional members [140]. Generally speaking, upcasts (i.e., casting a pointer of a derived class to a pointer of an ancestor class) are considered safe, because the memory scope an ancestor pointer can access is strictly smaller than the memory scope of a descendent class; however, downcasts (i.e., casting a pointer of an ancestor class to a pointer of a descendent class) may introduce memory corruption vulnerabilities, when the underlying allocated memory object has a smaller memory scope than the destination type demands. Figure 4.1 illustrates such an example; the code allocates a pointer of the base class and subsequently casts it to the derived class (which is always at least as big as the base class). In this example, the derived class includes an additional field `y`, and accessing this field on the improperly-casted `Base` pointer leads to an out-of-bounds memory access.

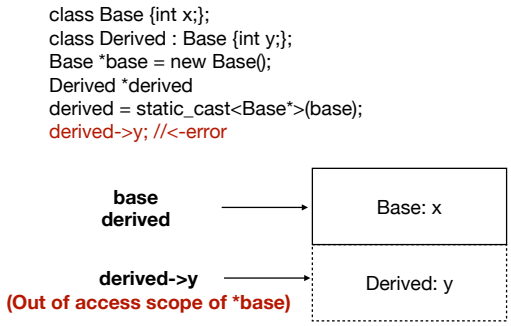

Figure 4.1: A code example and diagram of a type confusion problem where a base class is incorrectly accessed using a pointer to a dereived class. The static_cast results in type confusion and accessing the field y is out of the access scope of type `Base`.

C++ provides four built-in type casting operations, including `reinterpret_cast`, `static_cast`, `dynamic_cast`, and `const_cast`, which we describe below:

`reinterpret_cast<dest>(src).` is similar to an explicit cast in C, which allows conversion between any two arbitrary types, regardless of their compatibility. Although this primitive grants flexibility, the lack of safety checks can lead to type confusion bugs.

`static_cast<dest>(src).` casts a pointer/object of `src` type to a pointer/object of the `dst` type. Unlike `reinterpret_cast`, `static_cast` performs lightweight compile-time type checking to avoid bad casting. Specifically, the compiler will verify whether the source class and the destination class are within the same class hierarchy. However, it cannot detect an unsafe downcast and thus, can still result in type confusion bugs.

`dynamic_cast<dest>(src).` can avoid unsafe downcasts because it performs a runtime type check to make sure the allocation type of the source object is actually compatible with the destination type, using C++'s own runtime type information (RTTI). To perform a runtime type check, `dynamic_cast` first locates the RTTI of the source object from a pointer stored in its virtual function table. The RTTI contains a null-terminated byte string of the mangled type name as the type information of the current type, and one or more pointers to its base classes' RTTI. To check for type compatibility, `dynamic_cast` recursively compares the mangled name of all base classes of the source object with the mangled name of the destination type. If a match is found, the casting is valid and `dynamic_cast` returns a valid pointer; otherwise a null pointer is returned. Since the type check involves slow string comparison and possible recursive traversal of the base classes' RTTI, using `dynamic_cast` operators can be 90 times slower than using `static_cast` [97]. Due to its performance overhead, `dynamic_cast`

is intentionally avoided in release builds of well-developed applications such as Chrome.

`const_cast<dest>(src).` simply drops the `const` qualifier of the source object. Since it does not actually modify the type itself, it is not of interest to us.

For backward compatibility, C++ compilers also support C-style explicit casts. When encountered, the compiler will try the following sequence of casts one after another until the program can be compiled without an error: (1) `const_cast`, (2) `static_cast`, and (3) `reinterpret_cast`.

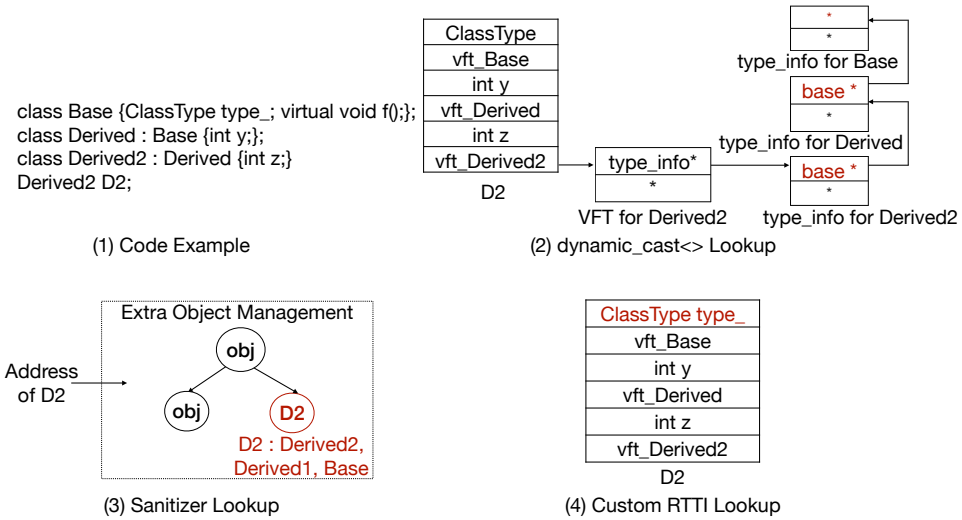

Figure 4.2: The code example and three different methods for verifying the safety of a type cast.

4.2.2 Type Confusion Sanitizers

Type confusion sanitizers [75,84,97] aim to overcome two limitations of `dynamic_cast`: (1) its high runtime performance overhead and (2) its limited protection scope (i.e., it only supports classes with virtual function tables. To reduce the runtime overhead, type confusion sanitizers typically employ two optimizations. First, instead of storing the type information

as a string, they store a unique hash (guaranteed at link time) of the mangled name, so that a compatibility check can be done using an integer comparison instead of a string comparison. Second, they compact all base classes' type information into a single RTTI, where the hash values are also sorted for faster binary search. To overcome the second limitation, type confusion sanitizers store the RTTI of an object in a disjoint lookup table. To perform runtime type checks, type confusion sanitizers first extract the type hierarchies during compile time and emit RTTI that records all the compatible types of a class (i.e., all ancestor classes, including itself). At runtime, after an object is allocated, type confusion sanitizers associate the object with its RTTI (e.g., by using a hash table where the key is the address of the object and the value is the RTTI entry). At the cast site, the sanitizer retrieves the RTTI and looks for a matching hash value among all compatible types with the destination type.

4.2.3 C++ Casting with Custom Run Time Type Information (RTTI)

Although designers of type confusion sanitizers have spent lots of effort trying to reduce the runtime overhead, these sanitizers still introduce significant overhead. For example, during our evaluation, a state-of-the-art type confusion sanitizer HexType [84] still introduces a $10.2\% - 65.7\%$ overhead on Chromium benchmarks. A large portion of this overhead comes from its lookup of RTTI. Specifically, to maintain binary compatibility, type confusion sanitizers use decoupled RTTI (i.e., the RTTI of an object is stored disjointedly at another memory location). As a result, they need extra steps to find the RTTI. In practice, large C++ projects like the Chromium browser and the LLVM compiler frameworks prefer developer-inserted custom RTTI and type checks to prevent type confusion vulnerabilities.

Because such RTTI is tightly coupled with an object (e.g., as a member field or a special virtual function), runtime type checking is much more efficient than both `dynamic_cast` and type confusion sanitizers. Figure 4.2 shows the differences between three different approaches (`dynamic_cast`, type confusion sanitizers, and developer-inserted RTTI) to verify the safety of type casts.

In fact, during our investigation, we found that most of the type confusion vulnerabilities in the Chrome browser are fixed by inserting developer-implemented type checks with custom RTTI [28, 29, 31]. This motivates us to develop T-Prunify. For instance, CVE-2021-30561 [51] is a type confusion vulnerability in Chrome. It allows a remote attacker to potentially exploit heap corruption via a crafted HTML page. Figure 4.3 shows the vulnerabilities and the main patch. Inside function `WasmJs::InstallConditionalFeatures()`, variable `maybe_webassembly` is retrieved as a type of `Object`; it is then directly cast into a `JSObject` and used later. However, the object retrieved could in fact be of types other than `JSObject`, which causes a type confusion vulnerability. The patch fixed the bug by adding a custom type check `webassembly_obj->IsJSObject()` at line 15 to ensure that the type is of `JSObject` before proceeding to the subsequent type cast.

4.3 Overview of T-Prunify

When developers have incorporated custom runtime type information (RTTI) into their C++ classes and have implemented their own type checks before casting, our objective is to pinpoint and eliminate unnecessary type confusion sanitizer checks. This streamlining process reduces the overhead imposed by sanitizers.

```
/* Main patch for CVE-2021-3056,
 *  * uninteresting code lines are ommited.
 *   */
void WasmJs::InstallConditionalFeatures(Isolate* isolate,
Handle<Context> context) {
-  Handle<JSObject> webassembly = Handle<JSObject>::cast(
-    maybe_webassembly.ToHandleChecked());
+  Handle<Object> webassembly_obj;
+  if (!maybe_webassembly.ToHandle(&webassembly_obj)) {
+    // There is not {WebAssembly} object.
+    // We just return without adding the
+    // {Exception} constructor.
+    return;
+  }
+  if (!webassembly_obj->IsJSObject()) {
+    // The {WebAssembly} object is invalid.
+    // As we cannot add the {Exception}
+    // constructor, we just return.
+    return;
+  }
+  Handle<JSObject> webassembly = Handle<JSObject>::cast(webassembly_obj);
}
```

Figure 4.3: The simplified patch for CVE-2021-30561

To this end, we design and implement a lightweight static analysis tool named T-Prunify to achieve this goal.Figure 4.4 depicts the workflow of T-Prunify. It consists of two high-level components, which are briefly described below:

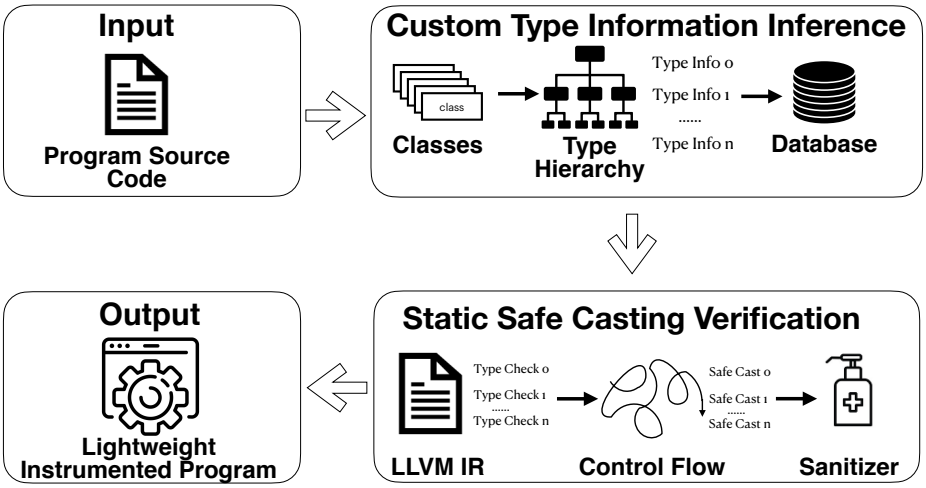

Figure 4.4: Work flow of T-Prunify.

1. Custom Type Check Inference.. In this step, T-Prunify takes the source code of the target program as input and attempts to recognize all the custom type checks inserted by developers. To do so, we first infer custom runtime type information (RTTI) encoded by developers by analyzing class definitions. The challenge lies in the lack of a unified standard for encoding or annotating such custom type information, which can vary across modules and class hierarchies in the program. Therefore, we need to have a comprehensive understanding of the various patterns developers can choose to encode custom types. In our solution, we perform an offline manual investigation of various class hierarchies in Chrome, and summarize them into three common patterns. With these patterns, T-Prunify performs static analysis to automatically recognize and validate custom RTTI in classes, and then stores them in a

database. Based on the identified custom RTTI, we then recognize developer-implemented type checks (e.g., in the form of `if` statements) as operations over the type information.

2. Static Safe Casting Verification.. Based on the identified custom type checks, we then try to prove statically whether a downcast is always safe (i.e., the destination class is always compatible with the class indicated by the type check under all execution paths). The challenge is that type casts may not be performed immediately after a type check, and furthermore the type check may not always be correct. To tackle this challenge, T-Prunify employs an intra-procedural flow- and field-sensitive static analysis to track the refined type suggested by developer-implemented type checks, and validate if a downcast will be safe. Once T-Prunify determines a cast is safe, it will instruct a sanitizer to not insert another redundant type check at compile time. In the end, the output is a hardened program that enjoys the same level of security guarantee but with a much lower runtime overhead.

4.4 Custom Type Check Inference

As mentioned above, T-Prunify uses existing developer-inserted custom type checks to eliminate type confusion sanitizer-induced checks. In this section, we describe how to identify developer-inserted custom type checks.

4.4.1 Systematic Investigation

Developers can choose to explicitly encode the runtime type information (RTTI) in a class definition directly. However, such encoding can be ad-hoc as it is entirely up to the developers to design a scheme to differentiate types. To understand how custom RTTI

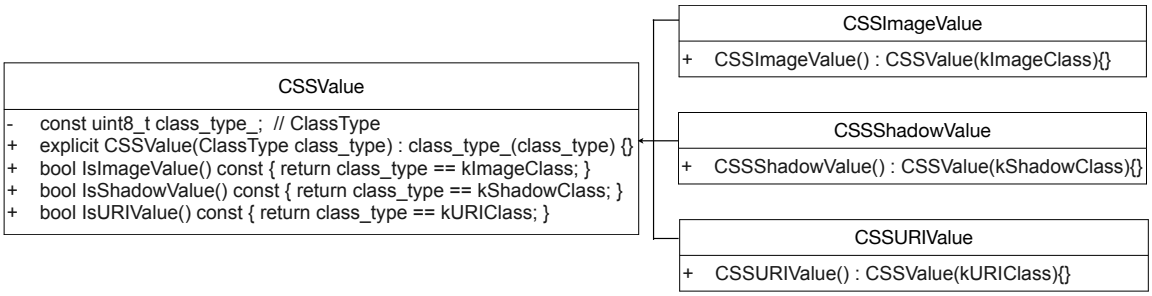

Figure 4.5: An example in Category 1: type information is stored in a member field, which is initialized with a different enumeration constant. Arrow indicates the inheritance relationship.

are commonly encoded, we conduct a manual investigation of over 100 Chrome hierarchies and identified several common categories as follows.

Category 1: Custom RTTI encoded in a Base Class Field.. One way to store type information in C++ is by using a field defined in the base class, which is initialized to a different and unique value in each subclass according to its concrete type. Figure 4.5 shows part of the type hierarchy that class `CSSValue` belongs to. Class `CSSValue` defines a private member called `class_type_`, which is initialized in the constructor. Three subclasses `CSSImageValue`, `CSSShadowValue`, and `CSSURIValue` initialize this member field through their respective constructors with different values. In this example, the three values, e.g., `kImageClass` are unique enumeration constants. The base class `CSSValue` also defines three utility functions for type checking `IsImageValue()`, `IsShadowValue()`, and `IsURIValue()`.

Category 2: Custom RTTI as a Constant Without Fields.. Classes in this category override a virtual method defined in the base class to return different constants to indicate

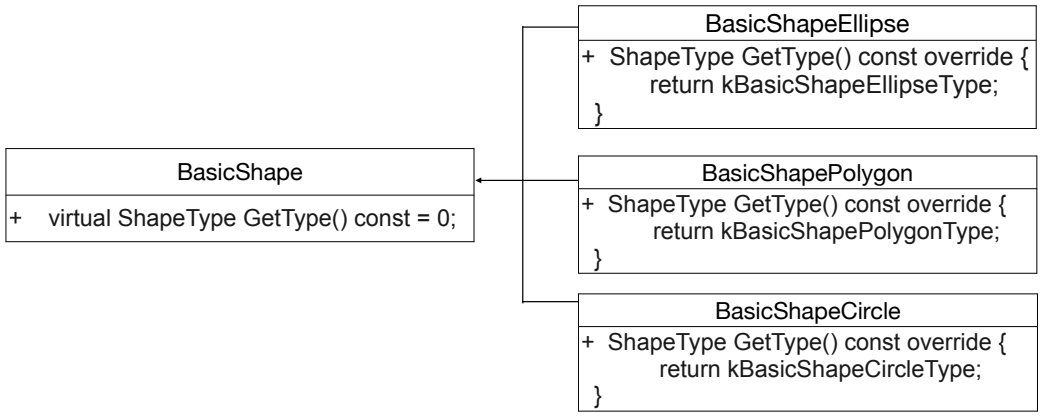

Figure 4.6: An example in Category 2: the type information returned by a virtual method is overridden in each subclass to return a different enumeration constant.

the actual type of the object. Consider the class hierarchy of class `BasicShape` shown in Figure 4.6, it defines a virtual method `GetType()`. Each subclass overrides this method by returning a different enumeration constant that uniquely identifies its object type. For example, the subclass `BasicShapeCircle` overrides `GetType()` inherited from the base class to return `kBasicShapeCircleType`, which is unique to this subclass.

Category 3: Custom RTTI as a Type Check Function.. Classes in this category do not use enumeration constants to indicate custom types; instead, they define custom type check functions. Figure 4.7 illustrates such an example where a base class `CanvasImageSource` defines a number of virtual methods, like `IsVideoElement()` and `IsCanvasElement()`, which returns `false` by default. The two subclasses: `HTMLVideoElement` and `HTMLCanvasElement` each override the corresponding type check method to return `true`. When the `IsVideoElement()` method is called, it returns `true` only if the object is of class `HTMLVideoElement`. This method, along with its return value `true`, serves as a way to uniquely identify the object type.

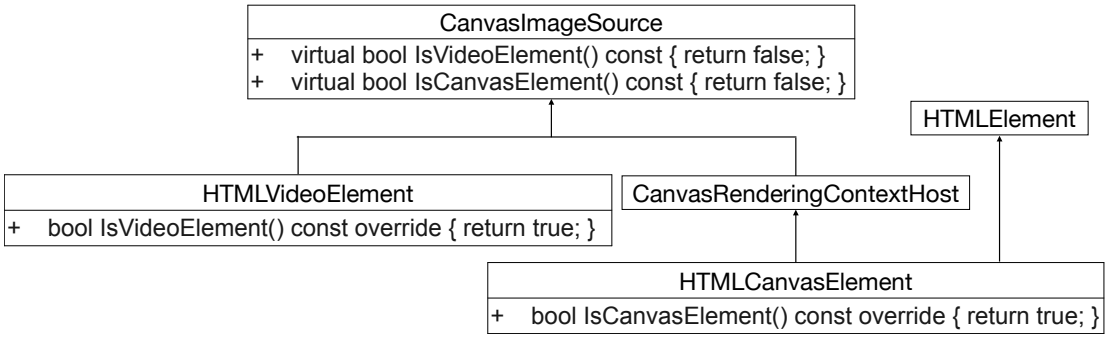

Figure 4.7: An example in Category 3: type information is encoded as the return value of a custom type checking function that is overridden to return `true` in the corresponding subclass.

4.4.2 Custom RTTI Identification

As the foundation of our solution, we need to construct a precise database that contains custom RTTI for a class hierarchy (i.e., a group of classes that share a base class), referred as **"class signatures"**, that can uniquely identify a class/type in the hierarchy. In other words, we will construct a map between a class signature and the actual (allocation) type. More specifically, we scan all the source code files (including header files) and look for class hierarchies that match the aforementioned three categories. For the first two categories, we check the following conditions at the syntax level:

1. a unique enumeration constant is either assigned to a member field of each class in the hierarchy, or returned by a virtual method;

2. if a member field is assigned with the constant, the assignment should happen inside the constructor and the field should not be modified once it is initialized.

For the third category, we use the following heuristics:

1. each class in the hierarchy overrides a unique method and changes its return value.
2. each class should have at least one overridden method that returns a unique value not seen in other classes in the hierarchy.

For each class hierarchy (classes that share a base class), we store the unique constant and method (collectively considered as the class signature) and its corresponding class type in our database. Note that we find sometimes a class hierarchy can have a subset of classes with signatures while the remaining classes do not have signatures by design. In other words, there may be a "sub-class-hierarchy" within a complete hierarchy that encodes class signatures. To accommodate such cases, we effectively look for such sub-class-hierarchies. As long as the heuristics described above apply to the sub-class-hierarchy, we still infer that it has encoded class signatures.

4.4.3 Custom Type Check Identification

Given the database, we can perform an analysis to find the type checks in the target program. In general, any statement that uses the custom type information to control the program flow is considered a type check. More specifically, we look for statements that are of the form of `type == c` (including `switch` cases), where the left-hand side can be any expression that evaluates to a previously-recognized type-indicating member variable or type-indicating method, and the right-hand side is a constant. Given that we have mapped each unique constant to a corresponding class type, we can tell exactly which type is checked for in the statement.

```
/* Some code snippet for use of custom type information,
* uninteresting code lines are ommited.
* */
std::unique_ptr<Shape> Shape::CreateShape(const BasicShape* basic_shape) {
  std::unique_ptr<Shape> shape;
  switch (basic_shape->GetType()) {
    case BasicShape::kBasicShapeCircleType: {
      /*To<> is implemented as a static_cast<>*/
      const BasicShapeCircle* circle = To<BasicShapeCircle>(basic_shape);
      //...
    }
    case BasicShape::kBasicShapeEllipseType: {
      const BasicShapeEllipse* ellipse = To<BasicShapeEllipse>(basic_shape);
      //...
    }
    case BasicShape::kBasicShapePolygonType: {
      const BasicShapePolygon* polygon = To<BasicShapePolygon>(basic_shape);
      //...
    }
  }
}
```

Figure 4.8: The simplified code for class `BasicShape`.

Figure 4.8 shows an example of type checks relating to class `BasicShape` which is described in Figure 4.6; the type check compares `basic_shape->GetType()` against enumeration constants that we record in the database (e.g., `kBasicShapeCircleType`), and then jump to different program branches. T-PRUNIFY can identify each switch case as a type check, and determine that the type of object `basic_shape` is `BasicShapeCircle` between line 7 and line 11. We also discuss how Chrome sometimes uses custom C++ template expressions to perform type checks in §section 4.6.

Note that we find rare cases where a type check would not look like a straight equality comparison in the form of `type == c`. Instead, it can use inequality comparisons, e.g., `>=` or `<=`. We do not currently recognize such type checks and leave them as future work.

4.5 Static Safe Casting Verification

After constructing the database of custom type information and identifying the custom type checks in the program, the next step is to determine whether the type casts are actually safe, i.e., sufficiently protected by those checks. At a high level, given a type check, we analyze all the type cast statements that are dominated by the type check. If the destination type in the type cast statement is compatible with the checked type, we consider it a safe type cast. Algorithm **??** illustrates the high-level procedure.

There are several considerations in performing such an analysis. First, type casts do not always happen immediately after a type check, and the pointer used for the type check may not be the same one that is used for type casts. Therefore, we perform a flow-sensitive

```
/* Some code snippet for use of custom type information,
* uninteresting code lines are ommited.
* */
std::unique_ptr<Shape> Shape::CreateShape(const BasicShape* basic_shape) {
  std::unique_ptr<Shape> shape;
  switch (basic_shape->GetType()) {
    case BasicShape::kBasicShapeCircleType: {
      /*To<> is implemented as a static_cast<>*/
      const BasicShapeCircle* circle = To<BasicShapeCircle>(basic_shape);
      //...
    }
    case BasicShape::kBasicShapeEllipseType: {
      const BasicShapeEllipse* ellipse = To<BasicShapeEllipse>(basic_shape);
      //...
    }
    case BasicShape::kBasicShapePolygonType: {
      const BasicShapePolygon* polygon = To<BasicShapePolygon>(basic_shape);
      //...
    }
  }
}
```

Figure 4.9: Different corner cases need to be considered when proving when a type cast is safe.

and field-sensitive intra-procedural pointer analysis to make sure that the pointer points to the same object at the type check and the type cast. For example, the casting at line 13 in Figure 4.9 is safe because `case1` and `base` point to the same object.

Second, the pointer analysis needs to consider different execution paths and summarize possible types along all execution paths. Therefore, T-PRUNIFY will only determine a casting as safe if all types are compatible with the destination type. For example, T-PRUNIFY cannot prove that the casting at line 20 in Figure 4.9 is safe. This is because `case2` may point to both an object of `Sub` type due to the type check at line 9, and an object of `Base` type due to the pointer reassignment at line 15. As not all aliased objects are compatible with the destination type `Sub*`, the casting is potentially unsafe and needs an additional runtime check.

Third, it is possible that the type check is insufficient in protecting subsequent type casts, e.g., the target of the type cast is not compatible with the type checked. Therefore, T-PRUNIFY does not blindly trust a custom type check; it also performs a type compatibility check to ensure that the refined type suggested by the type check is indeed compatible with the destination type of cast. If T-PRUNIFY cannot determine the casting is safe, it acts conservatively and will not eliminate a sanitizer check. This is because such cases can potentially be real type confusion bugs. For example, T-PRUNIFY cannot prove the casting at line 24 in Figure 4.9 as safe, because the check at line 9 only indicates `case3`, which is a must-alias with `base`, is of type `Sub*`, which is not compatible with type `SubSub*`.

Fourth, the analysis should consider multiple type checks that gradually narrow down the type to a more specific type (i.e., subclass). This also means the analysis should

be flow-sensitive. For example, the casting at line 29 in Figure 4.9 is safe, because after the check at line 25, the type of `base` is further narrowed down to `SubSub*`. However, without a flow-sensitive analysis, one cannot be sure `base`'s type must be `SubSub*`.

Finally, type casts can be performed in a separate function from the function that performed the type check. For example, the type cast may happen in a callee of a caller that performs the type check. In such cases, we cannot conclude that the type cast is safe simply because one caller has performed a safe type check. Instead, we need to analyze all callers to make sure safe type checks are always present before the type cast. In our current design, we perform only an intra-procedural analysis that makes sure the type cast happens in the same function that performs the type check. We consider this a conservative solution and will extend it to the inter-procedural case in the future.

Algorithm 2 summarizes the overall procedure in more detail. In the end, once T-PRUNIFY finds a safe casting, it will inform an existing type confusion sanitizer not to emit redundant type checks during the compilation.

Algorithm 2 *Input : F: Function Input : CRTTI: Customize Runtime Type Information Output : SCSet: SafeCastsSet*

for *Inst* $\in$ *F* ***do***

  if *isConditionInst(Inst)* ***then***

    condition $\leftarrow$ *getCondition(Inst)*

    class $\leftarrow$ *getClass(condition, CRTTI)*

    if *class* $\neq$ *null* ***then***

      classPtr $\leftarrow$ *getClassPtr(condition)*

      compatibleClass(classPtr) $\leftarrow$ *class*

```
    end if
end if
if isCastInst(Inst) then
    srcType ← getCastSrcType(Inst)
    dstType ← getCastDstType(Inst)
    classPtr ← getClassPtr(Inst)
    ptsSet ← getPtrSet(Inst)
    cType ← compatibleClass(classPtr)
    if pstSet ≠ ∅ then
        for object ∈ ptsSet do
            objectType=getType(object)
            if not_compatible(objectType, dstType) then
                continue
            end if
        end for
    end if
    if cType ≠ null then
        if is_compatible(dstType, cType) then
            SCSet.add(Inst)
            continue
        end if
    end if
    if pstSet ≠ ∅ then SCSet.add(Inst)
    end if
end if
```

end for

4.6 Implementation

In this section, we describe several key implementation details. Overall, we implement T-Prunify on top of the libclang library and LLVM (v14.0.5). The implementation consists of 9,652 lines of code in total. Specifically, we implement the component "custom runtime type information inference" using libclang by analyzing the source code of the target program (as certain information like the C++ class hierarchies is preserved better at the source code level). We implement the component "static safe casting verification" using LLVM passes as LLVM is more appropriate for pointer analysis. Since our analysis spans source code and LLVM IR, we need to pass intermediate analysis results from source code level to the IR level, which we will describe in this section. In addition, we modified a state-of-the-art type confusion sanitizer, HexType [84], to facilitate the evaluation of T-Prunify.

Class Hierarchy Construction. The C++ compiler front-end like Clang can accurately parse the C++ class hierarchy. However, as the LLVM IR language is generic (i.e., needs to support different source languages) and is relatively low level, the C++ class hierarchies are not explicitly stored at the IR level. Therefore, we implement a Clang plugin to store the C++ class hierarchies (i.e., inheritance relations and type compatibility) and store them for further use.

Class Signature Database Building. This part is done by analyzing the source code of a target program directly (instead of LLVM IR). Specifically, we use libclang's python binding. Besides missing the class hierarchy information at the LLVM IR level, another reason for source code analysis is that enumeration constants at C/C++ level will be lowered to integer constants thus losing their semantic information and become indistinguishable from other integer constants. libclang allows the user to iterate the abstract syntax tree (AST) to get compilation time information (e.g., the type of the variable, the name of the variable, the type of the functions). Using libclang, we iterate through the AST to (1) extract enumeration constants used to assist identification of custom runtime type information (see section 4.4 for details), and (2) record the use of these constants (e.g., assignment, comparison, return value).

Feeding Source Code Analysis Results to the LLVM Analysis. The class signature database stores variable names, method names, and enumeration constants to assist identification of custom type checks. However, at the LLVM IR level, names of C++ virtual methods are available only in type definitions. At method call sites, virtual methods will be lowered to indirect calls, thus losing the source code level semantics. For example, a simple method call of `basic_shape->GetType()` would look like the following in LLVM IR (simplified for reading):

```
%41 = getelementptr (%"class.blink::BasicShape"*)** %40, 5
%42 = load i32 (%"class.blink::BasicShape"*)** %41
%43 = call noundef i32 %42(%"class.blink::BasicShape"* %0)
```

```
switch i32 %43, label %342 [
...
```

To overcome this issue, we modified the Clang++ front-end to annotate LLVM IR with method names. In the above example, our annotation would label `%42` as `GetType()`. The IR snippet is shown as:

```
@.str.2 = "blink::BasicShape::GetType"
%vfn = getelementptr (%"class.blink::BasicShape"*)** %vtable, 5
%10 = load i32 (%"class.blink::BasicShape"*)** %vfn
%call31 = call noundef i32 %10(%basic_shape)
%11 = call i32 @llvm.annotation(i32 %call31, (@.str.2))
switch i32 %call31, label %sw.default [
...
```

The `%vtable` is the equivalent pointer as `%40` before the annotation.

Manually-Summarized Type Checks We find that some subsystems of Chrome choose to use custom C++ templates to implement type checks. For example, we have seen `IsA<T>` frequently which operates as a type check for type `T`. Behind the template, different classes can choose to implement it differently. Since our solution to recognize type checks is by analyzing the source code, we currently recognize these statements specifically through manually-curated domain knowledge. Technically, we could pre-process the source code into a version without templates and then perform our follow-up analysis. However, due

to implementation issues, we were not able to succeed at this point. We leave this as our future work.

4.7 Evaluation

To evaluate the efficacy of T-PRUNIFY, we performed a systematic analysis of Chrome version 98.0.4720.0 with the objective of answering the following research questions:

- **RQ1:** How effective is T-PRUNIFY at identifying safe casts statically?
- **RQ2:** What is the accuracy of T-PRUNIFY?
- **RQ3:** How much runtime overhead can T-PRUNIFY improve by pruning unnecessary sanitzier checks?

We chose Chrome because it is a complex, large-scale, and well-engineered piece of software. If our analysis works on Chrome, we argue that it should also work on other targets, as long as the project leverages developer-inserted custom runtime type information (RTTI) to avoid unsafe castings. In addition, unlike smaller programs that may not have complex class hierarchies (and hence, few developer-implemented type checks), according to our analysis, the Chrome source code does indeed have many complex class hierarchies and a large number of developer-implemented type checks. Therefore, we believe Chrome represents an ideal benchmark to validate the ideas proposed in this paper. We believe that other large-scale programs such as Firefox would also similarly benefit from our solution.

Experimental Setup.. We design experiments by compiling Chrome into three versions: the original Chrome (chrome) without any instrumentation, the fully HexType-instrumented Chrome (`chrome-hextype`), and the Chrome with reduced instrumentation after applying

T-PRUNIFY (`chrome-reduced`). For `chrome-hexytpe` and `chrome-reduced`, we did not instrument libc++ as the standard C++ library because (1) we consider it as safe and (2) it does not include any custom RTTI. The performance with these three versions are compared using standard browser benchmarks. The experiment is conducted on machines equipped with Intel(R) Xeon(R) E5-2695v4 processors and 256GB RAM, running on a 64-bit Ubuntu 16.04 LTS operating system.

HexType Configurations.. The original HexType [84] is implemented based on llvm-3.9.0, which is no longer compatible with the Chromium version we test. Therefore, we ported it to llvm-14.0.5, which could be used to compile the target Chromium we evaluate. When assessing the overhead, we take into account all the optimizations implemented by HexType, which include the elimination of checks for safe casts that can be verified during compilation time.

Benchmarks.. In order to effectively showcase the performance improvements from using T-PRUNIFY, we provide further details on the benchmarks that are used in our experiments. We chose three different benchmarks, namely Speedometer, JetStream2 and Motion Mark [27] that exercise different parts of a browser. Speedometer is a benchmark that measures the responsiveness of web applications by simulating user interactions with the browser (e.g., DOM manipulation). JetStream2 is a comprehensive benchmark suite that measures the performance of JavaScript and WebAssembly in advanced web applications. It consists of a variety of tests, including latency and throughput tests, that cover a wide range of web application use cases. Finally, Motion Mark is a benchmark designed to put the graphics systems of web browsers to the test. This benchmark includes a variety of subtests, including

Table 4.1: Overall statistics of the results.

# of class hierarchies	6, 671
# of classes in hierarchies	54,617
# of class hierarchies with downcasts	1,123
# of classes as downcast targets	5,160
# of class hierarchies w/ custom RTTI found	719
# of classes w/ custom RTTI found	3,585
# of classes w/ custom RTTI & as downcast targets	827
# of downcast ops	49,364
# of downcast ops where destination types w/ RTTI	23,721
# of downcast ops with type checks (safe casts)	6,704

the CSS, image and text rendering, that assess the performance of the browser's rendering capabilities.

4.7.1 Overall Results

We first report the overall results of analyzing Chrome. As shown in Table 4.1, there are a total of 54,617 classes that are part of class hierarchies, out of which 6,671 were base classes, forming class hierarchies. We observe all downcasts occur within 1,123 of these class hierarchies and in total there are 5,160 classes that appear as downcast targets.

Among 1,123 class hierarchies, we identify 719 hierarchies, or 3,585 classes within these hierarchies, that have custom RTTI. Furthermore, we find 827 classes in total that have both RTTI and appear as downcast targets. As we can see, many classes have custom RTTI

but are never used as downcast targets. Upon inspecting some such cases, we find that their custom RTTI is used in scenarios like serialization and deserialization [1] or logging [30].

Finally, we find 49,364 downcast operations in Chrome, and 23,721 of them have destination types with custom RTTI. A subset of these downcasts, i.e., 6,704, are determined to be safe casts. In other words, T-Prunify finds these downcasts are protected by developer-inserted type checks. Overall, this represents a significant fraction of downcasts that can be exempt from sanitizer checks.

To better understand the relationship between classes and downcasts, we sort the classes by the number of downcast operations where the class appears as the destination type. Figure 4.10 illustrates the CDF of the proportion of downcast operations with respect to top n classes The blue line shows the cumulative distribution of downcast operations over top n classes, e.g., over 80% of downcast operations are concentrated in the top 1,000 classes. The green line shows the cumulative distribution of downcast operations with destination types that have custom RTTI recognized by T-Prunify, e.g., 40% of downcasts are recognized to have destination types with custom RTTI when looking at only the top 100 classes. The yellow line shows the cumulative distribution of safe casts, e.g., the majority of them are concentrated in the top 2,000 classes.

In terms of runtime overhead reduction, after eliminating the safe casts T-Prunify identified, we compare the performance overhead that T-Prunify incurs to that of HexType. Overall, our solution incurs only 25% to 74.5% of the overhead incurred by HexType, representing a significant reduction for free. We discuss more details in subsection 4.7.3.

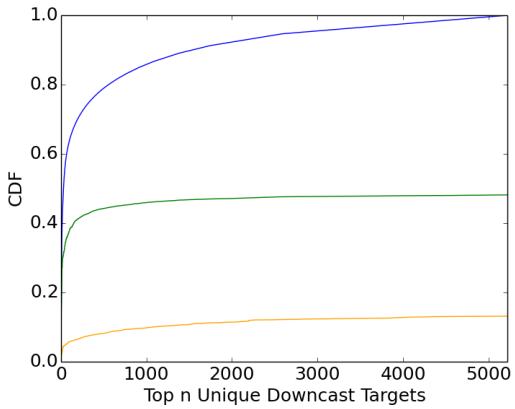

Figure 4.10: CDF of all 5,160 the downcast target classes and the downcast that we can validate.

4.7.2 Accuracy

To verify the accuracy of our approach, we check the intermediate results of each critical step. These steps include custom RTTI identification, type check identification, and safe casts identification. Then, we manually curated the ground truth of the above steps relating to the top 50 classes (that appeared as destination types of downcasts). We then use the ground truth to evaluate the false positives and false negatives of the results produced by T-Prunify.

Custom RTTI Identification. As shown in Table 4.2, we found that 25 of the top 50 classes had custom RTTI encoded, and our approaches correctly identified 20 of the 25, representing an 80% class-level coverage. We are unable to infer the custom type info encoded in 5 classes primarily due to the patterns that we currently do not recognize, as described in subsection 4.4.1. For instance, one of them is the class `v8:Uint32` which has a member function `IsUint32()` that returns `true` if the object is of class `v8::Uint32`. However, this function examines whether the object value is within the range of [0, kMaxUInt32].

Among the 20 classes that are identified to have custom RTTI by T-Prunify, we find no false positives.

Type Check Identification. In terms of false negatives, since T-Prunify failed to identify the RTTI of 5 classes, it will therefore automatically miss type checks relating to these 5 classes. To evaluate whether our safe cast identification will miss any additional cases, we sample 54 type checks from the remaining 20 classes with custom RTTI. For most classes, we sampled three checks per class. However, there are cases where we can find only one type check. The results show that T-Prunify can identify all 54 type checks. To evaluate false positives, we sample 50 type checks that are reported by T-Prunify, and all of them are true positives.

Safe Cast Identification. We follow a similar approach described above to evaluate the false negatives of safe cast identification. The same 54 sampled type checks are also in fact safe casts, according to our manual analysis. T-Prunify successfully identifies 51 to be safe casts, missing the remaining 3 because of the lacking of an inter-procedural analysis, i.e., the check is performed in a caller function but the cast happened in a callee. In addition, we sampled some cases to see whether the lack of inter-procedural analysis is the only reason. In particular, we find one false negative even when the check and cast are in the same function. The example is located in the v8 submodule: a base class `BaseSpace` which has two subclasses `NormalPageSpace` and `LargePageSpace`. Before casting an object `space` to type `NormalPageSpace`, the developer performed a type check `!(space.is_large())`. This case constitutes a safe cast. However, T-Prunify failed to identify it because it did not take into account the fact that there are only two possible types, `!is_large()` effectively

Table 4.2: The top 50 downcast classes targets in the Chrome, `# of cast` is the frequency, `Type Info?` is the ground truth whether the class has encoded some form of the custom type information. The last column shows whether T-PRUNIFY captured those information into our database.

Cast to	# of cast	Captured? Type Info?	Captured?
v8::FunctionTemplate	3117	Y	Y
v8::Object	2974	Y	Y
v8::Int32	2735	Y	N
blink::Element	1973	Y	Y
llvm::Constant	1541	N	N
blink::EventTarget	1507	N	N
v8::Uint32	1343	Y	N
llvm::Instruction	971	N	N
v8::Number	827	Y	Y
llvm::Function	806	N	N
blink::HTMLElement	716	Y	Y
blink::WebGLRenderingContextBase	476	N	N
blink::WebGL2RenderingContextBase	445	N	N
perfetto::trace_processor::TypedColumn	421	N	N
blink::LocalFrame	405	Y	Y
blink::LocalDOMWindow	401	Y	Y
blink::DOMWindow	397	N	N
v8::Boolean	395	Y	Y
blink::WebGLUniformLocation	374	N	N
v8::internal::Isolate	312	N	N
v8::Array	304	N	N
skjson::ObjectValue	294	N	N
v8::JSVisitor	286	N	N
GrGpuResource	276	N	N
blink::UniqueElementData	273	Y	Y
llvm::cl::OptionValueCopy	273	N	N
v8::String	273	Y	Y
blink::ShareableElementData	267	Y	Y
blink::JSBasedEventListener	241	Y	Y
blink::CSSPrimitiveValue	231	Y	Y
blink::SVGElement	225	Y	Y
tint::sem::Vector	213	N	N
blink::LayoutBoxModelObject	208	Y	Y
blink::LayoutBlockFlow	203	Y	N
v8::internal::compiler::HeapObjectData	196	N	N
blink::Longhand	195	Y	N
blink::LayoutBox	177	Y	N
ppapi::PPB_Graphics3D_Shared	166	N	N
blink::Node	153	N	N
content::WebContentsImpl	149	N	N
base::DictionaryValue	148	N	N
blink::Document	145	Y	Y
llvm::StructType	143	N	N
blink::HTMLInputElement	142	Y	Y
blink::NGPhysicalBoxFragment	140	Y	Y
blink::HTMLCanvasElement	135	Y	Y
blink::TransformPaintPropertyNode	134	N	N
llvm::GlobalValue	126	N	N
llvm::MDString	121	N	N
blink::JSEventHandler	118	Y	Y

Table 4.3: Overhead improvement in three benchmarks, the improvement is calculated based on the HexType instrumentation. Numbers in the parentheses is the overhead.

Benchmark	Chrome	Chrome-hextype	Chrome-reduced
Speedometer	40.3	36.2 (10.2%)	37.2 (7.7%)
JetStream2	68.6	56.3 (17.9%)	65.6 (4.4%)
MotionMark	72.3	24.8 (65.7%)	47.8 (33.92%)

indicates that `space` is of type `NormalPageSpace`. Finally, we also collect 3 patches that fix type confusion vulnerabilities in Chrome with developer-implemented type checks and they are all identified by T-Prunify, i.e., no false negatives. To evaluate false positives, we sample 50 safe casts reported by T-Prunify from six different submodules and find none of them are false positives. Overall, the results exhibit a high level of accuracy.

4.7.3 Runtime Overhead Reduction

So far, we have evaluated the results of T-Prunify statically, e.g., the effectiveness of T-Prunify in terms of the statically-identified safe casts. In this section, we will measure the runtime overhead achieved by T-Prunify compared to the state-of-the-art solution, HexType. As mentioned, we use three popular web browser benchmarks, and the corresponding results are shown in Table 4.3. We tested each benchmark four times back-to-back, and eliminate the first run as it would be a cold start. The results are relatively stable, and we use the median as the representative numbers in the table.

To compute the runtime overhead, we rely on the "scores" produced by each benchmark for chrome, chrome-hextype, and chrome-reduced, respectively Note that these

scores from different benchmarks may consist of different metrics, e.g., throughput or others. Nevertheless, we assume these scores captured the most suitable metrics as intended by each benchmark.

For Speedometer, we see that the original Chrome can achieve 40.3 runs/min, while chrome-hextype and chrome-reduced achieved 36.2 runs/min and 37.2 runs/min, respectively. Overall, the Speedometer benchmark has a relatively low overhead to being with, i.e., 10.2%, indicating that the exercised sanitizer-inserted checks have a relatively low proportion overall. Nevertheless, T-Prunify manages to reduce the overhead from 10.2% to 7.7%, representing a 25% relative reduction. In Table 4.4, we also report the total number of type checks that are executed at runtime with HexType and T-Prunify. We can see that T-Prunify successfully eliminated 100K checks that would otherwise be performed by HexType.

In JetStream2, the overall score of the original Chrome is 68.6, while chrome-hextype and chrome-reduced achieved 56.3 and 65.6 respectively. The improvement is significant, from 17.9% to 4.4%, representing a 75% relative reduction. Looking at Table 4.4, we can see the majority of the sanitizer checks are pruned, i.e., 261K out of 448K. We believe that the observed high ratio can be attributed to the predominant use of the V8 engine in the exercised code paths by JetStream2. The V8 engine has a history of being susceptible to numerous type confusion bugs [50–53, 55]. Consequently, developers have inserted a significant number of type checks as a precautionary measure to mitigate potential vulnerabilities.

In MotionMark, the overall score of the original Chrome is 72.31, while chrome-hextype and chrome-reduced achieved 24.75 and 47.78, respectively. The improvement is

Table 4.4: Number of dynamic cast verification performed by HexType before and after our pruning.

Benchmark	Chrome-hextype	Chrome-reduced	Reduced
Speedometer	7,070 K	6,891 K	179 K
JetStream2	448 K	187 K	261 K
MotionMark	4,551 K	2,163 K	2,387 K

the most significant out of the three benchmarks, i.e., from 65.7% to 33.9%, representing a 48% relative reduction. According to Table 4.4, T-Prunify results in 2.38M fewer sanitizer checks at runtime. The significant number of pruned checks in this benchmark is due to the many rendering elements such as SVG node, HTML elements with CSS style. Many such classes have custom RTTI and safe casts (as some of our examples showed in section 4.2).

4.8 Limitations and Extensions

While our experimental results are quite promising, we did encounter some practical limitations of T-Prunify while manually assessing accuracy. In the custom RTTI identification phase, we discovered cases where a wide range of constants or even structure pointers were used to assist encoding custom RTTI information, patterns not handled by our current approach. These cases could be addressed in future work by doing deeper semantic analysis during the RTTI identification phase. It is also interesting to investigate the encoding patterns of custom RTTI across open-source projects beyond Chrome. Additionally, our safe cast analysis is currently intra-procedural, which means that it may overlook certain safe casts that occur across different function calls. In future work we plan to make this

analysis inter-procedural, thereby supporting cases where a type check occurs in a caller while the cast occurs in the callee.

Beyond the aforementioned improvements, there are multiple fruitful ways to broaden and extend this work. First, we believe there are uses for automatic detection of custom RTTI schemes beyond reducing sanitizer overhead. For example, in cases where casts are not protected by a type check, one could automatically insert type checks into the code using the custom RTTI found by our technique. Also, we believe it would be worthwhile to extend our RTTI detection approach could to support `structs` in the C language—there also, custom RTTI has been used to ensure cast safety [8].

4.9 Related Works

In this section, we compare T-Prunify with closely related work in three areas.

Type Confusion Sanitizers. As mentioned before, type confusion sanitizers aim to detect bad castings that can introduce confusion vulnerabilities, by instrumenting the target program with additional runtime checks. Undefined Behavior sanitizer (UBSan) [147] is one of the earliest available type confusion sanitizers. It relies on the standard C++ RTTI to perform type compatibility check. As a result, it does not support non-polymorphic classes and may introduce crashes [97]. Recently, Clang CFI [145] also added support for detecting type confusing bugs by leveraging the standard C++ RTTI. CaVer [97] aims to address two main issues of the standard C++ RTTI: (1) it improves the speed of type checking by using unique hashes instead of mangled names, and by including all compatible types in a single RTTI entry, instead of requiring traversing the class hierarchy; (2) it supports

non-polymorphic classes by using a decoupled lookup table to find RTTI associated with a memory object. TypeSan [75] and HexType [84] further improve the performance and coverage over CaVer by using lower cost data structures, caching, and by expanding the instrumentation targets. EffectiveSan [57] can also detect type confusion vulnerabilities, besides other memory errors like out-of-bound and use-after-free. EffectiveSan uses low-fat pointer [56, 58, 95] to achieve efficient metadata access. It also supports checking casts between primitive types. T-Prunify is orthogonal and complementary to these type confusion sanitizers as our goal is to leverage developer-inserted custom type checks to eliminate redundant checks induced by type sanitizers. Therefore, T-Prunify can be combined with any of these type sanitizers.

Optimizing Sanitizer Checks. There are also some work to reduce the sanitizers' overhead, these work could be divided into two categories. The first is using sanitizer-specific static analysis to remove only semantically redundant checks, for example, RedCard [63] is designed to use static analysis to reduce the redundant instrumentation for dynamic race conditions. Similarly, DataGuard [81] uses a set of sophisticated static analyses to prove the safety of stack objects and migrate them to safe stack, thus reducing the runtime protection overhead. Furthermore, WPBound [141] utilizes value range analysis to effectively minimize the number of out-of-bound memory checks inserted by sanitizers. Besides static analysis, SIMBER [171] incorporates statistical inferences to identify redundant bound checks. Another approach develops the framework and use general heuristics to remove costly sanitizer checks irrespective their semantics, those work includes ASAP [153] and SanRazor [182]. ASAP [153] allows developers to specify the acceptable percentage of

runtime overhead based on their resource constraints. Leveraging this information, ASAP automatically instruments the program to maximize the security promise within the given budget. While SanRazor [182] combines runtime profiling and static analysis to identify and eliminate repeated and redundant checks, thereby optimizing computing resources. T-PRUNIFY falls within the first category, but with a specific focus on type confusion sanitizers.

Generic Spatial Memory Error Detector. As mentioned in section 4.2, exploiting a type confusion vulnerability usually manifests as a spatial memory error (i.e., accessing data beyond the boundary of the underlying object). Therefore, approaches that ensure spatial memory safety can also serve as mitigation against type confusion vulnerabilities. These approaches encompass static and dynamic techniques that guarantee spatial memory safety [86, 120, 121] as well as those that detect out-of-bound memory access [78, 122, 136].

Chapter 5

Conclusions

Within this thesis, we delve into the realm of memory errors that have introduced threats to computer systems. Our focus centers on the effective and efficient detection of use-before-initialization and type confusion bugs in large scale software.

We first target the principled detection of the underrated yet dangerous use-before-initialization (UBI) bugs in the Linux kernel. These bugs pose a security threat not only because they can lead to unpredictable behaviors but also because they are ex- ploitable to gain root privileges. We design and implement UbiTect, a framework that combines flow-, field-, and context-sensitive type qualifier inference with symbolic execution to identify UBI bugs with low false positive rates. A key characteristic that distinguishes UbiTect from other efforts is that it takes the best of the two meth- ods and performs scalable inter-procedural analysis to catch the uninitialized use of variables across functions. We apply UbiTect to the Linux 4.14 kernel and 138 new bugs are unearthed from which 52 of them are confirmed by Linux maintainers.

To align with the software development cycle, we design and implement InreLux, 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× 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.

Additionally, in tackling type confusion bugs, developers usually introduce custom RTTI and type checks to prevent them. Based on this observation, we implemented TPrunify, a tool that can automatically identify developer-inserted type checks and leverage these checks to validate the safety of type casts. Applying TPrunify to the Chrome browser allows us to identify a large number, i.e., 6,704, of safe casts. Leveraging this information, TPrunify can help remove redundant type casting checks induced by type confusion sanitizers like HexType and reduce the corresponding performance overhead by 25% to 75%.

Bibliography

[1] flattenable_is_valid_as_child(). https://github.com/google/skia/blob/main/src/core/SkRuntimeEffect.cpp#L371. 4.7.1

[2] How the development process works. https://www.kernel.org/doc/html/latest/process/2.Process.html. 3.5.2

[3] Infer is not deterministic. https://github.com/facebook/infer/issues/1110. 3.6.2

[4] Options to request or suppress warnings. https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html.

[5] Andersen's inclusion-based pointer analysis re-implementation in llvm. https://github.com/grievejia/andersen/graphs/contributors, 2014. 2.5

[6] CVE-2015-3636. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3636, 2015. 3.1

[7] Cve-2018-6981. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6981, 2018. 2.1

[8] Linux Commit f306dff7. https://github.com/torvalds/linux/commit/17cfe79a65f98abe535261856c5aef14f306dff7, 2018. 4.8

[9] CVE-2019-2215 - Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html, 2019. 3.1

[10] CVE-2020-25705. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25705, 2020.

[11] Cwe-476: Null pointer dereference. https://cwe.mitre.org/data/definitions/476.html, 2020. 2.1

[12] Qualifier type inference. https://github.com/seclab-ucr/UBITect/blob/master/QualifierTypeInference.pdf, 2020. 2.4.2, 2.4.2

[13] Ubitect, 2020. 2.1

[14] Clang: a C language family frontend for LLVM. https://clang.llvm.org, 2021.

[15] CVE-2021-22555: Turning
x00
x00 into 10000$. https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html, 2021. 3.1

[16] CWE-457: Use of Uninitialized Variable. https://cwe.mitre.org/data/definitions/457.html, 2021. 3.2.3

[17] GCC, the GNU Compiler Collection. https://gcc.gnu.org, 2021.

[18] HOWTO do Linux kernel development. https://www.kernel.org/doc/html/v4.14/process/howto.html, 2021.

[19] Kernel.org git repositories. git.kernel.org/pub/scm/linux/kernel, 2021. 3.1

[20] Linux. https://github.com/torvalds/linux, 2021. 3.2.1, 3.5.5

[21] 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. 3.1

[22] Smatch. https://repo.or.cz/w/smatch.git, 2021. 3.6.1

[23] Soot. https://github.com/soot-oss/soot, 2021.

[24] Xcode. https://developer.apple.com/xcode/, 2021.

[25] CWE-843: Access of Resource Using Incompatible Type ('Type Confusion'). https://cwe.mitre.org/data/definitions/843.html, 2022. 4.1

[26] IncreLux. https://github.com/seclab-ucr/IncreLux, 2022. 3.6.2

[27] Browser Benchmarks. https://browserbench.org, 2023. 4.7

[28] [compiler] fix bug in representationchanger::getword32representationfor. https://chromium.googlesource.com/v8/v8/+/fd29e246f65a7cee130e72cd10f618f3b82af232%5E%21/#F0, 2023. 4.1, 4.2.3

[29] [parser] fix eval tracking. https://chromium.googlesource.com/v8/v8/+/a4aece44c60ea1be4699667dbd27403574520df0%5E%21/#F1, 2023. 4.1, 4.2.3

[30] v8/src/objects/js-objects.cc. https://source.chromium.org/chromium/chromium/src/+/main:v8/src/objects/js-objects.cc;l=2865?q=JSObject::JSObjectShortPrint, 2023. 4.7.1

[31] [wasm] refine installation of the webassembly.exception constructor. https://chromium.googlesource.com/v8/v8/+/c0614e9bcef7266d2e4544602d668c01b5dcaa37%5E%21/#F0, 2023. 4.1, 4.2.3

[32] Iago Abal, Claus Brabrand, and Andrzej Wasowski. Effective bug finding in c programs with shape and effect abstractions. In International Conference on Verification, Model Checking, and Abstract Interpretation, pages 34–54. Springer, 2017. 3.6.1

[33] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. 2010.

[34] Steven Arzt and Eric Bodden. Reviser: efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In Proceedings of the 36th International Conference on Software Engineering, pages 288–298, 2014.

[35] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static detection of unsafe DMA accesses in device drivers. In USENIX Security Symposium, 2021. 3.1

[36] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. 2006.

[37] Marcel Böhme, Bruno CDS Oliveira, and Abhik Roychoudhury. Partition-based regression verification. In 2013 35th International Conference on Software Engineering (ICSE), pages 302–311. IEEE, 2013. 3.6.2

[38] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. 2012.

[39] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. 2008. 2.5

[40] Satish Chandra and Thomas Reps. Physical type checking for c. In Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 66–75, 1999. 4.1

[41] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley 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, pages 388–400, 2015. 3.2.2

[42] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. 2015.

[43] Andy Chou. False positives over time: A problem in deploying static analysis tools. https://www.cs.umd.edu/~pugh/BugWorkshop05/papers/34-chou.pdf. 3.6.1

[44] 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. 2.1, 2.2.1, 3.2.3

[45] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017. 3.1, 3.6.1

[46] Coverity. COVERITY SCAN STATIC ANALYSIS. https://scan.coverity.com/, 2021. 3.1, 3.6.1

[47] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. 2017.

[48] Dipok Chandra Das and Md Rayhanur 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), pages 316–321. IEEE, 2018. 3.6.3

[49] National Vulnerability Database. CVE-2015-3077. https://nvd.nist.gov/vuln/detail/CVE-2015-3077, 2015. 4.1

[50] National Vulnerability Database. CVE-2021-21224. https://nvd.nist.gov/vuln/detail/CVE-2021-21224, 2021. 4.1, 4.7.3

[51] National Vulnerability Database. CVE-2021-30561. https://nvd.nist.gov/vuln/detail/CVE-2021-30561, 2021. 4.1, 4.2.3, 4.7.3

[52] National Vulnerability Database. CVE-2022-1486. https://nvd.nist.gov/vuln/detail/CVE-2022-1486, 2022. 4.1, 4.7.3

[53] National Vulnerability Database. CVE-2022-4262. https://nvd.nist.gov/vuln/detail/CVE-2022-4262, 2022. 4.1, 4.7.3

[54] National Vulnerability Database. CVE-2023-1215. https://nvd.nist.gov/vuln/detail/CVE-2023-1215, 2023. 4.1

[55] National Vulnerability Database. CVE-2023-2033. https://nvd.nist.gov/vuln/detail/CVE-2023-2033, 2023. 4.1, 4.7.3

[56] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In Proceedings of the 25th International Conference on Compiler Construction (CC), pages 132–142, 2016. 4.9

[57] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 181–195, 2018. 4.1, 4.9

[58] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In Annual Network and Distributed System Security Symposium (NDSS), 2017. 4.9

[59] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. Detecting kernel memory leaks in specialized modules with ownership reasoning. In Network and Distributed System Security Symposium (NDSS), 2021. 3.1, 3.6.1

[60] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. volume 35, pages 57–72. ACM New York, NY, USA, 2001. 2.7

[61] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. ACM SIGOPS Operating Systems Review, 35(5):57–72, 2001.

[62] Dawson R. Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. 2007.

[63] Cormac Flanagan and Stephen N Freund. Redcard: Redundant check elimination for dynamic race detectors. In European Conference on Object-Oriented Programming, pages 255–280. Springer, 2013. 4.9

[64] Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. ACM SIGPLAN Notices, 34(5):192–203, 1999.

[65] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers, volume 37. ACM, 2002. 2.3.2, 2.4.1, 2.4.2, 2.4.2, 2.4.2, 2.7, 5

[66] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), pages 1–12, 2002. 3.6.1

[67] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. 2018. 2.7, 3.1, 3.6.1

[68] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. Software Testing, Verification and Reliability, 23(3):241–258, 2013. 3.6.2

[69] Google. Kernel addresssanitizer. https://github.com/google/kasan, 2021. 3.1

[70] Google. KMSAN. https://github.com/google/kmsan, 2021. 3.1

[71] Google. KTSAN. https://github.com/google/ktsan, 2021. 3.1

[72] Google. syzbot. https://syzkaller.appspot.com/upstream/, 2021. 3.1, 3.6.1

[73] Katerina Goseva-Popstojanova and Jacob 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), pages 344–355. IEEE, 2018. 3.6.3

[74] Shengjian Guo, Markus Kusano, and Chao Wang. Conc-ise: Incremental symbolic execution of concurrent software. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pages 531–542, 2016. 3.6.2

[75] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Typesan: Practical type confusion detection. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 517–528, 2016. 4.1, 4.2.2, 4.9

[76] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In ACM SIGSAC Conference on Computer and Communications Security (CCS), 2017. 3.1, 3.6.1

[77] Dominik Harmim, Vladimır Marcin, and Ondrej Pavela. Scalable static analysis using facebook infer. 3.1, 3.6.2

[78] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In Proceedings of USENIX Annual Technical Conference, 1992. 4.9

[79] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. 2018.

[80] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In ACM SIGSOFT Software Engineering Notes, volume 31, pages 13–19. ACM, 2005. 2.7

[81] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In Proceedings of the Network and Distributed Systems Security Symposium (NDSS), page 17, 2022. 4.9

[82] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. The taming of the stack: Isolating stack data from memory errors. In Proceedings of the Network and Distributed Systems Security Symposium, page 17, 2022.

[83] Wei Huang, Yao Dong, and Ana Milanova. Type-based taint analysis for java web applications. 2014. 2.7

[84] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2373–2387, 2017. 4.1, 4.2.2, 4.2.3, 4.6, 4.7, 4.9

[85] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In IEEE Symposium on Security and Privacy (SP). IEEE, 2019. 3.1, 3.6.1

[86] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In USENIX Annual Technical Conference, pages 275–288, 2002. 4.9

[87] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. volume 2, 2004. 2.3.2, 2.7, 3.3.2

[88] M. Jurczyk and E. Ghuliev. bochspwn-reloaded. https://github.com/googleprojectzero/bochspwn-reloaded, 2019. 3.6.1

[89] Abhijeet Kandalkar. Blink downcast helpers. https://www.slideshare.net/igalia/blink-downcast-helpers-blinkon-11, 2022.

[90] Kees Cook. Status of the Kernel Self Protection Project. In Linux Security Summit, 2016. 3.1

[91] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking# ifdef variability in c. In Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, pages 25–32, 2010. 3.6.1

[92] The kernel development community. Submitting patches: the essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/v4.10/process/submitting-patches.html, 2016.

[93] The kernel development community. How the development process works. https://www.kernel.org/doc/html/latest/process/2.Process.html, 2021. 3.5.1

[94] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In Network and Distributed System Security Symposium (NDSS), 2020. 3.1, 3.6.1

[95] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security (CCS), pages 721–732, 2013. 4.9

[96] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. 2015.

[97] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In 24th USENIX Security Symposium (USENIX Security 15), pages 81–96, 2015. 4.1, 4.2.1, 4.2.2, 4.9

[98] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, page 2201–2215, New York, NY, USA, 2017. Association for Computing Machinery. 3.1

[99] Changming Liu, Yaohui Chen, and Long Lu. Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel. In Network and Distributed System Security Symposium (NDSS), 2021. 3.1, 3.6.1

[100] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. 2018.

[101] LLVMLinux. The LLVMLinux Project. http://llvm.linuxfoundation.org/index.php/Main_Page, 2016.

[102] Kangjie Lu and Hong 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, pages 1867–1881, 2019. 3.4

[103] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. 2019. 2.6.2, 2.7

[104] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In USENIX Security Symposium, 2019. 3.1, 3.6.1

[105] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. 2016. 2.1, 2.7

[106] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 920–932, 2016. 3.2.3, 3.4, 3.6.1

[107] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In Network and Distributed System Security Symposium (NDSS), 2017. 3.2.3

[108] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. 2017. 2.1, 2.2.1, 2.6, 2.6.1

[109] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. 2017. 2.1, 2.7

[110] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In 26th {USENIX} Security Symposium ({USENIX} Security 17), pages 1007–1024, 2017. 3.1, 3.2.2, 3.3.2, 3.6.1

[111] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. Dns cache poisoning attack reloaded: Revolutions with side channels. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 1337–1350, 2020. 3.1

[112] Junjie Mao, Yu Chen, Qixue Xiao, and Yuanchun 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. 3.6.1

[113] Daniel Marjamäki. Cppcheck: a tool for static c/c++ code analysis. http://cppcheck.sourceforge.net/, 2019. 2.6.4

[114] Axel Menzel. Develop your own RTTI in C++. https://www.axelmenzel.de/articles/rtti, 2022.

[115] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. Safeinit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. 2017. 2.1, 2.7

[116] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In Network and Distributed System Security Symposium (NDSS), volume 17, pages 1–15, 2017. 3.2.3

[117] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In BlueHat IL, 2019. 2.1

[118] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In Proceedings of the 25th Symposium on Operating Systems Principles, pages 361–377, 2015. 2.7

[119] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In ACM Symposium on Operating Systems Principles (SOSP), 2015. 3.6.1

[120] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 245–258, 2009. 4.9

[121] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL), pages 128–139, 2002. 4.9

[122] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices, 42(6):89–100, 2007. 4.9

[123] Gene Novark and Emery D Berger. Dieharder: securing the heap. 2010.

[124] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In Static Analysis, pages 165–180, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 3.2.2

[125] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. Acm sigops operating systems review, 42(4):247–260, 2008. 3.6.1

[126] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In USENIX Security Symposium, 2018. 3.1, 3.6.1

[127] Aditya Pakki and Kangjie Lu. Exaggerated error handling hurts! an in-depth study and context-aware detection. In ACM SIGSAC Conference on Computer and Communications Security (CCS), 2020. 3.6.1

[128] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In USENIX Security Symposium, 2017. 3.1

[129] PaX Team. PaX - gcc plugins galore. https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf, 2013. 2.1, 2.7

[130] Poker-Edge.Com. Stats and analysis, March 2006.

[131] L.L. Pollock and M.L. Soffa. An incremental version of iterative data flow analysis. IEEE Transactions on Software Engineering, 15(12):1537–1549, 1989. 3.1

[132] David A Ramos and Dawson R Engler. Under-constrained symbolic execution: Correctness checking for real code. 2015. 2.1

[133] Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In Laurie Hendren, editor, Compiler Construction, pages 53–68, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 3.2.2

[134] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In USENIX Security Symposium, 2017. 3.1

[135] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12), pages 309–318, 2012.

[136] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In Proceedings of the 2012 USENIX conference on Annual Technical Conference, pages 28–28, 2012. 4.9

[137] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. 2001. 2.7

[138] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In Network and Distributed System Security Symposium (NDSS), 2019. 3.1, 3.6.1

[139] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. 2015. 2.1

[140] Bjarne Stroustrup. Multiple inheritance for c++. Computing Systems, 2(4):367–395, 1989. 4.2.1

[141] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. IEEE Transactions on Reliability, 65(4):1682–1699, 2016. 4.9

[142] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. IEEE Transactions on Reliability, 65(4):1682–1699, 2016.

[143] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In USENIX Security Symposium, 2021. 3.1, 3.6.1

[144] Robert Tarjan. Depth-first search and linear graph algorithms. SIAM journal on computing, 1(2):146–160, 1972.

[145] The Clang Team. Clang Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2022. 4.1, 4.9

[146] The Clang Team. HowToSetUpLLVMStyleRTTI. https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html, 2022.

[147] The Clang Team. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022. 4.1, 4.9

[148] The Clang Team. Clang static analyzer. https://clang-analyzer.llvm.org/, 2019. 2.1

[149] The Clang Team. Clang static analyzer. https://clang-analyzer.llvm.org/, 2021.

[150] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. 2017.

[151] Vegard Nossum. Getting Started With kmemcheck. https://www.kernel.org/doc/Documentation/kmemcheck.txt, 2015. 2.1

[152] Dmitry Vyukov. Syzkaller. https://github.com/google/syzkaller, 2021. 3.1, 3.6.1

[153] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In 2015 IEEE Symposium on Security and Privacy, pages 866–879. IEEE, 2015. 4.9

[154] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97, page 31–43, 1997. 3.1

[155] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. Odin: on-demand instrumentation with on-the-fly recompilation. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pages 1010–1024, 2022.

[156] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In ACM SIGSAC Conference on Computer and Communications Security (CCS. 3.1, 3.6.1

[157] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 1899–1913, 2018. 2.7

[158] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. 2012. 2.5, 2.7, 3.1, 3.4, 3.6.1

[159] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: from proof-of-concept to exploitable (one step towards automatic exploit generation). 2018.

[160] Dumidu Wijayasekara, Milos Manic, and Miles McQueen. Vulnerability identification and classification via text mining bug databases. In IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society, pages 3612–3618. IEEE, 2014. 3.6.3

[161] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In Network and Distributed System Security Symposium (NDSS), 2020. 3.6.3

[162] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20), 2020.

[163] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. Understanding and detecting disordered error handling with precise function pairing. In USENIX Security Symposium, 2021. 3.1, 3.6.1

[164] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. 2018.

[165] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. ACM Trans. Program. Lang. Syst., 29(3):16–es, May 2007. 3.2.2

[166] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In IEEE Symposium on Security and Privacy (SP). IEEE, 2020. 3.1, 3.6.1

[167] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018. 2.7, 3.1, 3.6.1

[168] Wen Xu and Yubin Fu. Own your android! yet another universal root. In 9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15), 2015.

[169] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In IEEE Symposium on Security and Privacy (Oakland), 2019. 3.6.1

[170] 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), pages 818–834. IEEE, 2019. 3.1

[171] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings 32, pages 413–426. Springer, 2017. 4.9

[172] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. 2014.

[173] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 499–510, 2013.

[174] Dacong Yan, Guoqing Xu, and Atanas 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, pages 9–14, 2012. 3.2.2

[175] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. 2015.

[176] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing {API} usages through semantic cross-checking. In 25th {USENIX} Security Symposium ({USENIX} Security 16), pages 363–378, 2016. 3.6.1

[177] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul 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, pages 221–232, 2020. 3.1, 3.2.2, 3.2.3, 3.3.3, 3.3.4, 3.3.4, 3.4, 3.5.2, 3.5.3, 3.6.1, 3.7

[178] Yizhuo Zhai, Yu Hao, Zheng Zhang, Weiteng Chen, Guoren Li, Zhiyun Qian, Chengyu Song, Manu Sridharan, Srikanth V Krishnamurthy, Trent Jaeger, et al. Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel. In 2022 Network and Distributed System Security Symposium, 2022.

[179] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. 2010. 2.7, 3.6.1

[180] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. Statically discovering high-order taint style vulnerabilities in os kernels. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 811–824, 2021. 3.6.1

[181] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In 27th USENIX Security Symposium (USENIX Security 18), pages 887–902, Baltimore, MD, August 2018. USENIX Association. 3.1

[182] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. {SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pages 479–494, 2021. 4.9

[183] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. 2019. 2.7

[184] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, August 2021. 3.1

[185] Hanqing Zhao, Yanyu Zhang, Kun Yang, and Taesoo Kim. Breaking turtles all the way down: An exploitation chain to break out of vmware esxi. 2019. 2.1

[186] Yaqin Zhou and Asankhaya 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, pages 914–919, 2017. 3.6.3

Appendix 1: Type System for UbiTect: .

We present the full qualifier inference system in this section. Our system extends the flow-sensitive analysis of Foster et al. [65]. In particular, we consider pair types (and more generally records) and present their corresponding type inference rules. Providing separate qualifiers for the elements of pairs is important in our problem domain, as records (C `structs`) are used extensively in the Linux kernel. More importantly, pointers to records are often passed between functions and whether a field of a record is or is not initialized is independent of the other fields of the record. We present a type qualifier inference system to infer a qualifier (either *init* or *uninit*) for each expression of the program.

Syntax. Our qualifier inference is performed after alias analysis. The alias analysis results are used to decorate aliased references with the same abstract locations ρ. This can be the line number of an object allocation statement. In the input programs, reference creation expressions are decorated with abstract locations and functions are decorated with effects (i.e., the set of abstract locations that they access). The abstract syntax is defined as follows:

$$
\begin{aligned}
e &:= x \mid n \mid \lambda^L x\colon t.\ e \mid e_1\ e_2 \mid \mathsf{ref}^\rho\, e \mid\ !e \\
&\mid e_1 := e_2 \mid \langle e_1, e_2\rangle \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
&\mid \mathsf{fst}(e_1) := e_2 \mid \mathsf{snd}(e_1) := e_2 \mid \\
&\mid \mathsf{assert}(e, Q) \mid \mathsf{check}(e, Q) \\
t &:= \alpha \mid \mathit{int} \mid \mathit{ref}(\rho) \mid t \to^L t' \mid \langle t_1, t_2\rangle \\
L &:= \{\rho, .., \rho\}
\end{aligned}
$$

An expression e can be a variable x, a constant integer n, a function $\lambda^L x\colon t.\ e$ with argument x of type t, effect set L and body e. The effect set L is the set of abstract locations ρ that the

function accesses. A type t is either a type variable α, an integer type int, a reference $ref(\rho)$ (to the abstract location ρ), a function type $t \rightarrow^L t'$ (that is decorated with its effects L) or a pair type $\langle t_1, t_2 \rangle$. The analysis will involve a store C that maps abstract locations ρ to types. The expression $e_1\ e_2$ is the application of function e_1 to argument e_2. The reference creation expression $\mathsf{ref}^\rho\, e$ (decorated with the abstract location ρ) allocates memory with the value e. The expression $!e$ dereferences the reference e. The expression $e_1 := e_2$ assigns the value of e_2 to the location e_1 points to. The expression $\langle e_1, e_2 \rangle$ is the pair of e_1 and e_2. The expressions $\mathsf{fst}(e)$ and $\mathsf{snd}(e)$ are the first and second elements of the pair e respectively. The expressions $\mathsf{fst}(e_1) := e_2$ and $\mathsf{snd}(e_1) := e_2$ assign the value of e_2 to the first element and second elements of the location e_1 points to respectively.

We use explicit qualifiers to both annotate and check the initialization status of expressions. The expression $\mathsf{assert}(e, Q)$ annotates the expression e with the qualifier Q. The expression $\mathsf{check}(e, Q)$ requires the top-level qualifier of e to be at most Q. We automatically insert the check expressions through a simple program transformation. Specifically, we consider two types of use as security critical: pointer dereferences and conditional branches. To detect UBI, we insert a $\mathsf{check}(e, init)$ statement before every statement where e is dereferenced or is used as the predicate of a conditional branch.

Types and Type Stores. We now define the qualified types.

$$\tau := Q\ \sigma$$

$$Q := \kappa \mid init \mid uninit$$

$$\sigma := int \mid ref(\rho) \mid (C, \tau) \to (C', \tau') \mid \langle \tau_1, \tau_2 \rangle$$

$$C := \epsilon \mid Alloc(C, \rho) \mid Assign(C, \rho \colon \tau)$$

$$\mid Merge(C, C', L) \mid Filter(C, L)$$

$$\eta := 0 \mid 1 \mid \omega$$

The qualified types τ can have qualifiers at different levels. Q can be a qualifier variable κ or a constant qualifier *init* or *uninit*. The flow-sensitive analysis associates a ground store C to each program point that is a vector that associates abstract locations to qualified types. Thus, function types are now extended to $(C, \tau) \to (C', \tau')$ where C is the store that the function is invoked in and C' is the store when the function returns.

Each location in a store C also has an associated linearity η that can take three values: 0 for unallocated locations, 1 for linear locations, and ω for non-linear locations. An abstract location is linear if the type system can prove that it corresponds to a single concrete location in every execution. An update that changes the qualifier of a location is called a strong update; otherwise, it is called a weak update. Strong updates can be applied to only linear locations. The three linearities form a lattice $0 < 1 < \omega$. Addition on linearities is as follows: $0 + x = x$, $1 + 1 = \omega$, and $\omega + x = \omega$. The type inference system tracks the linearity of locations to allow strong updates for only the linear locations.

Since a store C maps from each abstract location ρ_i to a type τ_i and a linearity η_i, we write $C(\rho)$ as the type of ρ in C and $C_{lin}(\rho)$ as the linearity of ρ in C. Store variables are denoted as ϵ. We use the following store constructors to represent the store after an

$$
\begin{aligned}
Alloc(C,\rho')(\rho) &= C(\rho) \\
Alloc(C,\rho')_{lin}(\rho) &= \begin{cases} 1 + C_{lin}(\rho) & \text{if } \rho = \rho' \\ C(\rho) & \text{otherwise} \end{cases} \\
Merge(C,C',L)(\rho) &= \begin{cases} C(\rho) & \text{if } \rho \in L \\ C(\rho') & \text{otherwise} \end{cases} \\
Merge(C,C',L)_{lin}(\rho) &= \begin{cases} C_{lin}(\rho) & \text{if } \rho \in L \\ C'_{lin}(\rho) & \text{otherwise} \end{cases} \\
Filter(C,L)(\rho) &= C(\rho) \quad \rho \in L \\
Filter(C,L)_{lin}(\rho) &= \begin{cases} C_{lin}(\rho) & \text{if } \rho \in L \\ 0 & \text{otherwise} \end{cases} \\
Assign(C,\rho' : \tau)(\rho) &= \begin{cases} \tau' \text{ where } \tau \preceq \tau' & \text{if } \rho = \rho' \wedge C_{lin}(\rho) \neq \omega \\ \tau \sqcup C(\rho) & \text{if } \rho = \rho' \wedge C_{lin}(\rho) = \omega \\ C(\rho) & \text{otherwise} \end{cases} \\
Assign(C,\rho' : \tau)_{lin}(\rho) &= C_{lin}(\rho)
\end{aligned}
$$

expression as a function of the store before it. $Alloc(C,\rho)$ returns the same store as C except for the location ρ. Allocating ρ does not affect the types in the store; however, as ρ is allocated once more, the linearity of ρ is increased by one. $Merge(C,C',L)$ returns the combination of stores C and C'; for a location ρ, if $\rho \in L$, then its type and linearity are taken from C, otherwise from C'. $Filter(C,L)$ restricts the domain of C to L. $Assign(C,\rho\colon\tau)$ overrides C by mapping ρ to a type τ' such that $\tau \preceq \tau'$. The condition $\tau \preceq \tau'$ allows assigning a subtype τ of resulting type τ' to ρ. If ρ is linear then its type in $Assign(C,\rho:\tau)$ is τ'; otherwise its type is conservatively the least-upper bound of τ and its previous type $C(\rho)$.

The type inference system generates subtyping constraints between stores. We define store subtyping in Figure .1.

$$\textsc{Int}_{\preceq}\quad \frac{Q \preceq Q'}{Q\ int \preceq Q'\ int} \qquad \textsc{Ref}_{\preceq}\quad \frac{Q \preceq Q'}{Q\ ref(\rho) \preceq Q'\ ref(\rho)}$$

$$\textsc{Fun}_{\preceq}\quad \frac{Q \preceq Q' \qquad \tau_2 \preceq \tau_1 \qquad \tau_1' \preceq \tau_2' \qquad C_2 \preceq C_1 \qquad C_1' \preceq C_2'}{Q\ (C_1, \tau_1) \rightarrow^L (C_1', \tau_1') \preceq Q'\ (C_2, \tau_2) \rightarrow^L (C_2', \tau_2')}$$

$$\textsc{Store}_{\preceq}\quad \frac{\tau_i \preceq \tau_i' \qquad \eta_i \preceq \eta_i' \qquad i = 1..n}{\{\rho_1^{\eta_1} : \tau_1, ..., \rho_n^{\eta_n} : \tau_1\} \preceq \{\rho_1^{\eta_1'} : \tau_1', ..., \rho_1^{\eta_n'} : \tau_n'\}}$$

$$\textsc{Pair}_{\preceq}\quad \frac{Q \preceq Q' \qquad \tau_1 \preceq \tau_1' \qquad \tau_1 \preceq \tau_2'}{Q\ \langle \tau_1, \tau_2 \rangle \preceq Q'\ \langle \tau_1', \tau_2' \rangle}$$

Figure .1: Store subtyping.

Constraints between stores yield constraints between linearities and types, which in turn yield constraints between qualifiers and linearities. The rule $\textsc{Int}_{\preceq}$ requires a corresponding subtyping relation for the qualifiers of the type *int*. The rule $\textsc{Ref}_{\preceq}$ requires the same subtyping relation between qualifiers and further, the equality of the two locations. The rule $\textsc{Fun}_{\preceq}$ requires the subtyping relation between the top-level qualifiers, and contra-variance for the argument and input store and co-variance for the return value and output store. The rule $\textsc{Store}_{\preceq}$ requires both subtyping and stronger linearity for corresponding locations. The rule $\textsc{Pair}_{\preceq}$ requires subtyping between the top-level qualifiers, and also subtyping for corresponding elements of the two pair type.

Type Inference System. We present the complete rules of the type inference system in Figure .2. The judgments are of the form $\Gamma, C \vdash e : \tau, C'$ that is read as in type environment

$$\textsc{Var}\ \frac{x \in dom(\Gamma)}{\Gamma, C \vdash x : \Gamma(x), C} \qquad \textsc{Int}\ \frac{\kappa \quad fresh}{\Gamma, C \vdash n : \kappa\ int, C} \qquad \textsc{Ref}\ \frac{\Gamma, C \vdash e\colon \tau, C' \qquad \tau \preceq C'(\rho)}{\Gamma, C \vdash \mathsf{ref}^{\rho} e\colon \kappa\ ref(\rho),\ Alloc(C', \rho)}$$

$$\textsc{Deref}\ \frac{\Gamma, C \vdash e : Q\ ref(\rho), C'}{\Gamma, C \vdash\ !e : C'(\rho), C'} \qquad \textsc{Assign}\ \frac{\Gamma, C \vdash e_1\colon Q\ ref(\rho), C' \qquad \Gamma, C' \vdash e_2\colon \tau, C'' \qquad \tau \preceq C''(\rho)}{\Gamma, C \vdash e_1 := e_2\colon \tau,\ Assign(C'', \rho\colon \tau)}$$

$$\textsc{Lam}\ \frac{\tau = sp(t) \qquad \epsilon, \epsilon', \kappa\ \ fresh \qquad \Gamma[x \mapsto \tau], \epsilon \vdash e : \tau', C' \qquad C' \preceq \epsilon'}{\Gamma, C \vdash \lambda^{L} x : t.e : \kappa(\epsilon, \tau) \rightarrow^{L} (\epsilon', \tau'), C}$$

$$\textsc{App}\ \frac{\Gamma, C \vdash e_1 : Q(\epsilon, \tau) \rightarrow^{L} (\epsilon', \tau'), C' \qquad \Gamma, C' \vdash e_2 : \tau_2, C'' \qquad \tau_2 \preceq \tau \qquad Filter(C'', L) \preceq \epsilon}{\Gamma, C \vdash e_1\ e_2 : \tau',\ Merge(\epsilon', C'', L)}$$

$$\textsc{Assert}\ \frac{\Gamma, C \vdash e : Q'\ \sigma, C' \qquad Q' \preceq Q}{\Gamma, C \vdash assert(e, Q) : Q\ \sigma, C'} \qquad \textsc{check}\ \frac{\Gamma, C \vdash e : Q'\ \sigma, C' \qquad Q' \preceq Q}{\Gamma, C \vdash check(e, Q) : Q'\ \sigma, C'}$$

$$\textsc{Pair}\ \frac{\Gamma, C \vdash e_1\colon \tau_1, C' \qquad \Gamma, C' \vdash e_2\colon \tau_2, C''}{\Gamma, C \vdash \langle e_1, e_2\rangle\colon \kappa\ \langle \tau_1, \tau_2\rangle, C''} \qquad \textsc{Fst}\ \frac{\Gamma, C \vdash e\colon Q\ \langle \tau_1, \tau_2\rangle, C'}{\Gamma, C \vdash \mathsf{fst}(e)\colon \tau_1, C'}$$

$$\textsc{FstAssign}\ \frac{\Gamma, C \vdash e_1\colon Q\ ref(\rho), C' \qquad \Gamma, C' \vdash e_2\colon \tau_1, C'' \qquad \kappa\ \langle \alpha_1, \alpha_2\rangle \preceq C''(\rho) \qquad \tau_1 \preceq \alpha_1 \qquad \kappa, \alpha_1, \alpha_2\ \text{fresh}}{\Gamma, C \vdash \mathsf{fst}(e_1) := e_2\colon \tau_1,\ Assign(C'', \rho\colon \langle \tau_1, \mathsf{snd}(C''(\rho))\rangle)}$$

Figure .2: Type inference system.

Γ and store C, evaluating e yields a result of type τ and a new store C'. The rules VAR and INT are standard. The rule REF creates a location and adds it to the store. The type τ of the expression e that is stored in the new location is constrained to be a subtype of the type of ρ in the post-store. The qualifier of the new location is initialized. The rule DEREF checks that the dereferenced expression is of a reference type $ref(\rho)$ and retrieves the type of the value stored at the location ρ from the store. Qualifiers are checked by the single check expression described before (and not when references are dereferenced). The rule ASSIGN checks that the left-hand side expression is of a reference type and checks that the type of the right-hand side is a subtype of the type of the value that the reference stores. It also checks that the right-hand side can be assigned to the left-hand side considering the linearity and type of the left-hand side reference and the type of the right-hand side expression (as described in the definition of *Assign* above). The rule LAM type-checks the function body e in a fresh initial store ϵ and with the parameter bound to a type with fresh qualifier variables. The resulting post-store of the function body C' should be a subtype of the post-store of the function ϵ'. This step essentially creates a function summary, which has been explained in the paper section 4.3. We use the function $sp(t)$ to decorate a standard type t with fresh qualifier and store variables:

$$
\begin{array}{rcll}
sp(\alpha) & = & \kappa\ \alpha & \kappa \text{ fresh} \\
sp(int) & = & \kappa\ int & \kappa \text{ fresh} \\
sp(ref(\rho)) & = & \kappa\ ref(\rho) & \kappa \text{ fresh} \\
sp(t \rightarrow^L t') & = & \kappa\ (\epsilon, sp(t)) \rightarrow^L (\epsilon', sp(t')) & \kappa, \epsilon, \epsilon' \text{ fresh} \\
sp(\langle t, t' \rangle) & = & \kappa\ \langle sp(t), sp(t') \rangle & \kappa \text{ fresh}
\end{array}
$$

The rule APP checks that the type of e_2 is a subtype of the parameter type of e_1, Further,

with the condition $\mathit{Filter}(C, L) \preceq \epsilon$, it checks that state of the locations that e_1 uses (captured by its effect set L) in the post-store C'' of e_2 are compatible with the store ϵ that the function e_1 expects. The resulting store $\mathit{Merge}(\epsilon', C'', L)$ joins the store C'' before the function call with the result store ϵ' of the function. Filtering and merging according to the effect set provides polymorphism as functions do not affect the locations they do not use. The rule Assert adds a qualifier annotation to the program, and the rule Check checks that the top-level qualifier Q' of e is more specific or equal to the the expected qualifier Q.

The rule Pair type-checks the expressions e_1 and e_2 in order and results in an initialized pair type. The rule Fst checks that the expression e is of a pair type and types $\mathsf{fst}(e)$ as the first element of the pair type. The qualifier Q of the pair type is unconstrained; qualifiers are only checked by the check expressions presented above. The rule FstAssign checks that the expression e_1 is of a reference type $\mathit{ref}(\rho)$, the post-store C'' (after checking e_1 and e_2) maps the reference ρ to a supertype of a pair type $\kappa\ \langle \alpha_1, \alpha_2 \rangle$, and the type τ_1 of e_2 is a subtype of α_1. The resulting store remaps ρ to a new pair type where the first element is the type of τ_1 and the second element is unchanged. More precisely, as described in the definition of *Assign* above, the *Assign* store updates ρ to the new pair type if ρ is linear; otherwise updates ρ to the least upper bound of the old and new pair types. We elide the rules for snd that are similar to the rules for fst. The constraints generated by the new rules Pair, Fst and FstAssign are type and store subtyping constraints that the previous rules generated too. Further, by the rule $\textsc{Pair}_{\preceq}$, the subtyping constraints between pair types are decomposed into subtyping constraints between qualifier and simpler types that are inductively decomposed into constraints between qualifiers and linearities. Thus, the

added inference rules do not increase the complexity of the generated constraints.

Soundness. The type inference has the following soundness property. Consider a given expression e. Consider the set of conditions $\mathcal{C}$ generated during type inference for e in the empty environment and empty store i.e., the constraints generated to derive the judgment $\emptyset, \emptyset \vdash e : \tau, C'$ for some type τ and store C'. A solution $\mathcal{S}$ for the constraints $\mathcal{C}$ is a mapping from store variables ϵ to concrete stores, from qualifier variables κ to concrete qualifiers, and from type variables α to concrete types such that $\mathcal{S}$ satisfies the constraints $\mathcal{C}$. In other words, substituting each variable in the constraints $\mathcal{C}$ with its mapping in $\mathcal{S}$ results in valid constraints. If there is a solution $\mathcal{S}$ for the constraints $\mathcal{C}$ then the evaluation of e cannot get stuck. The evaluation of an expression can get stuck if a non-reference value is dereferenced, a value is assigned to a non-reference value, a value of a mismatching type is assigned to a reference to a location of a specific type, the parameter of a function is instantiated with an argument of a mismatched type, and more importantly a qualifier check or assertion fails, i.e., the qualifier of a value is not a subtype of the expected qualifier.

Table .1: Appendix 2: Table:. Incremental analysis results for mainline versions v4.14 to v5.9. **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. **Dis(Disappearing).**: Number of warnings that disappear in the current version (compared with the last analyzed version). **Rem(Remaining).**: 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	Dis	Rem	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