How Should We Think About Memory-Safe Languages?

Matthew Sottile (sottile2@llnl.gov)
Center for Applied Scientific Computing

January 18, 2024

Outline of talk

- Memory Safe Languages: Why?
- Examples in the wild
- Success stories and skepticism
- Why should we care at LLNL?
- How should we think about this area for our work?

This should take less than the full time period by design: I'm happy for questions during the talk, and want to leave time for people to ask questions.

Definition: memory safety issues

- Memory safety issues are defects that lead to unexpected code execution, data loss/corruption, system instability.

Examples (not exhaustive):

- Out of bounds memory accesses
- Null pointer dereferences
- Data races
- Uninitialized variables
- Memory leaks
- Use-after-free errors

TL;DR

- Memory-safe languages (MSLs) summed up in one line:

A technology and mindset shift from **opt-in** safety to **opt-out** safety.

- That's basically it. What does that mean though?

Opt-in model

- Language definition provides no safety guarantees out of the box.
  - Undefined behavior
  - Pointer arithmetic
  - No bounds checked arrays
  - Manual memory management

Opt-in model

- Language definition provides no safety guarantees out of the box.
  - Undefined behavior
  - Pointer arithmetic
  - No bounds checked arrays
  - Manual memory management
- Opting in to safety is possible through:
  - Discipline, ”best practices”, and careful design
  - Safe components in language standard libraries (std::fancy_ptr) or third parties (Boost::fanciness)
  - Static analysis tools and runtime assertions: compiler can reason about only limited safety issues
  - Unable to have the compiler exclude unsafe parts of the language

Opt-in model

- Language definition provides no safety guarantees out of the box.
  - Undefined behavior
  - Pointer arithmetic
  - No bounds checked arrays
  - Manual memory management
- Opting in to safety is possible through:
  - Discipline, ”best practices”, and careful design
  - Safe components in language standard libraries (std::fancy_ptr) or third parties (Boost::fanciness)
  - Static analysis tools and runtime assertions: compiler can reason about only limited safety issues
  - Unable to have the compiler exclude unsafe parts of the language
- Basically, what we’ve done for the whole time C and C++ have existed (40-50 years).
  - “Trust us, we’re going to get it right this time!”

Opt-out model

- Dangerous features are not part of the language by default.
  - Pointers are more limited, Hoare's "billion-dollar mistake" not present
  - Undefined behavior eliminated
  - Bounds checking on memory
  - Automatic memory management (ARC, GC, static borrowing/lifetimes)

Opt-out model

- Dangerous features are not part of the language by default.
  - Pointers are more limited, Hoare's "billion-dollar mistake" not present
  - Undefined behavior eliminated
  - Bounds checking on memory
  - Automatic memory management (ARC, GC, static borrowing/lifetimes)
- When we need to get to the metal and work without safeties on, we do so in a controlled, isolated way.
  - Unsafe regions, FFI, etc.

Opt-out model

- Dangerous features are not part of the language by default.
  - Pointers are more limited, Hoare's "billion-dollar mistake" not present
  - Undefined behavior eliminated
  - Bounds checking on memory
  - Automatic memory management (ARC, GC, static borrowing/lifetimes)
- When we need to get to the metal and work without safeties on, we do so in a controlled, isolated way.
  - Unsafe regions, FFI, etc.
- Reduce chance of human error, reduce reliance on "discipline", promote static checking of safety properties.
  - Mistakes are avoidable if they can't be written down in the first place!

Why now?

- We've had 40+ years to get the opt-in model of safety right.
- The computing community appears to have come to a realization:

Cost(adding memory safety to existing languages) +
Cost(building analysis tooling to find safety bugs) +
Cost(fixing safety issues existing languages allow) + …

>

Cost(adopting new language for new code) +
Cost(integration with existing code) +
Cost(workforce development) + …

Why now?

- We've had 40+ years to get the opt-in model of safety right.
- The computing community appears to have come to a realization:

Cost(adding memory safety to existing languages) +
Cost(building analysis tooling to find safety bugs) +
Cost(fixing safety issues existing languages allow) +
Cost(legal liability of software defects) +
Cost(regulatory requirements about software) + …

>

Cost(adopting new language for new code) +
Cost(integration with existing code) +
Cost(workforce development) + …

I think people anticipate these coming in the future!

Skepticism

- No universal agreement about which route is best:
  - sort-of-safer opt-in,
  - or opt-out model.
- Still easy to write bugs in MSLs :
  - Tons of non-memory related bugs to keep us busy.
- Very real concerns about MSLs:
  - Learning curve.
  - Long-term stability of languages/infrastructure.
  - Supply chain considerations vs semi-chaotic OSS world.
  - Limited current capabilities in certain areas like HPC.

Skepticism: LKML example

- Some recent LKML comments (searching January ‘24 LKML archive for “rust”)

Syntax painful, difference in language felt at design level.

```
Beyond the syntax, which I'm trying to force myself not to focus on, the
compatibility layers are turning out to be quite extensive. This is just
another way of saying that Rust is a deeply, completely different language.
Whereas C++ is closer to a dialect, as far as building and linking anyway.
```

Workforce issue: skills gap

Immaturity

```
The biggest merit of using C++ in the kernel is that in comparison to
other systems language (Zig, Rust, Swift to name a few) it requires the
least re-skilling of existing contributors. A close second would be the
low barrier to integrate various C++ and C codebases. Especially when
taking into account the architectures that the kernel needs to support
vs the other languages. Even Rust with its big push towards being a
replacement isn't there yet today (e.g. PA-RISC).
```

Recognition of what I mentioned earlier.

```
If you want C++ to do what Rust could do, then you need the compiler
to stop the stupid before you can even write it, otherwise people will
still write the bad stuff and code review won't catch it all.
```

Examples in the wild: Rust

- The one you've probably heard a lot about
- First mainstream MSL that fits systems developer needs:
  - Lifetime/borrowing model of memory w/ static enforcement: no runtime GC or ARC
  - Ownership model well suited to reasoning about data races
  - Allow unsafe code to integrate with safe code, useful for low level code like device drivers
  - Performance competitive with C and C++
- Seeing strong industrial + open-source uptake
  - Microsoft Windows kernel
  - Android
  - Linux kernel*
  - Lots of web/internet infrastructure

** Linux kernel dev community is not 100% on board with this…. Easy to find grumpiness online.*

Examples in the wild: Go

- Systems language: distant successor to C via Alef/Limbo (Plan-9/Inferno)
  - Emphasis on concurrency
- Google language that has strong presence in networked applications.
  - Memory managed via garbage collection
  - Bounds checked arrays
  - Designed for simplicity, which meant that some features were explicitly excluded from the language
- Strong uptake in the server/infrastructure world.
  - Docker, Kubernetes
  - Companies like Netflix, Dropbox, Twitch, Soundcloud, etc.

Examples in the wild: Swift

- Apple language designed by (among others) Chris Lattner (LLVM fame).
  - Aims to replace C, C++, Obj-C, Obj-C++ throughout all Apple products
  - Automatic reference counting
  - Designed for interoperability with the languages it is replacing
  - Designed to be easy to learn: notably lower learning curve versus Rust
- Obviously strong uptake in Apple (macOS, iOS, etc.) ecosystem.
  - Some uptake in industry in server-side applications
  - Fits well in teams where user-facing app developers and server-side developers desire same development environment
  - Suffers from perception problem (“I don’t target Apple; I don’t do iOS”)
- Open-source language + compiler on all major platforms (macOS, Windows, Linux).

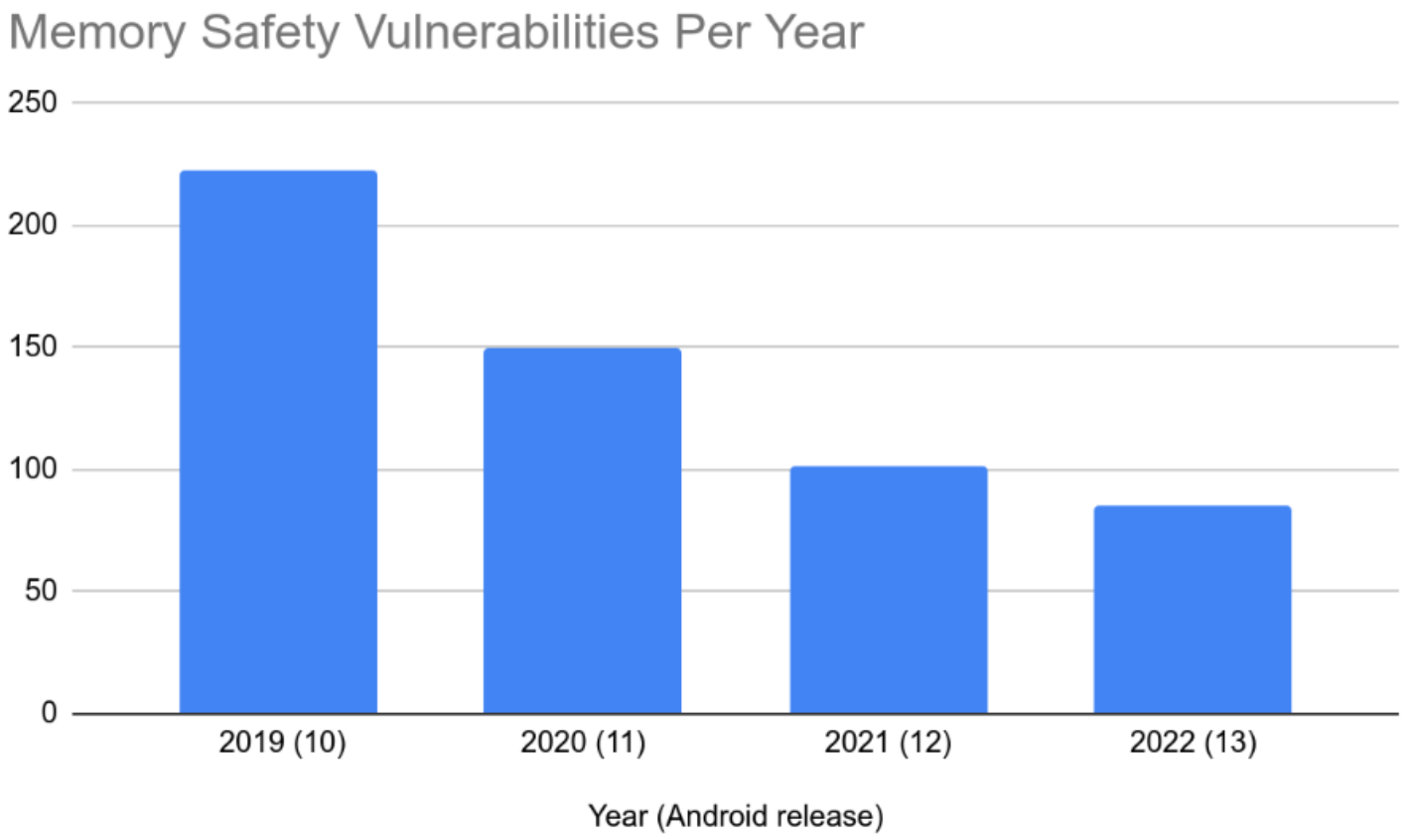
Examples in the wild: Java/C#

- The “managed-code” or “bytecode-based” languages
  - Language targets an abstract machine and is Just-In-Time compiled to real machine
  - Automatic garbage collection for memory management
  - No pointer arithmetic
  - Effects model in type system (exceptions)
  - JVM + .NET support other languages that inherit the safety properties of the VM
    - Scala, Kotlin, F#, Clojure
- Heavy presence in the enterprise world
  - Large scale concurrent systems
  - Databases
- Open source compilers + virtual machines (OpenJDK, Roslyn .NET, Mono)
  - C#/CLI (Common Language Infrastructure) is ECMA standard
  - Java driven by Java Community Process

Examples in the wild: honorable mentions

- **Python**: memory safety, FFI, but dynamic features can lead to other non-memory related safety issues.
- **Ocaml + friends**: memory safety, strong type systems, some industrial presence, but sometimes too far outside comfort zone of die-hard C-family developers.
- **Ada**: many safety features, but possible to work around and write unsafe code. Notable for concept of “profiles” enforceable by compiler, and strong presence in aerospace industry and verification community.
- … and many others I missed.

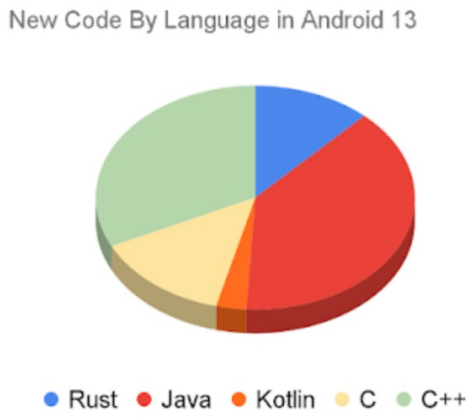
Success story: Google + Android 13



https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

Success story: Google + Android 13

This drop coincides with a shift in programming language usage away from memory unsafe languages. **Android 13 is the first Android release where a majority of new code added to the release is in a memory safe language**.



As the amount of new memory-unsafe code entering Android has decreased, so too has the number of memory safety vulnerabilities. From 2019 to 2022 it has dropped from 76% down to 35% of Android's total vulnerabilities. **2022 is the first year where memory safety vulnerabilities do not represent a majority of Android's vulnerabilities**.

https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

Myths

"MSLs are academic: this is just the languages people doing their usual promotion process to drum up money for their pet projects."

- Industry leading the charge.
- Swift = Apple, C# = Microsoft, Java = Sun/Oracle, Rust = Mozilla, Go = Google …
  - Those aren't universities.
- You aren't hearing people pushing the academic languages we hear about in the PL community.

Myths

”MSLs are a fad. Carry on with current tools and they’ll eventually die.”

- Industry doesn’t like spending money. (I know - I spent 12 years in it between LANL and LLNL)
- Industry is spending **lots** of money on MSL migration.

Myths

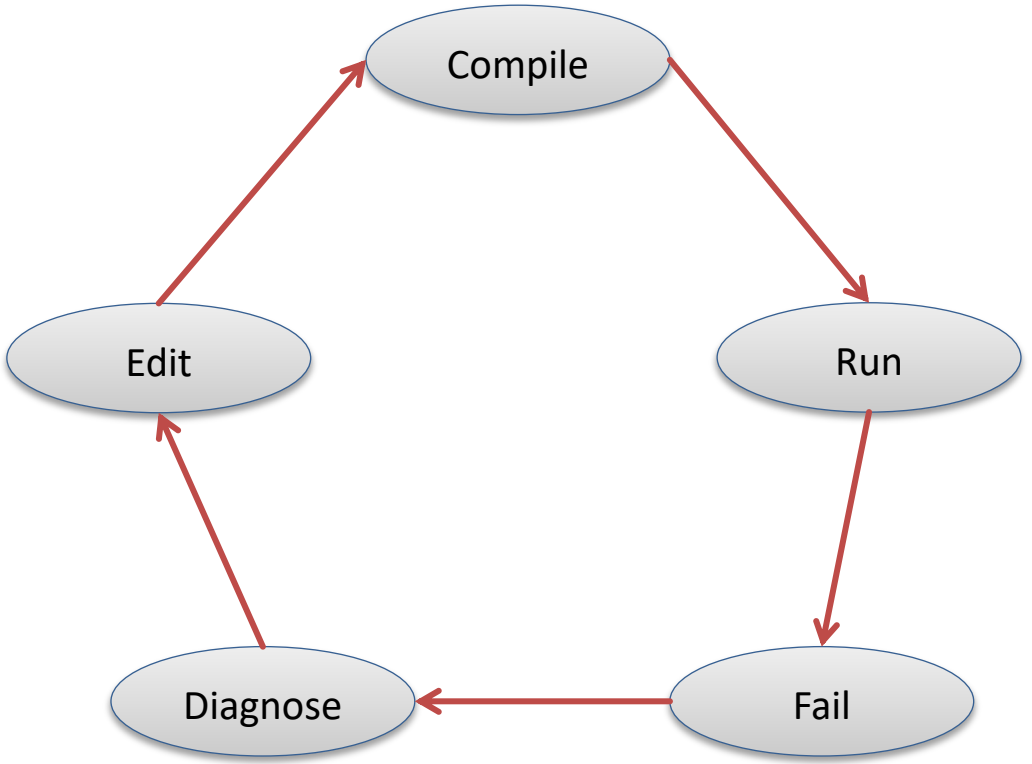
"MSLs are inherently inefficient, and **we** need absolute performance."

- This is a fair critique as many techniques in MSLs were introduced decades ago and performance was bad.
- Compilers in 2024 are better than they were in 1984.
- For really critical performance, the languages aiming at systems developers provide mechanisms to control performance.
  - This is the **opt-out** concept at play: opt-out of overhead when necessary and make safety tradeoff.

Common complaint

"MSL X is frustrating: I fight with the compiler endlessly – I just want to run my code."

- Each compiler rejection is a cycle you don't need to take.
- Many memory safety rejections may not even obviously fail.
- Those that do fail may have very nontrivial diagnose step.
- I would argue we should treat this dialogue with the compiler useful: forces us to really understand what our intention really is for the code.



Why should we care at LLNL? Pressure from above.

Important!

They're not messaging developers: they're messaging the executives and advisors.

The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

Publication: December 2023

United States Cybersecurity and Infrastructure Security Agency
United States National Security Agency
United States Federal Bureau of Investigation
Australian Signals Directorate's Australian Cyber Security Centre
Canadian Centre for Cyber Security
United Kingdom National Cyber Security Centre
New Zealand National Cyber Security Centre
Computer Emergency Response Team New Zealand

Why should we care at LLNL?
Direct safety/security concerns

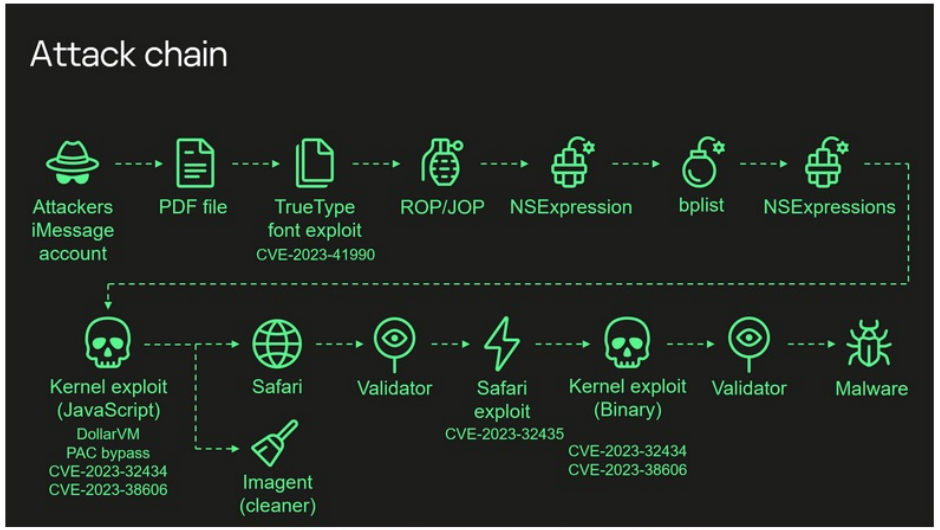
- We build systems with high security and safety requirements.
- We need to make strong, defensible arguments that our software meets those needs.
- Current practices to assess correctness are limited.
  - Testing: better than nothing, but limited defensible extrapolation from tests to general cases
  - Some ”certification” activities appear to be more paper-pushing than actual assurance
  - Static analyzers overwhelm people with false positives, and miss some things
- We should use tools that give us strong assurances about what we build. MSLs are such a tool.

Why should we care at LLNL?
Indirect safety/security concerns

- "I don't write internet-facing code. Doesn't apply to me!"
- Don't be the person who provides the attack surface.
  - I doubt the people who added a TTF instruction in the 90s anticipated being a critical part of a network messaging attack decades later either.

0-day attack chain to 0-click iMessage exploit

This vulnerability existed until iOS 16.2 was released in December 2022.



via Boris Larin, Leonid Bezvershenko, and Georgy Kucherin at Kaspersky

Considerations for applying MSLs at LLNL

- Costs: Time and $
- People factors
- Appropriateness: where do they best fit?
- Timing: when should we care?
- Contributions: where can we help the MSL community?

Costs: Time and $

- This is not free.
- Software development needs funding to pay for people.
  - Professional development.
  - Doing the actual development.
- Funding agencies traditionally hard to convince to pay for software maintenance, tech debt cleanup, modernization.
- There is hope that the pressure from above may convince $ holders to pay for things they wouldn't before.
- But it will be expensive.

People factors

- Support:
  - People who want to push the boundaries should be encouraged to do so
  - People who have concerns should be listened to and not bulldozed over by MSL zealots
- Attachment:
  - Some people really, really love their tools (e.g.: emacs vs vim – discuss)
  - This love for their tools can breed highly negative responses (e.g.: emacs vs vim - discuss)
  - Often this is a manifestation of fear of losing hard earned investments of time and effort. Very valid.
- Cycle of life:
  - New grads who know more than Python often drawn to MSLs
  - Eventually they'll replace existing workforce
  - The shift to MSLs may fall out of this workforce evolution naturally: *but do we have time to wait?*

Appropriateness

- No technology is a one-size-fits-all solution: assuming that is silly
- Rust is running into this with its popularity
  - Super-eager Rust advocates applying it everywhere.
  - It is not pleasant for some applications: end up battling the borrow checker to achieve what I can do just as safely, yet much simpler in other languages.

- Those other languages have their place
  - Positive: appropriate tool for appropriate job is good.
  - Negative: tower of Babel problem – workforce and interoperability chaos if every corner of the lab is based on a totally different language.

LLNL Domains matched to MSLs

- Embedded:
  - VxWorks ships Ada, Rust along with C, C++. Rust appearing in other embedded contexts as well.
- Developer tools:
  - Extreme performance not critical, but need more than Python. Swift, Rust, Ocaml, C#/Java strong contenders.
- User interfaces:
  - TypeScript, Reason, Java/C# all commonly used in user-facing applications and are cross platform.
- Server tools:
  - Go, Rust, C#/Java all have strong industrial foothold in this area (see: devops tools)
- HPC:
  - This is where we can contribute to fill a MSL community gap: we push the HPC community, so we can push the appropriate MSLs in a direction suitable for HPC.

When?

- For some application areas (HPC especially), some of these languages are not quite ready yet.
- Timing needs to account for how critical a system is.
  - Embedded systems, instrumentation, networking, cyber: sooner than later.
  - Data crunching, modsim: eventually.
- Timing should include plans to not just adopt a technology, but the *precursor steps to mature the technology to the point of being ready.*
  - Again, this is where LLNL + DOE should participate in the maturation process.
- Story time.
  - A sad story.
  - A happy story.

Lessons Learned from ASCI

Douglass Post

Los Alamos National Laboratory

International Workshop on Advanced Computational Materials Science for Nuclear Materials

Washington, DC

April 1, 2004

Abridged version of
LA-UR-04-0388

Approved for public release;
Distribution is unlimited

March 30, 2004

Los Alamos

Employ modern computer science techniques, but don't do computer science research.

!!!!!!!

- Main value of the project is improved science (e.g. physics and math)
- Implementing improved physics (3-D, higher resolution, better algorithms, etc.) on the newest, largest massively parallel platforms that change every two years is challenging enough. Don't increase the challenge!!!
- LANL spent over 50% of its code development resources on a project that had a major computer science research component. It was a massive failure (~$100M).

T. Demarco, T. Lister (2002); Post and Cook, 2000: Post and Kendall, 2002

The happy story

- The year is 1996….

The happy story

- The year is 1996….
- Scientific developers want a scripting language good for numerics.

The happy story

- The year is 1996….
- Scientific developers want a scripting language good for numerics.
- Enter: Python + numeric / numarray packages, SWIG wrapper generators.

The happy story

- The year is 1996….
- Scientific developers want a scripting language good for numerics.
- Enter: Python + numeric / numarray packages, SWIG wrapper generators.
- …A few years later, led to NumPy: which powers most SciComp, ML, DS Python today.

- Look at the authors of those packages that set the stage for what we have today.

National Laboratories

Maybe do that again?

Concluding thoughts

- Memory safety (and other forms of language-enforceable safety) are a good thing and pressure is mounting to take advantage of systems that provide it
- Opt-in models of safety haven't really panned out in 40+ years: opt-out appears more trustworthy
- While Rust is the most famous of the MSLs, there are many options that fit the bill
- The right plan is a slow migration of the appropriate parts of existing codebases
- Interim use of safer C++ features is fine to modernize existing codes, but not an ideal solution in the long term
- Where there are gaps, don't reject the languages: participate in filling the gaps, especially for our specific needs and specializations (e.g., HPC)

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC