UBITect: A Precise and Scalable Method to Detect Use-Before-Initialization Bugs in Linux Kernel

Yizhuo Zhai
yzhai003@ucr.edu
UC, Riverside
USA

Yu Hao
yhao016@ucr.edu
UC, Riverside
USA

Hang Zhang
hang@cs.ucr.edu
UC, Riverside
USA

Daimeng Wang
dwang030@ucr.edu
UC, Riverside
USA

Chengyu Song
csong@cs.ucr.edu
UC, Riverside
USA

Zhiyun Qian
zhiyunq@cs.ucr.edu
UC, Riverside
USA

Mohsen Lesani
lesani@cs.ucr.edu
UC, Riverside
USA

Srikanth V. Krishnamurthy
krish@cs.ucr.edu
UC, Riverside
USA

Paul Yu
paul.l.yu.civ@mail.mil
U.S. Army Research Laboratory
USA

ABSTRACT

Use-Before-Initialization (UBI) bugs in the Linux kernel have serious security impacts, such as information leakage and privilege escalation. Developers are adopting forced initialization to cope with UBI bugs, but this approach can still lead to undefined behaviors (e.g., NULL pointer dereference). As it is hard to infer correct initialization values, we believe that the best way to mitigate UBI bugs is detection and manual patching. Precise detection of UBI bugs requires path-sensitive analysis. The detector needs to track an associated variable's initialization status along all the possible program execution paths to its uses. However, such exhaustive analysis prevents the detection from scaling to the whole Linux kernel. This paper presents UbiTect, a UBI bug finding tool which combines flow-sensitive type qualifier analysis and symbolic execution to perform precise and scalable UBI bug detection. The scalable qualifier analysis guides symbolic execution to analyze variables that are likely to cause UBI bugs. UbiTect also does not require manual effort for annotations and hence, it can be directly applied to the kernel without any source code or intermediate representation (IR) change. On the Linux kernel version 4.14, UbiTect reported 190 bugs, among which 78 bugs were deemed by us as true positives and 52 were confirmed by Linux maintainers.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Systems security**.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409686

KEYWORDS

Use-Before-Initialization, bug detection, type qualifier, symbolic execution

ACM Reference Format:
Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: A Precise and Scalable Method to Detect Use-Before-Initialization Bugs in Linux Kernel. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3368089.3409686

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 [19]. UBI bugs open up significant security threats against the operating system: they could enable attackers to take control over the entire system [2, 6, 15, 32], leak sensitive information [14, 18], and can be exploited using automated means [15].

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) [25], can perform more accurate analysis, but due to path explosion, its ability to perform inter-module holistic program analysis is limited. Dynamic analysis used in MemorySanitizer [24] and kmemcheck [26] 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 [21] forces the initialization of kernel stacks during context switches between the kernel and user space. UniSan [14] forces the initialization of memory objects that may be uninitialized and may leave the kernel space (*e.g.,* `copy-to-user`). SafeInit [18] 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 [3] (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 [16] 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 [22] 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

```
1  /* file: drivers/crypto/mv_cesa.c
2   * uninteresting code lines are omitted
3   */
4  typedef void (*crypto_completion_t)(
5    struct crypto_async_request *req, int err);
6
7  struct crypto_async_request {
8    crypto_completion_t complete;
9  };
10
11 static int queue_manag(void *data)
12 {
13   /* backlog is defined without initialization */
14   struct crypto_async_request *backlog;
15   if (cpg->eng_st == ENGINE_IDLE)
16     backlog = crypto_get_backlog(&cpg->queue);
17   if (backlog)
18     /* uninitialized pointer dereferenced! */
19     backlog->complete(backlog, -EINPROGRESS);
20   return 0;
21 }
```

Figure 1: A UBI bug in the Linux kernel. 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.

52 bugs in total.Details are provided in section 6.

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 [4].
- **Results.** UbiTect found 78 bugs in the v4.14 Linux kernel, where 11 were already fixed and 37 were confirmed by Linux maintainers.

2 USE-BEFORE-INITIALIZATION BUGS

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

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) 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 [15], 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 [6]). Moreover, if a critical decision variable (*e.g.,* authenticated) is uninitialized, an attacker can bypass security checks and induce other unexpected control flows. A subsequent research effort has shown that such attacks are practical and can be constructed in an automated manner [15].

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 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.

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.

```
1  /* file: drivers/gpu/drm/vmwgfx/vmwgfx_execbuf.c
2   * uninteresting code lines are omitted
3   */
4  static int vmw_translate_mob_ptr(
5      struct vmw_private *dev_priv,
6      struct vmw_sw_context *sw_context,
7      SVGAMobId *id,
8      struct vmw_dma_buffer **vmw_bo_p)
9  {
10   struct vmw_dma_buffer *vmw_bo;// = NULL;
11   uint32_t handle = *id;
12   int ret = vmw_user_dmabuf_lookup(
13     sw_context->fp->tfile, handle, &vmw_bo);
14   if (unlikely(ret != 0))
15     return -EINVAL;
16   *vmw_bo_p = vmw_bo;
17   return 0;
18 }
```

Figure 2: An inter-procedural UBI bug in the Linux kernel. Argument vmw_bo_p may remain uninitialized during error return.

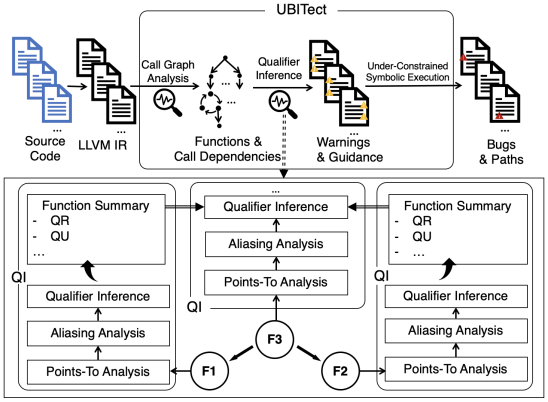


Figure 3: The workflow of UBITect, "QI":Qualifier Inference, "QR":qualifier requirements, "QU": qualifier updates

3 OVERVIEW

In this section, we show how UBITect combines type qualifier inference and symbolic execution to detect UBI bugs. Figure 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 4.2.

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

3.2 Type Qualifier Inference

Type qualifiers have been used in previous works to detect security bugs. For example, Johnson and Wagner [12] 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 [8] 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 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 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 4 as an example. At the entry of the function (line 5), `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 QR and QU for the input arguments `ns_name` and `ns_len`. Let us assume that the actual argument types are κ_1 *const char* κ_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 9) and `*ns_len` (line 10), 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 20, 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 20 (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 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.

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.

4 UBITECT DESIGN

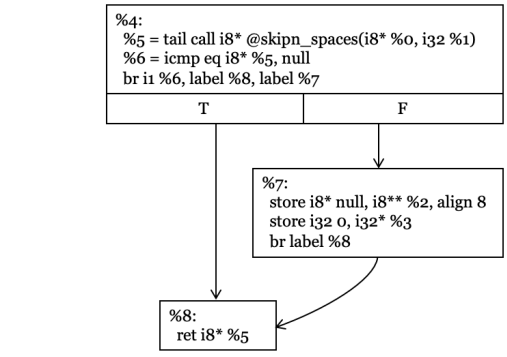
This section describes the design details of UbiTect, including points-to and aliasing analysis, the formalization of the type inference, and the symbolic execution engine.

4.1 Points-to and Aliasing Analysis

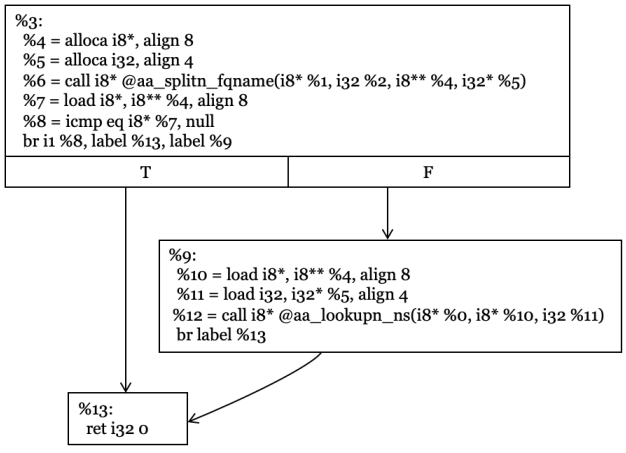
As a precursor to flow-sensitive qualifier inference [8], UbiTect performs a flow-sensitive and field-sensitive intra-procedural points-to

```c
/* file: security/apparmor/policy.c
 * uninteresting code lines are omitted
 */
const char *aa_splitn_fqname(const char *fqname, size_t n,
      const char **ns_name, size_t *ns_len) {
  const char *name = skipn_spaces(fqname, n);
  if (!name)
    return NULL; //*ns_name is not initialized
  *ns_name = NULL;
  *ns_len = 0;
  /* populate *ns_name */
  return name;
}

int aa_fqlookupn_profile(struct aa_label *base,
      const char *fqname, size_t n) {
  const char *name, *ns_name;
  size_t ns_len;
  name = aa_splitn_fqname(fqname, n, &ns_name, &ns_len);
  if (ns_name) { // UBI!
    //ns = aa_lookupn_ns(labels_ns(base), ns_name, ns_len);
  }
  return 0;
}
```



CFG for 'aa_splitn_fqname' function



CFG for 'aa_fqlookupn_profile' function

Figure 4: An inter-procedural UBI bug in the apparmor module and corresponding LLVM IR with control-flow graph (CFG).

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 (subsection 4.3).

4.2 Qualifier Inference

Our qualifier inference component is an extension of the flow-sensitive analysis by Foster *et al.* [8], 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 the qualifiers, 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 [8] for further details. We retain the standard qualifier notation from Foster *et al.* [8], and only present the type inference rules for pair expressions; the full set of inference rules is available to the interested reader in the supplementary material.

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.*

4.2.1 Syntax. Our qualifier inference is performed on LLVM IR after our alias analysis. For simplicity of discussion, we use the following abstract syntax following the one used in Foster *et al.* [8], instead of the full LLVM IR syntax.

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

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, which is calculated as part of our alias analysis. 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 $t \to^L t'$ (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 e_1 points to, respectively.

Note that, following the style of Foster *et al.* [8], 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.

4.2.2 Qualified Types and Type Stores. Given the subtyping relations, we now define the qualified types.

$$\begin{aligned}
\tau &:= Q\ \sigma \\
Q &:= \kappa \mid init \mid uninit \\
\sigma &:= int \mid ref(\rho) \mid (C, \tau) \to (C', \tau') \mid \langle \tau_1, \tau_2\rangle \\
C &:= \epsilon \mid Assign(C, \rho\colon \tau) \mid \ldots \\
\eta &:= 0 \mid 1 \mid \omega
\end{aligned}$$

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 $(C, \tau) \to (C', \tau')$

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

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

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

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

Figure 5: Store subtyping.

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.

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

$Assign(C, \rho\colon \tau)$ overrides C by mapping ρ to a type τ' such that $\tau \preceq \tau'$. (τ' can be any super-type of τ.) 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 5. 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 Pair$_{\leq}$ requires subtyping between the top-level qualifiers, and also subtyping for corresponding elements of the two pair type.

4.2.3 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) check(e, $init$) and (manually inserted) assert(e, $init$) 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 in Foster *et al.* [8], is field-sensitivity, we only present the rules for the pair expressions in this Section (Figure 6). The complete set of rules are in the supplementary material. The judgments are of the form $\Gamma, C \vdash e : \tau, C'$ 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 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 Pair$_{\leq}$, 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.

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 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.

4.3.1 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 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.

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

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

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

Figure 6: Type inference rules for the pair expressions (`C struct` fields).

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.

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 the exploration from the beginning of the function where the involved variable is allocated. Once SE enters a basic block in the avoidlist, it terminates the search along the current path. SE always choose the basic blocks in the must list when it decides which path to analyze first. In this way, type qualifier inference reduces the searching space for SE and makes it more scalable.

Since a partial path (the portion between uninitialization and use) is explored instead of a full execution path from entry to the kernel (*e.g.,* system call) to the use, some false positives could still pass the filter. Similarly, false positives caused by an imprecise call graph (*i.e.,* indirect call targets) will not be filtered. However, we ensure that no true positives are wrongly excluded.

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.

Table 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

5 IMPLEMENTATION

UbiTect is built upon the LLVM compiler infrastructure. We adopt the whole kernel analysis infrastructure from KINT [28] and modify it to match the bottom-up analysis. Points-to analysis is based on the structure analysis code from [1] while under-constrained symbolic execution stands on KLEE [5]. Overall, 13,446 LoC are added, the distributions of which are shown in Table 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 like `memset` and `memcpy` or functions implemented in pure assembly; thus, in these cases we cannot construct summaries through automatic inference.
- *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

Table 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

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, 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.

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 [15] 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.

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 [15]. Table 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.

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 the variable is actually 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 [13] 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 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):* when the path crosses multiple functions. (2) *Imprecise indirect call resolution (26 cases):* the indirect call target is infeasible. (3) *LLVM optimization (16 cases):* 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). and (5) *Assembly code (10 cases).*

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

Table 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

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, the reduction rate is actually higher than 90%.

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 [17]. 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 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 ≈ 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 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).

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.

7 RELATED WORK

Mitigating UBI Bugs. Automated mitigation of UBI bugs is pioneered by PaX's `STACKLEAK` plugins [21], 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 [18] and UniSan [14]. SafeInit [18] 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 [14] 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 [28] put together a number of static analysis techniques such as taint and range analysis to discover integer overflow vulnerabilities in the Linux kernel. Juxta [20] detects semantic bugs in Linux file systems by finding deviant behaviors in different file system implementations [7]. Dr. Checker [16] is a static taint analysis engine that can be used to find taint-style vulnerabilities in the Linux kernel. K-Miner [9] performs context-sensitive value-flow analysis to identify memory-corruption vulnerabilities. Deadline [29] and Check-it-Again [27] detect a special type of time-of-check-to-time-of-use (TOCTTOU) bugs due to lack of re-checks. CRIX [13] detects missing security checks in the Linux kernel. PeX [31] 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 [11]. Some examples of applying type systems for bug finding include finding user/kernel pointer bugs [12], format string vulnerabilities [23], integer-overflow-to-buffer-overflow [30], null pointer deference bugs [10], lock/unlock bugs and file descriptor bugs [8].

8 CONCLUSIONS

In this paper, we 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 exploitable 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 methods 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.

9 ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. We thank Manu Sridharan for his useful comments, and Weiteng Chen for his assistance. This research was partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. It was also partially supported by NSF award CNS-1718997 and ONR under grant N00014-17-1-2893.

REFERENCES

[1] 2014. Andersen's inclusion-based pointer analysis re-implementation in LLVM. https://github.com/grievejia/andersen/graphs/contributors.
[2] 2018. CVE-2018-6981. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6981.
[3] 2020. CWE-476: NULL Pointer Dereference. https://cwe.mitre.org/data/definitions/476.html.
[4] 2020. UBITect. https://github.com/seclab-ucr/UBITect
[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[6] K. Cook. 2011. Kernel Exploitation Via Uninitialized Stack. https://www.defcon.org/images/defcon-19/dc-19-presentations/Cook/DEFCON-19-Cook-Kernel-Exploitation.pdf..
[7] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5, 57–72.
[8] Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. 2002. *Flow-sensitive type qualifiers*. Vol. 37. ACM.
[9] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linu. In *Annual Network and Distributed System Security Symposium (NDSS)*.
[10] David Hovemeyer, Jaime Spacco, and William Pugh. 2005. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGSOFT Software Engineering Notes*, Vol. 31. ACM, 13–19.

[11] Wei Huang, Yao Dong, and Ana Milanova. 2014. Type-Based Taint Analysis for Java Web Applications. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
[12] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference. In *USENIX Security Symposium (Security)*, Vol. 2. https://dl.acm.org/doi/10.5555/1251375.1251384
[13] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic-and Context-Aware Criticalness and Constraints Inferences. In *USENIX Security Symposium (Security)*.
[14] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *ACM Conference on Computer and Communications Security (CCS)*.
[15] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. 2017. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *Annual Network and Distributed System Security Symposium (NDSS)*.
[16] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *USENIX Security Symposium (Security)*.
[17] Daniel Marjamäki. 2019. Cppcheck: a tool for static c/c++ code analysis. http://cppcheck.sourceforge.net/.
[18] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *Annual Network and Distributed System Security Symposium (NDSS)*.
[19] Matt Miller. 2019. Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape. In *BlueHat IL*.
[20] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 361–377.
[21] PaX Team. 2013. PaX - gcc plugins galore. https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf.
[22] David A Ramos and Dawson R Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security Symposium (Security)*.
[23] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. 2001. Detecting format string vulnerabilities with type qualifiers.. In *USENIX Security Symposium (Security)*.
[24] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *International Symposium on Code Generation and Optimization (CGO)*.
[25] The Clang Team. 2019. Clang Static Analyzer. https://clang-analyzer.llvm.org/.
[26] Vegard Nossum. 2015. Getting Started With kmemcheck. https://www.kernel.org/doc/Documentation/kmemcheck.txt.
[27] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.
[28] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[29] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and scalable detection of double-fetch bugs in OS kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*.
[30] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. 2010. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *European Symposium on Research in Computer Security (ESORICS)*.
[31] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. 2019. PeX: A Permission Check Analysis Framework for Linux Kernel. In *USENIX Security Symposium (Security)*.
[32] Hanqing Zhao, Yanyu Zhang, Kun Yang, and Taesoo Kim. 2019. Breaking Turtles All the Way Down: An Exploitation Chain to Break out of VMware ESXi. In *USENIX Workshop on Offensive Technologies (WOOT)*.

A QUALIFIER TYPE INFERENCE

We present the full qualifier inference system in this section. Our system extends the flow-sensitive analysis of Foster *et al.* [8]. 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.

A.1 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:

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

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 $\mathit{ref}(\rho)$ (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 $\mathsf{check}(e, \mathit{init})$ statement before every statement where e is dereferenced or is used as the predicate of a conditional branch.

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

A.2 Types and Type Stores

We now define the qualified types.

$$
\begin{aligned}
\tau &:= Q\ \sigma \\
Q &:= \kappa \mid \mathit{init} \mid \mathit{uninit} \\
\sigma &:= \mathit{int} \mid \mathit{ref}(\rho) \mid (C, \tau) \rightarrow (C', \tau') \mid \langle \tau_1, \tau_2\rangle \\
C &:= \epsilon \mid \mathit{Alloc}(C, \rho) \mid \mathit{Assign}(C, \rho\colon \tau) \\
&\quad \mid \mathit{Merge}(C, C', L) \mid \mathit{Filter}(C, L) \\
\eta &:= 0 \mid 1 \mid \omega
\end{aligned}
$$

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 $(C, \tau) \rightarrow (C', \tau')$ 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 $C_{\mathit{lin}}(\rho)$ 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 expression as a function of the store before it. $\mathit{Alloc}(C, \rho)$ 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. $\mathit{Merge}(C, C', L)$ 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'. $\mathit{Filter}(C, L)$ restricts the domain of C to L. $\mathit{Assign}(C, \rho\colon \tau)$ 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 $\mathit{Assign}(C, \rho : \tau)$ 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 5.

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.

A.3 Type Inference System

We present the complete rules of the type inference system in Figure 7. The judgments are of the form $\Gamma, C \vdash e : \tau, C'$ that is read as in type environment 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 autorefinter-analysis. 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 $Filter(C, L) \preceq \epsilon$, 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 $Merge(\epsilon', C'', L)$ 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 $\text{fst}(e)$ 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 $ref(\rho)$, 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.

A.4 Soundness

The type inference has the following soundness property. Consider a given expression e. Consider the set of conditions 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 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 C. In other words, substituting each variable in the constraints C with its mapping in $\mathcal{S}$ results in valid constraints. If there is a solution $\mathcal{S}$ for the constraints 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.

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

$$\textsc{Assign}\ \frac{\Gamma, C \vdash e_1 : Q\ ref(\rho), C' \qquad \Gamma, C' \vdash e_2 : \tau, C'' \qquad \tau \preceq C''(\rho)}{\Gamma, C \vdash e_1 := e_2 : \tau,\ Assign(C'', \rho : \tau)} \qquad \textsc{Lam}\ \frac{\tau = sp(t) \qquad \epsilon, \epsilon', \kappa\ fresh \qquad \Gamma[x \mapsto \tau], \epsilon \vdash e : \tau', C' \qquad C' \preceq \epsilon'}{\Gamma, C \vdash \lambda^{L} x : t.e : \kappa(\epsilon, \tau) \rightarrow^{L} (\epsilon', \tau'), C}$$

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

$$\textsc{check}\ \frac{\Gamma, C \vdash e : Q'\ \sigma, C' \qquad Q' \preceq Q}{\Gamma, C \vdash check(e, Q) : Q'\ \sigma, C'} \qquad \textsc{Pair}\ \frac{\Gamma, C \vdash e_1 : \tau_1, C' \qquad \Gamma, C' \vdash e_2 : \tau_2, C''}{\Gamma, C \vdash \langle e_1, e_2 \rangle : \kappa\ \langle \tau_1, \tau_2 \rangle, C''} \qquad \textsc{Fst}\ \frac{\Gamma, C \vdash e : Q\ \langle \tau_1, \tau_2 \rangle, C'}{\Gamma, C \vdash \mathrm{fst}(e) : \tau_1, C'}$$

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

Figure 7: Type inference system.