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

Hang Zhang
hang@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.

KEYWORDS
Use-before-Initialization, bug detection, type qualifier, symbolic execution

ACM Reference Format:

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 [20]. UBI bugs open up significant security threats against the operating system: they could enable attackers to take control over the entire system [2, 7, 16, 33], leak sensitive information [15, 19], and can be exploited using automated means [16].

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) [26], 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 [25] and kmemcheck [27] can also detect UBI bugs.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ESEC/FSE ’20, November 8–13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.1145/3368089.3409686
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 [22] forces the initialization of kernel stacks during context switches between the kernel and user space. UniSan [15] forces the initialization of memory objects that may be uninitialized and may leave the kernel space (e.g., copy-to-user). SafeInit [19] 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 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 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 [23] 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 yesconfig, 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 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 [5].
- **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 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).

![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.](image)
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 [16], 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 overwriting attacker-controlled data with uninitialized pointers (e.g., CVE-2010-2963 [7]). 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 [16].

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.

```c
/* file: drivers/gpu/drm/vmwgfx/vmgfx_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 = NULL;
    struct vmw_dma_buffer *vmw_bo = vmw_bo;
    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:** An inter-procedural UBI bug in the Linux kernel. Argument `vmw_bo_p` may remain uninitialized during error return.

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 4.2.4.

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 [13] 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 [9] to detect UBI bugs.

From a high level, we introduce two new qualifiers: init and uninit, where init ≤ uninit (i.e., init is a subtype of uninit); and defines the subtype relations between qualified types (e.g., init int ≤ 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 $\kappa$. (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 6), ns_name and ns_len are assigned with two symbolic types $k_1$ const char $k_2$ * $k_3$+ and $k_4$ size_t $k_5$+. Because ns_name ($k_2$) and ns_len ($k_3$) 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 Generation. 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 $k_1$ const char $k_2$ * init+ and $k_4$ size_t init+, where $k_1$ 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 $k_1$ 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 $k_2$ and init (i.e., $k_2 \lor$ init), as well as $k_4$ and init.

To enable context-sensitive inter-procedural analysis, we keep $k_2$ and $k_4$ 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 ($k_4$) and &ns_len ($k_5$) 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 $k_2 = uninit$, $k_2 \lor$ init = init.

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 ($k_7$) 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 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 [9], UbiTect performs a flow-sensitive and field-sensitive intra-procedural points-to
/* 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 = skpn_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;
}

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

4.2 Qualifier Inference

Our qualifier inference component is an extension of the flow-sensitive analysis by Foster et al. [9], 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 [9] for further details. We retain the standard qualifier notation from Foster et al. [9], 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 [4].

The subtyping relation between the two qualifiers is straightforward: \( \text{init} \leq \text{uninit} \) (i.e., init is a subtype of uninit), meaning that a variable of \( \text{init} \) type could be valid wherever uninit type 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 \( \text{int} \), its subtyping relation of qualified \( \text{int} \) is:

\[
\frac{Q \leq Q'}{\text{int} \leq Q'}
\]

This means that if qualifier \( Q \leq Q' \), then \( \text{int} \) is a subtype of \( Q' \) int. For instance, \( \text{init} \) int is a subtype of \( \text{uninit} \) int. When it comes to references, the rule is more complicated. The following rule defines the subtyping relation between qualified references.
Specifically, it requires that the type of the \( t \) to which the references point, be the same.

4.2.1 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. [9], instead of the full LLVM IR syntax.

An expression \( e \) can be a variable \( x \), a constant integer \( n \), a function \( \lambda \ t \ x . e \) with argument \( t \) of type \( \tau \), effect set \( L \) and body \( e \). The effect set, \( L \), is the set of abstract locations \( \rho \) that the function accesses, which is calculated as part of our alias analysis. A type \( \tau \) is either a type variable \( \xi \), an integer type \( \int \), a reference \( \rho \) (to the abstract location \( \rho \)), a function type \( t \rightarrow \tau \) (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 \( \rho \ e \) (decorated with the abstract location \( \rho \)) allocates memory to store the value \( e \). The expression \( \text{ref} \ e \) 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 \( \text{fst}(e) \) and \( \text{snd}(e) \) are the first and second elements of the pair \( e \), respectively. The expressions \( \text{fst}(e_1) := e_2 \) and \( \text{snd}(e_1) := e_2 \) assign the value of \( e_2 \) to the first and second elements of the location \( e_1 \) points to, respectively.

Note that, following the style of Foster et al. [9], we use explicit qualifiers to both annotate and check the initialization status of expressions. The expression \( \text{assert}(e, \rho) \) annotates the expression \( e \) with the qualifier \( \rho \), which is used to manually annotate types (e.g., the \text{from} argument of \text{copy_to_user}). The expression \( \text{check}(e, \rho) \) requires the top-level qualifier of \( e \) to be at most \( \rho \). We automatically insert the \( \text{check}(e, \text{init}) \) expressions by a simple program transformation before every security-critical use to enforce the safety of the operations. Specifically, we consider a pointer dereference \( (\text{ref} \ e) \) 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.

\[
\begin{align*}
\tau & := \emptyset \sigma \\
\emptyset & := Q \sigma \\
\sigma & := \kappa \mid \text{init} \mid \text{uninit} \\
\kappa & := \int \mid \text{ref}(\rho) \mid (C, \tau) \rightarrow (C', \tau') \mid \langle t_1, t_2 \rangle \\
C & := e \mid \text{Assign}(C, \rho : \tau) \mid \ldots \\
\eta & := 0 \mid 1 \mid \omega
\end{align*}
\]

The qualified types \( \tau \) can have qualifiers at different levels. \( Q \) can be a qualifier variable \( \kappa \) or a constant qualifier \text{init} \ or \text{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 \((C, \tau) \rightarrow (C', \tau')\).

\[
\begin{align*}
\text{INT}_Q & \quad \frac{Q \leq Q'}{Q \text{ ref}(\tau) \leq Q' \text{ ref}(\tau)} \\
\text{FUN}_Q & \quad \begin{array}{c}
\frac{Q \text{ int} \leq Q' \text{ int}}{\text{ref}(\rho) \leq Q' \text{ ref}(\rho)} \\
\frac{Q \leq Q'}{\langle C_1, \tau_1 \rangle \rightarrow^L \langle C_2, \tau_2 \rangle \leq \langle C_3, \tau_3 \rangle \rightarrow^L \langle C_4, \tau_4 \rangle} \\
\frac{\text{ref}(\rho) \leq \text{ref}(\rho')}{} \\
\frac{\tau_1 \leq \tau_1'}{\eta_1 \leq \eta_1'} \\
i = 1..n
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{PAIR}_Q & \quad \begin{array}{c}
\langle \rho^1_1 : \tau_1, ..., \rho^m_1 : \tau_1 \rangle \leq \langle \rho^1_2 : \tau_2, ..., \rho^m_2 : \tau_2 \rangle \\
\tau_1 \leq \tau_1' \\
\tau_2 \leq \tau_2'
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{STORE}_Q & \quad \begin{array}{c}
\langle C_1, \tau_1 \rangle \leq \langle C_2, \tau_2 \rangle \\
\tau_1 \leq \tau_1' \\
\tau_2 \leq \tau_2'
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{ASSIGN}_Q & \quad \text{Assign}(C, \rho : \tau)(p) = \\
\begin{cases}
\tau' \text{ where } \tau \leq \tau' & \text{if } \rho = \rho' \land C_{\text{lin}}(\rho) \neq \omega \\
\tau' \lor C(\rho) & \text{if } \rho = \rho' \land C_{\text{lin}}(\rho) = \omega \\
C(\rho) & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{ASSIGN}_Q & \quad \text{Assign}(C, \rho : \tau)(\text{lin})(p) = C_{\text{lin}}(\rho)
\end{align*}
\]

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 \( \eta \) 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 \( \omega \) 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 \( \rho_j \) to a type \( \tau_j \) and a linearity \( \eta_j \), we write \( C(\rho) \) as the type of \( \rho \) in \( C \) and \( C_{\text{lin}}(\rho) \) as the linearity of \( \rho \) in \( C \). Store variables are denoted as \( \omega \). We use the following store constructor to represent the store after an assignment expression as a function of the store before it.

Assigning \( C, \rho : \tau \) overrides \( C \) by mapping \( \rho \) to a type \( \tau' \) such that \( \tau \leq \tau' \). \( \tau' \) can be any super-type of \( \tau \). The condition \( \tau \leq \tau' \) allows assigning a subtype \( \tau \) of resulting type \( \tau' \) to \( \rho \). If \( \rho \) is linear then the type \( \text{Assign}(C, \rho : \tau) \) is \( \tau' \); otherwise its type is conservatively the least-upper bound of \( \tau \) 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 \( \text{INT}_Q \) requires a corresponding subtyping relation for the qualifiers of the type \( \text{int} \). The rule \( \text{REF}_Q \) requires the same subtyping relation between qualifiers and further, the equality of the two locations. The rule \( \text{FUN}_Q \) requires the subtyping relation between the top-level qualifiers, and contra-variability for the argument and input store and co-variability for the return value and output store. The rule \( \text{STORE}_Q \) requires both subtyping and stronger linearity.
4.2.3 Type Inference System. A type inference system consists of 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. [9], 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 [4]. The judgments are of the form $\Gamma, C \vdash e : \tau, C'$ that is read as: in the type environment $\Gamma$ and store $C$, evaluating $e$ yields a result of type $\tau$ 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 $\text{ref}(\rho)$, the post-store $C'$ (after checking $e_1$ and $e_2$) maps the reference $\rho$ to a supertype of a pair type $\tau (\alpha_1, \alpha_2)$, and the type $\tau_1$ of $e_2$ is a subtype of $\alpha_1$. The resulting store remaps $\rho$ to a new pair type where the first element is the type of $\tau_1$ 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, subtyping constraints between pair types are decomposed into subtyping constraints between qualifier and simpler types that are inductively decomposed into constraints between qualifiers and lineairities. 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 $\rho$ to their types $\tau$ before the function is called, $\tau$ is the parameter type, $C'$ maps locations $\rho$ to (possibly) updated types $\tau$ after the function is called, $\tau'$ is the return type, and $L$ is the set of locations accessed by the function. The concept is further exemplified by the following example:

$$\text{init} ([\rho \mapsto \text{uninit int}, \rho' \mapsto \text{init int}], \text{ref}(\rho))$$

$$(\rho, \rho')$$

It represents an (initialized) function that starts with a pre-store where $\rho$ is uninitialized and $\rho'$ is initialized. The input is the reference for $\rho$, and the function accesses both $\rho$ and $\rho'$. The function initializes $\rho$ and leaves $\rho'$ initialized. This function is summarized as follows – no initialization requirements for its parameter and one update: update parameter $\rho$ 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 treated similarly: if it depends on the qualifier of the input value(s), UbiTect keeps it 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 callee 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 believe are important.
whose summaries are not available, the subtyping constraints are
This process is repeated until there are no changes to the summaries.

false positives caused by an imprecise call

kernel (use) is explored instead of a full execution path from entry to the
beginning of the function where the involved variable is allocated.

UbiTect

UbiTect

UbiTect

Up to this point, the type qualifier inference reported all the sus-
picious UBI locations. Next, UniTect moves on to analyze its callers using this imprecise sum-
mary. Following the dependency circle, the function is analyzed again. Because this time the summaries of its callees will be avail-
able, despite being imprecise, a new summary would be generated. This process is repeated until there are no changes to the summaries.

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 UniTect 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 sus-
picious UBI locations. Next, UniTect uses under-constrained sym-
bofic 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.

Since a partial path (the portion between uninitialized 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

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 addrs; 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 UniTect 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.

Table 1: LoC for different analysis of UniTect.

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

UniTect is built upon the LLVM compiler infrastructure. We adopt the whole kernel analysis infrastructure from KINT [29] 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 [6]. Overall, 13,446 LoC are added, the distributions of which are shown in Table 1.

We manually summarize 26 functions from three major cate-
gories. (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 manu-
ally summarize typical kernel heap allocation functions, in-
cluding 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.

<table>
<thead>
<tr>
<th>Commit or CVE No</th>
<th>Type</th>
<th>UbiTect</th>
</tr>
</thead>
<tbody>
<tr>
<td>bde6f9d</td>
<td>intra-procedural</td>
<td>Yes</td>
</tr>
<tr>
<td>1a922b9</td>
<td>intra-procedural</td>
<td>Yes</td>
</tr>
<tr>
<td>8134233</td>
<td>inter-procedural</td>
<td>Yes</td>
</tr>
<tr>
<td>c94a3d4</td>
<td>inter-procedural</td>
<td>Yes</td>
</tr>
<tr>
<td>da5e7ff</td>
<td>inter-procedural</td>
<td>Yes</td>
</tr>
<tr>
<td>CVE 2010-2963</td>
<td>inter-procedural</td>
<td>Yes</td>
</tr>
<tr>
<td>7814657</td>
<td>inter-procedural</td>
<td>Yes</td>
</tr>
<tr>
<td>6f44b15</td>
<td>inter-procedural</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 [16] 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 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 it is returned by SE is infeasible, or the variable is actually initialized along the path.

Experimental Setup. 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 7 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 [14] 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 happen. The last column: A-Patch Applied; C-Confirmed by developers

<table>
<thead>
<tr>
<th>No.</th>
<th>Sub-System</th>
<th>Module</th>
<th>Variable</th>
<th>Line No.</th>
<th>Patch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>iommu/amd</td>
<td>iommu.c</td>
<td>unmap_size</td>
<td>1525</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>assoc/rf55</td>
<td>rt5613.c</td>
<td>ref</td>
<td>1759</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>assoc/rf274</td>
<td>rt274.c</td>
<td>buf</td>
<td>364</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>assoc/rf275</td>
<td>rt275.c</td>
<td>val</td>
<td>1133</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>net/stmacs</td>
<td>stmacs.c</td>
<td>val</td>
<td>646</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>clk/gemini</td>
<td>clk-gemini.c</td>
<td>val</td>
<td>320</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>i/o/adc</td>
<td>max9365c0adc.c</td>
<td>regval</td>
<td>286</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>i/o/adc</td>
<td>max9365c0adc.c</td>
<td>regval</td>
<td>313</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>i/o/adc</td>
<td>max9365c0adc.c</td>
<td>val</td>
<td>454</td>
<td>C</td>
</tr>
<tr>
<td>10</td>
<td>i/o/adc</td>
<td>max9365c0adc.c</td>
<td>regval</td>
<td>631</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>i/o/adc</td>
<td>max9365c0adc.c</td>
<td>val</td>
<td>789</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>regulator</td>
<td>pfuize100-regulator.c</td>
<td>val</td>
<td>635</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>dem/bridge</td>
<td>sti902c.c</td>
<td>status</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>ccer</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>crl</td>
<td>140</td>
<td>C</td>
</tr>
<tr>
<td>16</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>cce</td>
<td>168</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>crl</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>crl</td>
<td>222</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>psi</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>arr</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>dat</td>
<td>411</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>i/o/trigger</td>
<td>stm32-timer-trigger.c</td>
<td>dat</td>
<td>454</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>media</td>
<td>atmel-isoc.c</td>
<td>isc_interr</td>
<td>1255</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>media</td>
<td>atmel-isoc.c</td>
<td>isc_intmask</td>
<td>1255</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>mfd</td>
<td>fol-uniq21-tscd.c</td>
<td>status</td>
<td>40</td>
<td>C</td>
</tr>
<tr>
<td>26</td>
<td>mfd</td>
<td>ti_am35x53xscd.c</td>
<td>reg</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>net/ethernet</td>
<td>has_indio.c</td>
<td>reg_val</td>
<td>165</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>clk/axi-clkgen</td>
<td>clk-axi-clkgen.c</td>
<td>d</td>
<td>314</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>power/supply</td>
<td>max17042_battery.c</td>
<td>read_val</td>
<td>485</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>power/supply</td>
<td>max17042_battery.c</td>
<td>vffsoc</td>
<td>667</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>power/supply</td>
<td>max17042_battery.c</td>
<td>vffsoc</td>
<td>682</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>pwm</td>
<td>pwm-stm32-lp.c</td>
<td>val</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>pwm</td>
<td>pwm-stm32-lp.c</td>
<td>psi</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>power/supply</td>
<td>max17042_battery.c</td>
<td>full_cap0</td>
<td>681</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>power/supply</td>
<td>max17042_battery.c</td>
<td>val</td>
<td>1082</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>power/supply</td>
<td>rt5033_battery.c</td>
<td>val</td>
<td>31</td>
<td>C</td>
</tr>
<tr>
<td>37</td>
<td>i/o/adc</td>
<td>bcmipro_adc.c</td>
<td>intstat</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>i/o/adc</td>
<td>bcmipro_adc.c</td>
<td>intr_mask</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>i/o/adc</td>
<td>bcmipro_adc.c</td>
<td>intstat</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>i/o/adc</td>
<td>bcmipro_adc.c</td>
<td>ch_instat</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>i/o/adc</td>
<td>bcmipro_adc.c</td>
<td>channel_status</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>i/o/adc</td>
<td>bcmipro_adc.c</td>
<td>val_chk</td>
<td>299</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>pwm</td>
<td>pwm-stm32.c</td>
<td>psc</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>pwm</td>
<td>pwm-stm32.c</td>
<td>arr</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>pwm</td>
<td>pwm-stm32.c</td>
<td>cce</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>pwm</td>
<td>pwm-stm32.c</td>
<td>cce</td>
<td>312</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>regulator</td>
<td>ltc_power100.c</td>
<td>qptf_flags</td>
<td>419</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>regulator</td>
<td>max90770-regulator.c</td>
<td>val</td>
<td>303</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>media</td>
<td>pvrusl42-hdw.c</td>
<td>qptf_flags</td>
<td>793</td>
<td>A</td>
</tr>
<tr>
<td>50</td>
<td>x86/hpet</td>
<td>hpet.c</td>
<td>msg.EZ</td>
<td>503</td>
<td>C</td>
</tr>
<tr>
<td>51</td>
<td>staging/ddk750</td>
<td>ddk7170_chip.c</td>
<td>dat</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>power/supply</td>
<td>max17042_battery.c</td>
<td>val</td>
<td>837</td>
<td>C</td>
</tr>
</tbody>
</table>

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 cpptcheck [18]. Both UbiTect and cpptcheck 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, cpptcheck’s analysis is only intra-procedural and reports the warning when the uninitialized variable is read. We ran the cpptcheck 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 cpptcheck. The remaining 131 false positives were correctly filtered by UbiTect’s inter-procedural static analysis.

Opposite to the cpptcheck’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 packing. 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 [22], 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 Safelnit [19] and UniSan [15]. Safelnit [19] 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 [15] 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 [29] put together a number of static analysis techniques such as taint and range analysis to discover integer overflow vulnerabilities in the Linux kernel. Juxta [21] detects semantic bugs in Linux file systems by finding deviant behaviors in different file system implementations [8]. Dr. Checker [17] is a static taint analysis engine that can be used to find taint-style vulnerabilities in the Linux kernel. K-Miner [10] performs context-sensitive value-flow analysis to identify memory-corruption vulnerabilities. Deadline [30] and Check-it-Again [28] detect a special type of time-of-check-to-time-of-use (TOCTTOU) bugs due to lack of re-checks. CRIX [14] detects missing security checks in the Linux kernel. PeX [32] 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 [12]. Some examples of applying type systems for bug finding include finding user/kernel pointer bugs [13], format string vulnerabilities [24], integer-overflow-to-buffer-overflow [31], null pointer dereference bugs [11], lock/unlock bugs and file descriptor bugs [9].

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.

ACKNOWLEDGMENTS

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


