Custom Memory Functions Demystified: A tutorial of memory corruptions detection using Goshawk

1   Abstract

Static analysis identifies bugs without running the code. However, to find bugs in real world complex software, even those modern static analysis tools suffer from scalability issues, and there still remains a huge gap between research based analysis and pratical bug detection. In this tutorial, we demonstrate Goshawk, a static analysis tool to find memory corruption bugs with the help of Natural Language Processing (NLP), and our audiances are expected to 1) leverage Goshawk to find memory-related bugs in real world C/C++ projects in several minutes; 2) learn how to utilize latest AI techniques to extend static code analysis.

2   Motivation

Existing static analysis tools for the automated memory-related bug (Double Free, Use-After-Free, etc) detection are not effective, especially when analyzing complex projects with large code bases. This ineffectiveness is mainly caused by the following two reasons: programmer likes to 1) design non-standard nested allocator for multi-object struct and 2) follow an unpaired-use model between allocators and deallocators.

In 2022, we presented a new memory bug detection tool--Goshawk[1] (published at IEEE S&P '22). Goshawk introduces the concept of structure-aware and object-centric Memory Operation Synopsis (MOS) to enhance memory bug detection. The MOS model abstractly describes the memory objects of a given MM function and how they are managed by the function. By utilizing MOS, Goshawk are capable of handling the above two development characteristics (achieve accuracy) while exploring much less code (achieve efficiency).

We conducted a comprehensive test on OSS projects such as OS kernel, OpenSSL and IoT SDKs and compared Goshawk with other SOTA data-flow driven bug detection prototype, such as Clang Static Analyzer's MallocChecker[2], K-MELD[3] and SinkFinder[4]. Goshawk outperforms those tools by an order of magnitude in speed and accuracy. By the time of publication, Goshawk has detected 92 new double-free and use-after-free bugs and reported them with developer-friendly MOS descriptions.

Goshawk is constantly evolving after it has been published and it is now public on GitHub[5] and project website[6]. While the website comes with a simple tutorial which takes OpenSSL as an example, Goshawk can also handle other C/C++ projects.

2.1   New version of Goshawk

Technically speaking, the initial version of Goshawk has a two-stage working process: 1) memory-management function and MOS annotation and 2) Data-flow analysis based bug detection on those functions. Between the two stage, user must keep the intermediate output and call Codechecker in command line manually, which is not very convinient and error-prone for novices. The latest version of Goshawk combines the two stages into one script run.py and users can directly view the analysis results.

We have also updated its functionality of each stage. As for the first stage, Goshawk has provided some custom functionalities. Users can [re-train Siamese network](#) for their customized target function identification task and use that model for inferring similarities between function prototypes. This work greatly expands the capabilities of Goshawk, making it no longer limited to memory management functions.

As for the second stage, we have ported the [Clang Static Analyzer checker plugin](#) in Goshawk to Clang-15.0.0 for supporting modern C/C++ projects, and re-organized the four checkers into one entry checker--[GoshawkAnalyzer](#).

2.2 New findings with Goshawk

Consequently, we have applied Goshawk to a wider range of softwares and get more surprising results. In addition to the vulnerabilities which are already listed on the buglist [7], we have discovered UAF bugs in general user programs, like [nasm-2.15.05](#), [cairo-1.17.4](#), [flite](#). We have also discovered 15 bugs (13 UAFs and 2 Double Frees) in 90 OpenWrt pacakges, like [libcoap-4.2.1](#), [kplex-1.4](#), and we manually confirm them. Generally speaking, the average length of bug's data-flow trace is 16.86, which means they are hard to explore manually. Please visit our website [6] for details.

We have been continuously finding new bugs with Goshawk and comfirming their detials with developers. Once approved, we will show the step-by-step process of uncovering those vulnerabilities to every audience of this tutorial.

In a nutshell, we believe it's time for goshawk to showcase itself once again on the world stage!

3 Objectives

This tutorial is to present two technical contributions of Goshawk.

3.1 Apply NLP model to bug detection

One of the most innovative idea of Goshawk is the MOS model. From the point of view of its implementation, it is actually a NLP model for identifying function prototypes. While Goshawk just proved its worth in identifying memory functions, we will show in this tutorial how to apply it to other funtions with nested feature, such as cryptography-related and network-related functions. Then we will be able to discover bugs like crypto-misuse or command-injection.

3.2 Apply open-source SAST tool to real-world projects

We already have got a lot of great C/C++ static application security testing (SAST) tools, but as the empirical study [8] says, 66% of their invesgated GitHub projects define how to use specific SAST tools, but only 37% enforce their usage for new contributions. This is mainly because of the following two reasons:

1. Open-source SAST tools reports a high (47-80%) rate of FP/NP [9], increased the burden of developers;

2. Open-source SASTs tools are not customized for projects, thus it needs to be configured manually in order for the tools to work effectively.

The above two problems are especially obvious when facing large projects, but Goshawk can easily solve them with the following four features: 1) The MOS model are built automatically, users only need to provide the initial corpus of function prototypes to obtain a project-specific MOS model. 2) Goshawk uses [Clang Static Analyzer](#) as its analysis engine, which is part of the Clang compiler infrastructure. 3) Goshawk only performs data-flow analysis on those interested functions and prunes other irrelavant functions. 4) Goshawk uses [Z3 Solver](#) to eliminate false warnings due to infeasible paths.

4   Target Audiences

We assume this tutorial is beneficial for the following three types of audiences:

1. Who want to learn the internal of C/C++ static analysis for memory-related issues;

2. Who want to get familiar with Goshawk, Clang Static Analyzer and their idiomatic usages;

3. Who want to integrate security analysis into their development workflow, especially in large projects.

5   Outline

Below is the timeline of our tutorial, about 90 minutes in total:

Order	Duration	Activity	Content
1	5'	Preparation	Distribute our prebuilt Goshawk docker image
2	30'	Introduction	Introduce Goshawk and its internal implementation
3	10'	Demo	Show Goshawk's basic usage for finding memory-related bugs
4	20'	Interaction	Ask audiences to try to find memory-related bugs by themselves
5	10'	Co-review	Collect the output of Goshawk from each audience and review them together
6	5'	Comparison	Compare with Clang Static Analyzer's MallocChecker
7	10'	Extension	Show how to further modify and extend Goshawk to your security research
8	NaN	Q&A	

Below is the detailed description of each activity:

1. We will upload docker image to DockerHub. Audiences are required to download it and create its container for later interactive activity;

2. We will mainly introduce Goshawk's principle and Clang Static Analyzer's mechanism;

3. We will show Goshawk's basic usage taking a prepared real-world buggy program as an example. Then we show the intermediate output and analysis results;

4. Audiences will try to find bugs by themselves in our provided about 15 real-world open-source projects, and there will be tutorial assistants in the auditorium to help answer questions;

5. We will provide a web interface to upload analysis results and display them together. We will analyze those 15 projects in advance. 1) For those easily comfirmed bugs, we will reward their finders; 2) For those implicit bugs or bugs we have not detected, we will confirm them with finders after the tutorial;

6. We will analyze those 15 projects using Clang Static Analyzer's MallocChecker in advance, and show the differences between the results of two analyzers.

7. We will show how to design and re-train MOS model for identifying cryptography function and helping discovering crypto-misuse issues;

8. We will answer questions about Goshawk's usage and other related topics.

6   Speaker

- Xiang Chen

    – affiliation: Shanghai Jiao Tong University / Shanghai Qizhi Institute

    – biography: Major in cyber security (master degree) at Shanghai Jiao Tong University and as a member of G.O.S.S.I.P, Xiang Chen is now focusing on applying static program analysis to find bugs effectively in real world projects. He is one of the current maintainers of the Goshawk project.

- Siqi Ma

    – affiliation: The University of New South Wales

    – biography: Siqi Ma is currently the senior lecturer of the University of New South Wales. She mainly works in the area of software security. She has published over 40 papers to the top conferences in the areas of cybersecurity and software engineering, such as Security & Privacy, Usenix Security, International Conference and Software Engineering.

7   Reference

1. https://goshawk.code-analysis.org/sp22.pdf ↵

2. https://clang.llvm.org/doxygen/MallocChecker_8cpp_source.html ↵

3. https://www-users.cse.umn.edu/~kjlu/papers/k-meld.pdf ↵

4. https://rucsesec.github.io/papers/FSE2020.pdf ↵

5. https://github.com/Yunlongs/Goshawk ↵

6. https://goshawk.code-analysis.org ↵ ↵

7. https://github.com/Yunlongs/Goshawk/blob/master/bug_list.md ↵

8. https://link.springer.com/article/10.1007/s10664-019-09750-5 ↵

9. https://dl.acm.org/doi/10.1145/3533767.3534380 ↵