Gmail **Mohsen Lesani <mohsen.lesani@gmail.com>**

[ESEC/FSE 2020] Congratulations! Accepted submission #172 "UBITect: A Precise and Scalable Method..."

1 message

ESEC/FSE 2020 HotCRP <noreply@esecfse2020.hotcrp.com> Tue, May 19, 2020 at 10:10 AM

Reply-To: esecfse2020-chairs@googlegroups.com
To: Mohsen Lesani <lesani@cs.ucr.edu>

Dear Mohsen Lesani,

Congratulations! We are delighted to inform you that your submission #172 has been accepted to appear in the conference.

Title: UBITect: A Precise and Scalable Method to Detect
Use-Before-Initialization bugs in Linux Kernel
Authors: Yizhuo Zhai (University of California, Riverside)
Yu Hao (University of California, Riverside)
Hang Zhang (University of California, Riverside)
Daimeng Wang (University of California, Riverside)
Chengyu Song (University of California, Riverside)
Zhiyun Qian (University of California, Riverside)
Mohsen Lesani (University of California, Riverside)
Srikanth Krishnamurthy (University of California, Riverside)
Paul Yu (U.S. Army Research Laboratory)
Site: https://esecfse2020.hotcrp.com/paper/172

After a rigorous review process, 101 of 360 submissions were accepted, some of them conditionally (acceptance rate: 28%). Congratulations!

To earn badges for your paper, please submit to the Artifacts track, the submission date is soon: *Thursday, June 4th.* Note that artifact badges can be earned for this paper and also other papers where the main results were subsequently replicated and/or reproduced. For more information: https://2020.esec-fse.org/track/esecfse-2020-artifacts

Please read the following information carefully.

1. Based on the current situation in Central California, it’s difficult to know whether there will be a physical conference and it is highly likely that the conference might be held entirely virtually. *Please do not book any travel until further notice.* We will make an announcement by mid June on the format of the conference.

Whether the conference is physical or virtual, we expect to continue the previous tradition of requiring at least one registered author. But we expect that the virtual option costs will be much lower, similar to that of other similar recent virtualized conferences

2. The reviews and comments are included below. Please make sure to address reviewers comments when you prepare the final version of your paper.

3. Each paper will have one additional page that can be used for addressing the reviewers' comments. The final page limit is: *11 pages + 2 pages for references*.

4. The camera ready paper due date is Thursday, September 20th. You will receive a separate email with the author-kit in late June. Your paper id for the publication process and registration system will be fse20main-p172-p

We encourage you to also consider submissions to the other tracks (industry track, tool demos, doctoral symposium, etc.) and workshops of ESEC/FSE. Please check the web-page for the submission dates.
https://2020.esec-fse.org/

Congratulations again! We look forward to your paper at the conference.

Myra Cohen and Thomas Zimmermann <esecfse2020-chairs@googlegroups.com>
ESEC/FSE 2020 Program Co-Chairs

Review #172A
===

Overall merit
----
3. Weak accept

Paper summary

The paper presents UBITech, an inter-procedural analysis for use-before-initialization detection in the Linux kernel. UBI first leverages a flow-sensitive type qualifier analysis to find potential UBI bugs, and then uses under-constrained symbolic execution to check if a feasible path exists for the UBI bug to expose.

The evaluation of UBITect shows its effectiveness in detecting more UBI bugs than existing static analyzers, with a moderate false positive rate. The tool reported 190 bugs in 4.14 Linux kernel with 52 confirmed by the Linux maintainers.

Strengths and weaknesses

Strengths:
UBITech aims to detect UBI bugs, which are known to be severe, hard to detect and can lead to security vulnerabilities in Linux kernel.
Combining qualifier inference and symbolic execution achieves a good trade-off between precision and scalability for the Linux kernel.
Evaluation results are strong. 78 bugs were reported as true bugs. The false positive rate is acceptable.

Weaknesses:

UBITech is useful for finding UBI bugs but does not have any theoretical guarantee. It has both false positives and false negatives. Clearly UBITech is not precise as claimed in the paper's title. Around half of the detected bugs are false positives.

While I understand UBITech is more scalable than whole-program symbolic execution techniques such as CSA, the evaluation section lacks evaluating scalability of UBITech. The paper could be improved if the authors listed the time each component of UBITech takes.

Comments for the authors

The paper is well written and easy to follow. The workflow of UBITech looks sound to me. The key technical contribution is a way to combine type qualifier inference with under-constrained symbolic execution to hit a sweet spot in the trade-off between precision and scalability.

I have a few questions on the paper's evaluation results, as reported in Section 6.2.

Is a week the total time spent on qualifier inference and symbolic execution or just the qualifier inference?

147643 potential use of uninitialized stack variables were reported by the qualifier inference, but the paper says only 4150 of them were filtered as false positives by the symbolic execution. If I understand this correctly, this leaves over 140k warnings unfiltered, which is a very huge number and may contain many false positives. It is strange that the authors offered no explanation on how many of these 140k warnings are false positives.

It is not clear to me how the 190 bugs are chosen for manual inspection among the 140k warnings. Are all the potential warnings analyzed by symbolic execution and only 190 of them finish within 2 minutes?

The false positive rate is calculated from the 190 bugs manually chosen rather than all potential warnings deemed feasible by the symbolic execution, which seems to me is not an objective measurement.

Section 6.2 says manual checks over the callers of `regmap_read()` lead to additional 60 bugs. Why are they missed (not reported) by UBITech?

How many of the 52 confirmed bugs in Table 3 were found without manual inspection of the callers of `regmap_read()`?

Do you have any insights on the difficulty of patching these 78 true positives?

* *

Review #172B
===

Overall merit
----
3. Weak accept

Paper summary

This paper presents a static analysis technique to find Use-Before-Initialization (UBI) vulnerabilities, that is, uses of uninitialized variables that attackers can exploit to point to arbitrary memory locations to execute malicious code.
The approach combines inter-procedural flow-sensitive analysis, path-sensitive analysis, type qualifier

inference, and under-constrained symbolic execution.
The paper report the results of an experimental evaluation on the Linux Kernel, and indicates the ability to reveal previously unknown UBI vulnerabilities, many of which have been confirmed by Linux kernel developers.

Strengths and weaknesses

Strengths

+ The approach addresses a relevant set of dangerous vulnerabilities
+ The paper well motivated the need of inter-procedural and path sensitive analysis
+ The approach is well-founded, and the basis of static analysis and symbolic execution seem sound
+ The proposed technique detects true positives in the Linux Kernel confirmed by developers
+ The approach could detect potential issues that could not be detected by other
static analysis tools

Weaknesses

- Some aspects of the technique are unclear and seems contradicting, in particular the strategy used for detecting new bugs (Section 6.2) (see detailed comments)

- The novel aspects of the approach are not clearly explained with respect to the state of the art
- In the evaluation section, the meaning of false positive is not clear
- The paper does not evaluate the conservative rules for heap objects.

Comments for the authors

The paper tackles the important class of use-before-initialization vulnerability, and the proposed combination of static analysis and symbolic execution can detect relevant vulnerabilities as indicated by the experimental results.

The presentation is not always precise, especially in the experimental details.
The static analysis generates 147,643 warnings, Symbolic execution identifies 4,150
as false positives, cannot handle 1,190 cases and finds 190 issues with a time budget of 2 minutes per case.
The paper indicates that several warnings correspond to the same variable, but different usages, however, it does not clearly discuss how this is determined and handled.
At page 9, the authors indicate that 20 of the 78 identified true positives have been fixed by Linux developers and “further (manual) checks over the rest callers led to additional 60 bugs." (Line 938)
Later in the paper, the authors indicate 118 reported bugs to Linux developers, 52
confirmed, and 35 disregarded as "will not happen in reality".
It would be interesting to know which of the 52 confirmed have been found automatically
and which have been found manually.

The comparison with the Clang Static Analyzer (CSA) is not entirely fair. The authors check whether CSA can detect the true positives detected with the approach proposed in this paper, by running CSA on the 78 files where the approach proposed in this paper found the issues, and notice that CSA cannot detect all 78 issues, but it could detected 7 new
issues.
The authors justify their decision to restrict the CSA experiments to 78 of the 16,163 files of the whole Linux kernel, by noticing that analyzing all files with CSE requires 13 days, while analyzing 78 files requires 1.5 hours.
However, the execute their approach for about a week (Line 910).
It would be reasonable to provide CSA with a similar time budget.

The process for manually identifying false positives is not clear: the paper does not clarify if false positives are about feasibility of the paths or about maliciousness of the uninitialized uses.

Under-constrained symbolic execution checks individual functions (intra-procedural) rather than the whole program (Line 269). This makes Section 2.1 confusing. First, it argues for the importance of inter-procedural analysis, and then the under-constrained symbolic execution is intra-procedural.
Under-constrained symbolic execution is indeed path-sensitive (symbolic execution is by definition path sensitive). Under-constrained symbolic execution remains path sensitive with respect to each individual function.

Typos:
117 a soundy -> a sound
643: conflict -> conflicting
791: generated.This -> generated. This
889: other open sourced static analyzer -> other open sourced static analyzers
609 C(\rho) \neq \omega -> C_lin(\rho) \neq \omega (same in Line 610)
780: After building call graph -> After building a call graph
889: how does other open sourced static analyzer perform for finding UBI bugs in the Linux kernel? -> how do other open source static analyzers perform for finding UBI bugs in the Linux kernel?
926: are due to use of -> are due to the use of
945: back -> black
1068: Threats of Validity -> Threats to Validity

Questions for the authors’ response
----
1) What is your intended interpretation of false positive? (infeasible path, benign UBI, or both?)
2) How many of the confirmed bugs were detected with UbiTect and how many were
detected manually?
3) Could you give example of bugs that are considered to not happen in reality?
4) Why do you consider only pairs instead of tuples of arbitrary size?

* *

Review #172C
===

Overall merit

4. Accept

Paper summary

This paper proposes a precise and scalable approach, UbiTect, to detect use-before-initialization (UBI) bugs in the Linux kernel. First, UbiTect utilizes flow-sensitive type qualifier inference to detect UBI bugs in a bottom-up and inter-procedural way. Second, UbiTect adopts guided symbolic execution to prune false positives of detected UBI bugs. The combination of these two techniques makes UbiTect precise and scalable. The experiments on the Linux kernel shows that UbiTect can detect real UBI bugs effectively. Specially, 37 UBI bugs detected by UibTect have been confirmed by Linux maintainers.

Strengths and weaknesses

+ This paper targets at an important and practical problem.
+ The approach seems sound, and capable of detecting UBI bugs.
+ The paper is well written, and easy to read.
+ UbiTect is applied in the Linux kernel, and real UBI bugs are detected and confirmed by Linux maintainers.
- Guided symbolic execution does not seem very effective in pruning false positives.
- Some experimental details are not explained clearly.

Comments for the authors

===Soundness===
* The motivation is well discussed in the paper. First, UBI bugs in the Linux kernel can cause serious security impacts, e.g., arbitrary code execution. Second, existing works suffer from precision and scalability issues for the Linux kernel analyses. Third, the challenges are well discussed in Section 2.2. Although I’m not familiar with the analyses on the Linux kernel, I can fully understand the motivation.

* The description about the proposed approach is quite detailed, and easy to follow. I like the example in Figure 4, which makes the whole approach described in Section 3 easy to understand. UbiTect adapts type qualifier inference proposed by Johnson and Wagner, to detect UBI bugs, and then adopts guided symbolic execution to validate reported UBI bugs in the static analysis. The proposed approaches are sound and reasonable to me.

* UBI uses two assertions to detect UBI bugs (Line 303). Are these two assertions representative? Are there some other assertions? For example, will adding two initialized variables cause issues?

* UbiTect adopts guided symbolic execution to prune false positives. This sounds good. However, only about 3% (4150/147643) of UBI warnings are pruned by guided symbolic execution in the experiment. It seems that symbolic execution is not so effective in pruning false positives. Why?

* In the experiment, the authors performed experiments on known UBI bug detection and new UBI bug detection, and compared with existing approaches. The experiments are comprehensive. Specially, 37 UBI bugs detected by UbiTect have been confirmed by Linux maintainers. It is impressive!

* However, it is unclear why only UBI bugs validated by guided symbolic execution in 2 minutes are investigated. Have you investigated more UBI bugs? Is It possible that the false positive rate will increase when the time limit is higher?

* In Section 6.2, 60 new bugs are found by manual inspection. Why these new bugs are not detected by UbiTect?

===Significance===
* This paper addresses an important problem in Linux kernel, i.e., use-before-initialization bugs. This could be helpful in improving the security of the Linux kernel.

===Novelty===
* UbiTect combines flow-sensitive type qualifier and symbolic execution to achieve high precision and scalability in detecting UBI bugs in the Linux kernel. This is novel to me. However, it seems ineffective to prune false positives by guided symbolic execution.

===Recoverability, Replicability and Reproducibility===
* UbiTect tool will be made publicly available. There should not be issues in this aspect.

===Presentation===

* This is a well written and organized paper. It was easy to read.

* How many UBI bugs detected by UbiTect are confirmed by Linux maintainers? The number (37) is not clearly presented in Section 6.2.

* The number 147,644 in Section 6.3 is not consistent with 147,643 in Section 6.2.

Questions for the authors’ response
----
1. Are the two assertions in UBI bug detection representative? Are there some other assertions to detect UBI bugs?

2. Guided symbolic execution seems ineffective in pruning false positives. Why?

3. 60 new bugs are found by manual inspection. Why these new bugs are not detected by UbiTect?

* *

Review #172D
===

Metareview

This paper proposes a promising approach to detecting Use-Before-Initialization(UBI) bugs. All reviewers believe that this is a solid contribution. Nice work! We hope the authors could address the reviewers' comments in the final version.