**Week 4: C++ In Research (Part B)**

Nate Bachmeier

March 31, 2019

TIM-8101: Principals of Computer Science

Northcentral University

# Examine current research being done with C++

“Coppelia is an end-to-end tool that given a processor design and collection of security critical requirements, can automatically generate exploit code (Zhang, Deutschbein, Huang, & Sturton, 2018).”

The authors acquired a list of 31 known vulnerabilities in two open source processors. Their goal was to reproduce these issues purely through automated discovery technics. The system detected 29 of the 31 issues and found four new issues. After finding an issue, Coppelia will generate a short exploit script to reproduce the issue. This was used to confirm no false positives were reported and provide a mechanism to validate future patches are accurate.

# How did C++ fit into the research?

The first challenge was to get the processor designs into a format which standard tools can consume and manipulate. The researches decided to accomplish this by compiling the Verilog hardware design language (HDL) into C++. This reduced the time to start collecting results as rich open source tooling already exist for C++.

After they generated the C++ code it was then compiled another level into LLVM Intermediate Representation (IR). KLEE is a general-purpose test case generator, which looks at the branches within a function and determines the argument set to gain the maximum coverage (KLEE Team, 2019). These determinations are driven by IR byte code source files.

# What were the research questions?

With the build pipeline configured the researchers could then focus on the goal of efficiently finding processor code defects. They settled on an approach where they begin at the end state of a security constraint was violated. Then they would traverse backwards through the call graph and determine what paths can eventually lead to this. If a path could not be found within a configured number of steps, that branch would be pruned from the search space and the next tried.

For example, one of the vulnerabilities was caused by an incorrect branch taken when comparing a 16bit and 32bit value. If the most significant bit was set on the 32bit value it resulted in an integer overflow. Their system was able to determine example values and the specific assembly instructions required to trigger vulnerability.

# What did the authors find?

# What is some future work which could be built upon this research?

# What are some similar/contrasting works and their findings?