Last time

Memory safety

Memory-safe language concepts

Safe unsafe languages?

So... perfection?

Write all software in a memory-safe language?

TCB considerations

Memory safety in compiled languages

1. Compiler-added run-time safety checks
2. Limited unsafety
3. Continued dangers of native instructions

High-level language interpreters have to be written in something. You might be able to write a lot of a Java interpreter in Java, but at the lowest levels you will find lots and lots of C++ code. At the lowest levels of the C standard library, you will find ____________________, sometimes ____________________________.

Languages like __________ and ____________ claim to provide memory safety, but they are compiled languages. How is this possible?

The compiler can add extra code to check some accesses at run time. For example, if you are indexing within an array, the compiler can implicitly add code such as `if 0 <= i < n`.

Languages that aspire to "systems programming" (i.e., things that have to be aware of or manipulate the lowest-level primitives such as hardware registers) have to allow for unsafe operations. There is no memory-safe way to perform arbitrary register, memory or I/O operations, so these kinds of languages have to provide some way to break abstraction layers. C code can include assembly via the `asm` keyword. Rust code can explictly violate memory safety guarantees if it uses the `unsafe` keyword.

Even with those checks, however, if you load someone else's native instructions and execute them, ____________________!

Safe compiled code?

What is a language?

Software

AddressSanitizer, CCured, Cyclone, "fat pointers", Go, Rust, ...

Hardware:

Arm MTE, CHERI, Hardbound, MPX, segmentation, Watchdog, ...

When we think of a language, we typically think about ____________________ and the ______________ for writing it. However, in addition to ______________, we also have ______________ that are defined by language specifications and — crucially — ____________ _____________________. If we take this expanded view of what makes a language, we can see a number of approaches applied in various places that can be used to improve the security of compiled code, too.

Software

AddressSanitizer (and other "sanitizers" like Thread Sanitizer and the Undefined Behaviour Sanitizer) can help spot memory errors during testing that might otherwise have gone unnoticed. CCured is an example of an approach that uses static analysis to figure out how pointers in a C program are "meant" to be used and dynamic analysis to ensure that they are, in fact, used that way. Cyclone is a C dialect with better memory safety properties than vanilla C, which it is designed to be compatible with (or at least easy to adapt from). Newer languages like Go and Rust have more expressive type systems that make it possible to write memory-safe code even in high-performance compiled languages with limited run-time checking.

Hardware

Arm MTE has been adopted by Android to detect memory safety violations at run time. Hardbound, MPX and Watchdog attempt to provide various forms of hardware memory safety enforcement. CHERI is a designed-for-security instruction set extension for ARM and MIPS that is just about to ship its first hardware prototypes; it has the potential to change _________________ by allowing high-level object accesses to be precisely enforced by hardware.

Today

Finding memory safety violations

- testing
- formal methods
- hybrid approaches

Testing

Default approach... if we're lucky!

Code coverage

- what's that?
- `gcov`, `llvm-cov`, `SanitizerCoverage`...

Limitations

Hopefully software developers are employing practices such as ______________ ________________. Testing software helps us find defects, but it's more valuable than just that: designing software to be testable also tends to encourage ___________________ via _________ ______________ with _________________________, all of which also help with software quality. So, testing is good in more than one way, and we know we ought to do it! Let's be honest, though: we don't always do what we know we ought to do. And even when we do, __________________________________?

We'll have a little demo involving code coverage data, how it's generated and how it's used.

Testing is great, but ultimately testing can never ________________________________: it can only _________________________________.

Formal methods

Modeling programs

Proving properties of programs* and compilers†

Curry-Howard Correspondence

* Can prove more than you'd think, e.g., Cook et al., Proving that programs eventually do something good, *ACM SIGPLAN Notices 42(1)*, 2007 (DOI: 10.1145/1190215.1190257) and Cook et al., Proving Program Termination, *Communications of the ACM 54(5)*, 2011, (DOI: 10.1145/1941487.1941509).

† e.g., Kästner et al., CompCert: Practical experience on integrating and qualifying a formally veritied optimizing compiler , *ERTS 2018: Embedded Real Time Software and Systems*, 2018.

Constructing a model of a high-level functional program is one thing: after all, functional programming languages take their cues from mathematics! Constructing a model of a low-level program in a language with lots of ________________ and pointer-based ________________, however, is something else.

People do, however, use formal methods in real (albiet someone size-limited) systems. The seL4 microkernel has been formally verified by first verifying a model, then generating code *from* that model, then proving that the generated code corresponds *to* the model. That's not quite proving properties about C code, but it's close!

In addition to proving properties about software in its source code, one practical (though computationally-challenging) approach is to prove properties about software in its __________ ____________ form. This is useful for the artifact that ultimately matters most: ________ ________________________. To do this, you need a formal model not of your software, but of the hardware it will execute on.

People also use _______________________ to prove that the output of a compiler really matches its inputs. This is meant to address the problem raised in Ken Thompson's famous Turing Award lecture, Reflections on trusting trust. COMPCERT is such a compiler... although every new release "fixes a few bugs"??

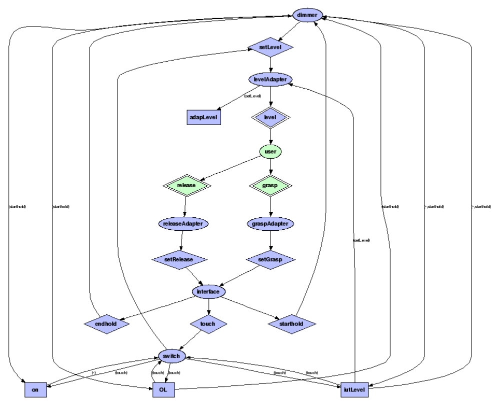
If you like monads, π-calculus or intuitionistic modalities... or just _____________ __________! Why are functional programs easier to verify? Because the language __________________________________. The more sophisticated the type system, the fewer programs you can write... with the caveat that the programs you *can't* write are the ________________ ones. For a lot of programmers (including me!), a ____________________ is more natural to work with than a _____________________.

Hybrid approaches

Concolic testing:

- model checking
- concolic execution

Fuzzing



Concolic is a portmanteau of ________________ and ________________. It aims to be practical (the concrete execution part, not just all theorems) while also gaining some of the generality of symbolic approaches.

Model checking is heavily used by hardware engineers to verify (not just ___________) things like state machines.

Concolic execution can be applied to real software using tools like KLEE. In such testing (for which we'll have a brief demo if there's time), we can run real test cases while telling the tool (KLEE in this case) that it should treat some values as ________________, i.e., effectively try ________________________ instead of one specific one. This can help us spot tricky corner cases that might escape our testing!

Fuzzing is the process of running software with inputs that we mutate a little bit every time we run it. This isn't strictly as powerful as concolic execution, but it benefits from being highly practical: we can fuzz software _________________________ for concolic execution.

Fuzzing

Black-box fuzzing

Glass-box fuzzing

DART, SAGE, AFL, LibFuzzer

OSS-Fuzz and its trophies

Black-box fuzzing mutates input data with no knowledge of the target program's internals. This can find bugs, but it's nowhere near as powerful as *glass-box* fuzzing.

In glass-box fuzzing, we allow the compiler to instrument programs, much like ____________________. This information is then used to decide whether we've explored the control-flow graph of a particular function, etc., "enough". It can also be used to decide whether two fuzz-created crashes are ______________________________!

There are lots of tools for doing glass-box fuzzing. We'll have a demo (time permitting) of AFL: American Fuzzy Lop (named for a species of fuzzy rabbit).

The large-scale OSS-Fuzz project applies fuzzing to a number of critical, widely-used open-source software projects. It includes components with fun names like ClusterFuzz, and it also has quite a few trophies in large, complex projects.

Summary

Testing

Formal methods

Hybrid approaches

… to *mitigate* risk

All of this work — with the exception of formal proof, which is limited in scope — serves to *mitigate* the risks of imperfect software.