**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 (1) a processor design and (2) a collection of security critical requirements, can (3) 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 on physical hardware that no false positives were reported and provides a mechanism to validate future patches.

# 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.

This build pipeline ultimately provides a mechanism for rapidly simulating realistic attacks against the processor. If they had used real hardware collecting the results would have been more tedious. Simulated processors are also cheaper as they can use commoditized hardware instead of specialized units.

# 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 that begins at the end state of a security constraint being violated. The system traverses backwards through the call graph and determine which paths lead to eventuality. 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 the vulnerability.

# What did the authors find?

The initial approach was a naïve search strategy which enumerated through each of the combinations. This resulted in over a million test cases and no defects after 24 hours. It was then concluded that a smarter search method was required as the simple search was taking exponential time.

The search algorithm was replaced with 10,000 iterations of breath first search (BFS) followed by 500,000 iterations of depth first search (DFS). This resulted in a very wide net that could spider as required in parallel. Using the hybrid BFS+DFS model reduced the time required to find defects from hours to minutes. This makes sense as Table II shows that most exploits are only 1 to 5 instructions.

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

## More Hardware

Security vulnerabilities that are printed into hardware applications are inherently expensive. A recent example can be seen in the Intel Meltdown vulnerability which enables a user mode process to read arbitrary physical memory (Horn, et al., 2018). The issue impacts all Intel processors since 2010, which is nearly a decade of vulnerable hardware that is unlikely to be fixed.

Having systems such as Coppelia involved in the design process early on can mitigate some of these scenarios. This creates an economic incentive to expand the approach across more processors. A specific area to investigate would be Internet of Things Socket on Chip (IoT SoC). These complex chips contain the attack surface of an entire server and historically do not have the best track record (Kamble & Bhutad, 2018).

## More Languages

The researchers compiled Verilog source code into C++ so that they could leverage extensive open source tooling. A similar approach might be possible for other niche languages. This would allow the application to quickly gain these specializations and identify complex application faults.

## Injected Specializations

These specializations are not limited to vulnerability detection and could also include modern protections that are not possible in legacy systems. For instance, the legacy application can be compiled into IR and then a mutator instrument all file I/O operations with calls to a hardware encryption device (Gossen, Beubauer, & Steffen, 2017).

Having the ability to inject specializations into software could lay the foundation for automated consulting firms. Image uploading line of business (LOB) code to a web service and it automatically instruments more efficient patterns directly into the Abstract Syntax Tree.

## More Targets

The journal focused specifically on finding code defects in hardware implementations, but a similar approach might be possible for system kernels. Within a kernel is a collection of system calls and security constraints. It might be possible to augment the search and symbolic execution components to fit this similar model.

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

## Blackbox Approach

One of the challenges with the proposed approach is that it starts at the security violation and walks backwards. What happens if we do not know some or all these places? Clearly issues will be missed.

An alternative method is to start with black box fuzzing as a mechanism to select the components to focus further attention (Avgerinos, et al., 2014). Avgerinos provided an example where every binary in the /usr/bin was passed a valid 16kb file path. Within 15 minutes the simple test found 59 distinct crashes.

These crashes were used to identify which developers misunderstood low-level memory management or the implementation of various API sets. In some ways this is analogous to the BFS+DFS model used by Zang, et al to rapidly find the specific subset of interest.

# References

Avgerinos, T., Kil Cha, S., Rebert, A., Schwartz, E., Woo, M., & Brumley, D. (2014). Automatic Exploit Generation. *Communications of the ACM Vol 57. No. 2*, 74-84.

Gossen, F., Beubauer, J., & Steffen, B. (2017). Securing C/C++ applications with a SEcube™-based model-driven approach. *12th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*.

Horn, J. H., Prescher, T., Gruss, D., Lipp, M., Mangard, S., & Schwarz, M. (2018). Meltdown: Reading Kernel Memory from User Space. *27th {USENIX} Security Symposium ({USENIX} Security 18*.

Kamble, A., & Bhutad, S. (2018). Survey on Internet of Things (IoT) security issues & solutions. *2018 2nd International Conference on Inventive Systems and Control (ICISC) Inventive Systems and Control (ICISC)*, 307-312.

KLEE Team. (2019). *Testing a Small Func*. Retrieved from Github: http://klee.github.io/tutorials/testing-function/

Zhang, R., Deutschbein, C., Huang, P., & Sturton, C. (2018). End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. *Annual IEEE/ACM International Symposium on Microarchitecture*, 815-827.