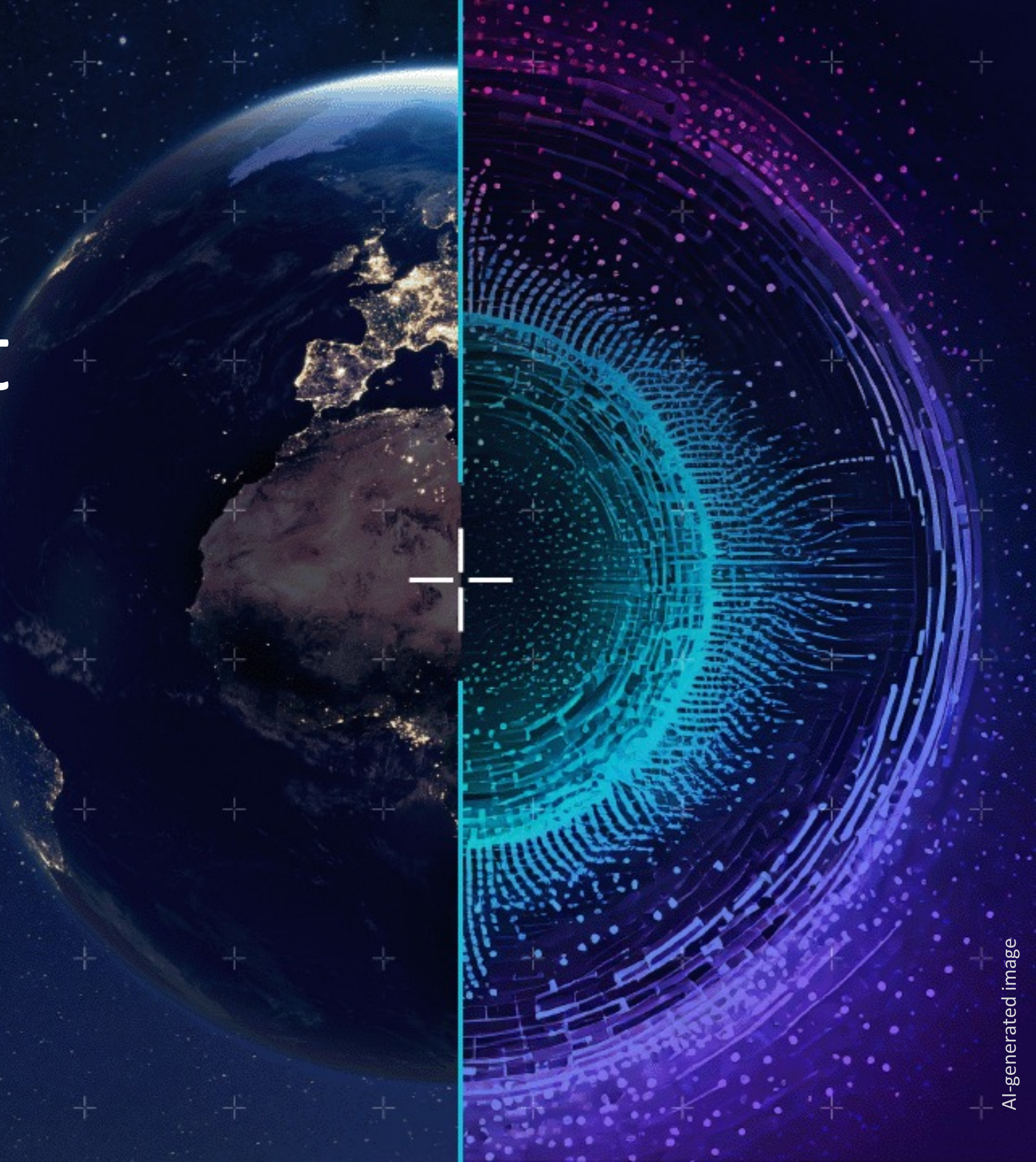
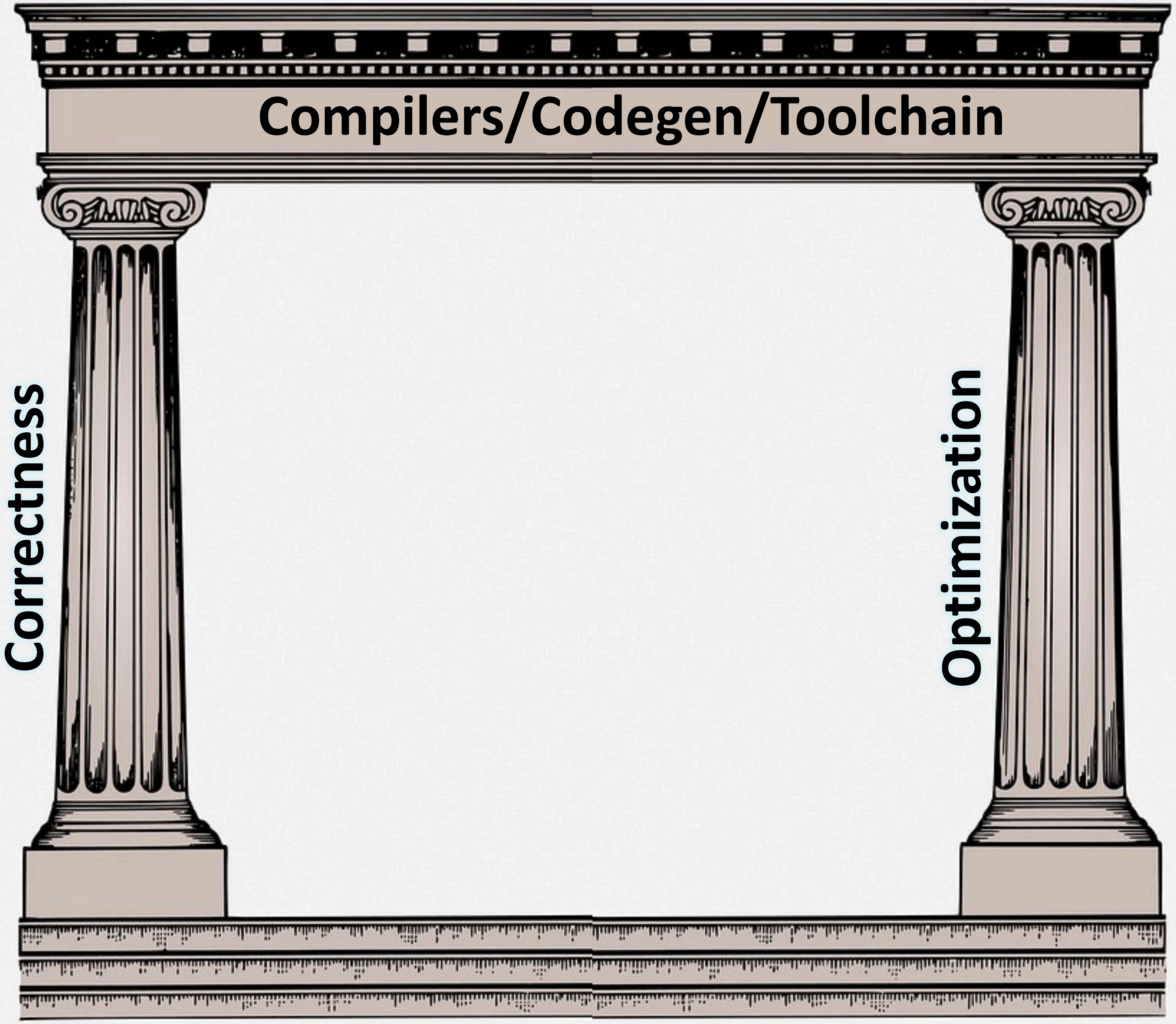
arm

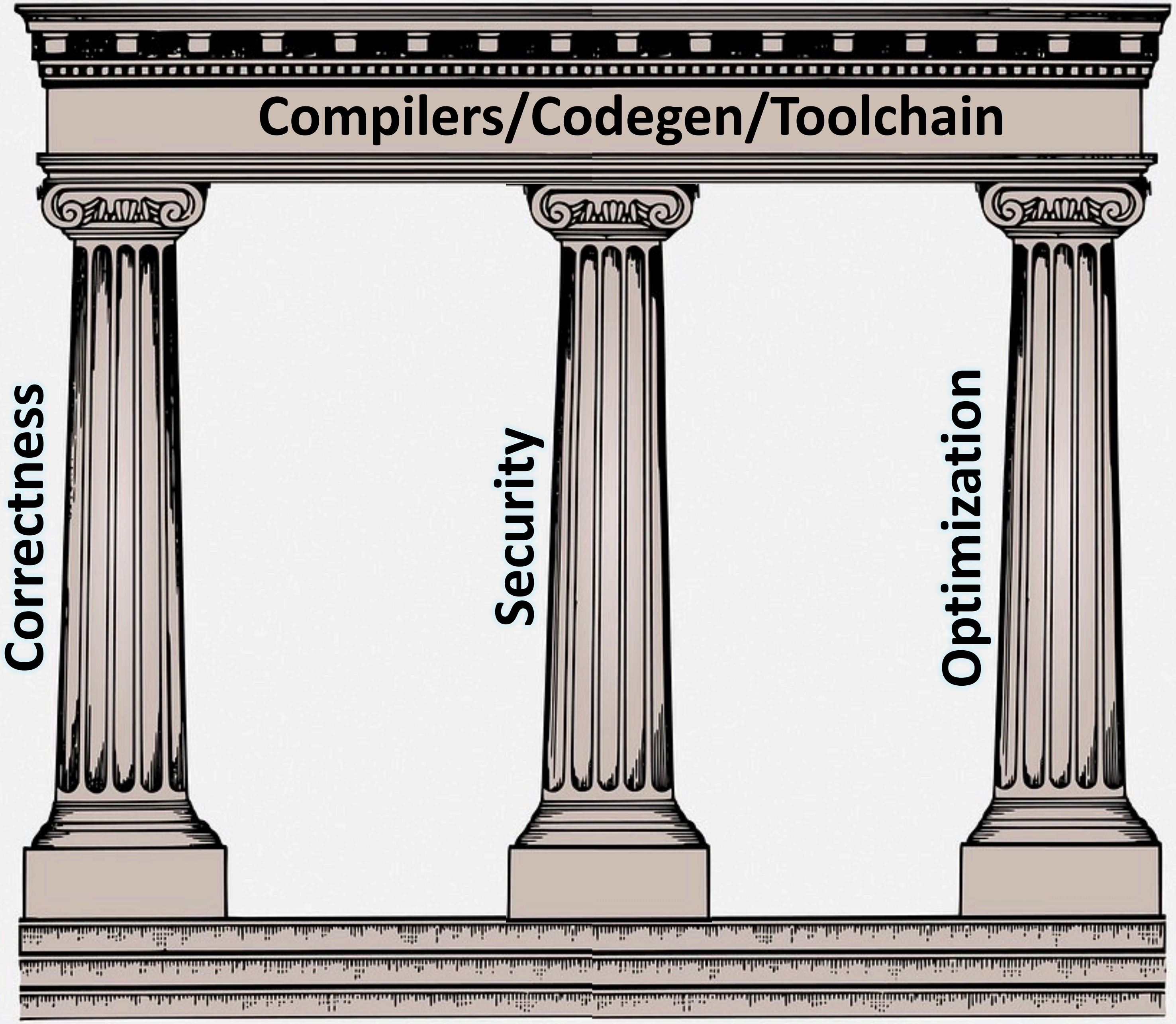
Does LLVM implement security hardenings correctly?

A BOLT-based static analyzer to the rescue?

Kristof Beyls
April 10, 2024




Compilers/Codegen/Toolchain
Correctness
Optimization


Compilers/Codegen/Toolchain
Correctness
Security
Optimization

Compilers & the 3 pillars over time

1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2024

Optimizing
Fortran
compiler

Compilers & the 3 pillars over time

1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2024

Optimizing
Fortran
compiler

Compilers & the 3 pillars over time

1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2024

Optimizing Fortran compiler

Turing award winner Ken Thompson demonstrates malicious compiler patch supply chain attack

Compilers & the 3 pillars over time

1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2024

Optimizing Fortran compiler

Turing award winner Ken Thompson demonstrates malicious compiler patch supply chain attack

Smashing the stack for fun and profit

Compilers & the 3 pillars over time

1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2024

Optimizing Fortran compiler

Turing award winner Ken Thompson demonstrates malicious compiler patch supply chain attack

Smashing the stack for fun and profit

`-fstack-protector` in gcc 4.1

Compilers & the 3 pillars over time

1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2024

Optimizing Fortran compiler

Turing award winner Ken Thompson demonstrates malicious compiler patch supply chain attack

Smashing the stack for fun and profit

`-fstack-protector` in gcc 4.1

Rust, llvm, gcc PSIRT

Compilers & the 3 pillars over time

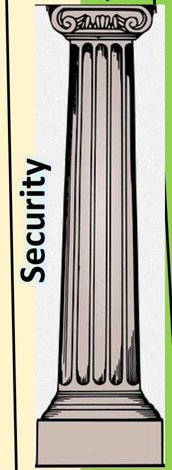
1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2024

Optimizing Fortran compiler

Turing award winner Ken Thompson demonstrates malicious compiler patch supply chain attack

Smashing the stack for fun and profit

`-fstack-protector` in gcc 4.1



Security becoming a third pillar

Rust, llvm, gcc PSIRT

Compilers & the 3 pillars over time

1950 | 1955 | 1960 | 1965 | 1970 | 1975 | 1980 | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2015 | 2020 | 2024

Optimizing Fortran compiler

Turing award winner Ken Thompson demonstrates malicious compiler patch supply chain attack

GCC

Gcc emerges (open source compiler)

Smashing the stack for fun and profit

Compilers are mostly open source

-fstack-protector in gcc 4.1

Security

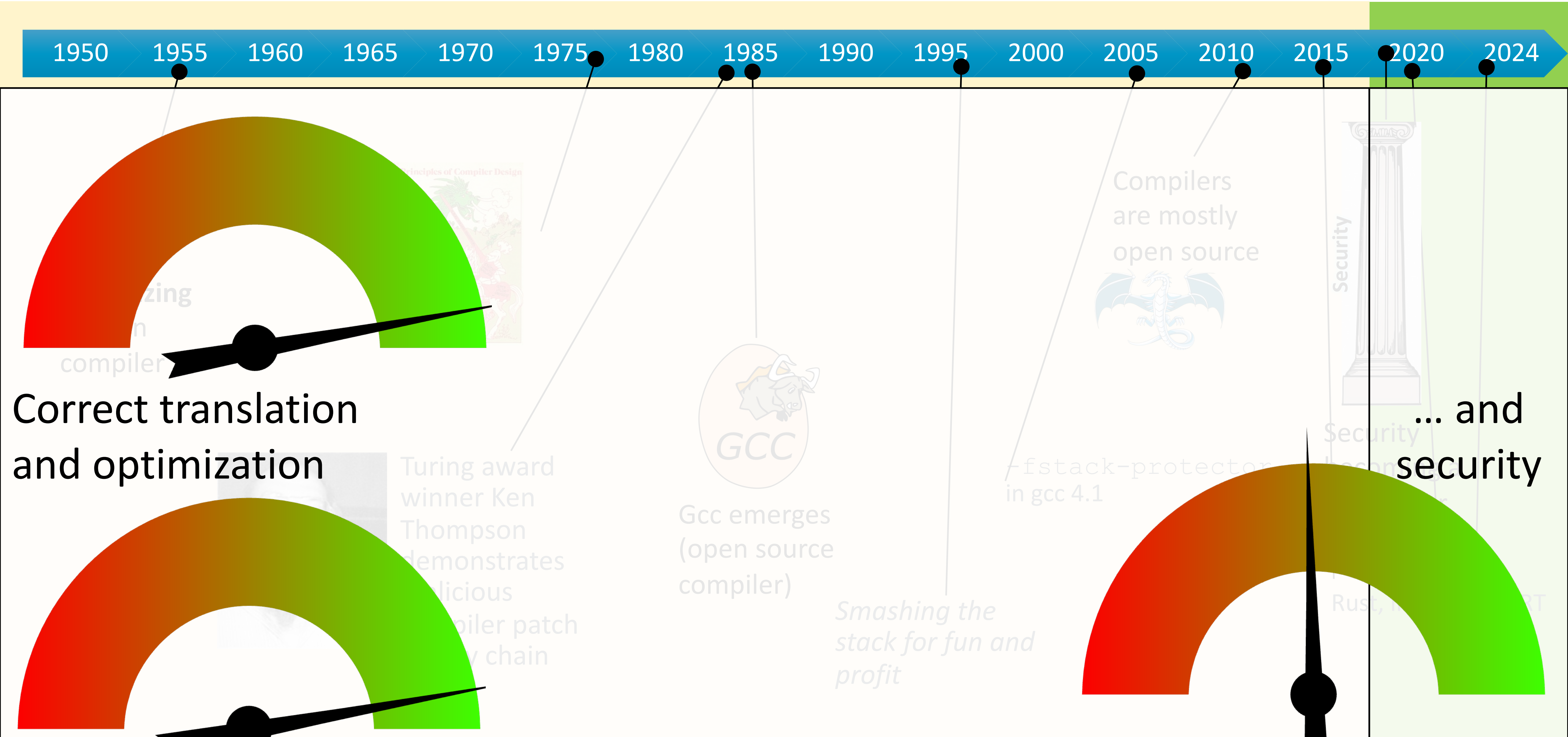
Security becomes third pillar

Rust, llvm, gcc PSIRT

Correct translation and optimization

… and security

Maturity gauges



What kinds of security aspects in toolchains?

Looking at data from llvm security group

Toolchain security aspects

OSS software

run-time libraries, most widely used libraries in the world

- memory vulnerabilities
- gadgets
- other vulnerabilities

OSS supply chain security

- Compromised github account
- outdated dependencies

codegen-specific

features helping security of built binaries

- SW-only or HW-specific hardening features
- Sanitizers and other debugging tools
- SBOM generation?

supply chain (malicious codegen)

- backdoor in generated code?

Toolchain security aspects

OSS software

run-time libraries, most widely used libraries in the world

- memory vulnerabilities
- gadgets
- other vulnerabilities

OSS supply chain security

- Compromised github account
- outdated dependencies

codegen-specific

features helping security of built binaries

- SW-only or HW-specific hardening features
- Sanitizers and other debugging tools
- SBOM generation?

supply chain (malicious codegen)

- backdoor in generated code?

Toolchain security aspects

Amongst the most frequent and highest complexity security issues in toolchains.

supply chain (malicious codegen)

- memory vulnerabilities
- gadgets
- other vulnerabilities
- Compromised github account
- outdated dependencies
- **SW-only or HW-specific hardening features**
- Sanitizers and other debugging tools
- SBOM generation?
- backdoor in generated code?

Data from 3 years of LLVM Security Group

- 4x gaps in existing mitigations (e.g. CHOP, CFI, BTI)
- 3x request for new mitigation for vulnerability outside of LLVM (e.g. Retbleed, Ultimate SLH, Trojan Source)

More details on LLVM Security Group stats in other presentation later today

arm

Data from 3 years of LLVM Security Group

- 4x gaps in existing mitigations (
CFI, BTI)
- 3x requ
mitigat
outsid
Retl
Trojan

More details on LL
Security Group stats in
presentation later today

e.g. see https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++

```
-fcf-protection=full
-mbranch-protection=standard
-ftrivial-auto-var-init=zero
-fstack-protector-strong
-D_FORTIFY_SOURCE=3
-fstack-clash-protection
```

…

What are possible root causes of issues related to security hardening?

- Documentation often somewhat under-specifies what a hardening does exactly
  - Results in a few security issue reports by users seeing hardening not applied when they thought it should.
    Implementers of hardening claim it's a "known", deliberate gap.

- Sometimes though simply a bug in the implementation and indeed there is an non-deliberate gap

- Potential causes for non-deliberate gaps:
  - Do compiler engineers creating, adapting or touching hardening implementations know enough about attacks and software security?
  - How can we test correct implementation of hardening?

Help compiler engineers to learn about security "stuff"

https://llsoftsec.github.io/llsoftsecbook/

https://llsoftsec.github.io

Low-Level Software Security for Compiler Developers

Version: 0-176-g9dbdb74

1 Introduction

Compilers, assemblers and similar tools generate all the binary code that processors execute. It is no surprise then that these tools play a major role in security analysis and hardening of relevant binary code.

Often the only practical way to protect all binaries with a particular security hardening method is to have the compiler do it. And, with software security becoming more and more important in recent years, it is no surprise to see an ever increasing variety of security hardening features and mitigations against vulnerabilities implemented in compilers. Indeed, compared to a few decades ago, today's compiler developer is much more likely to implement security features than not.

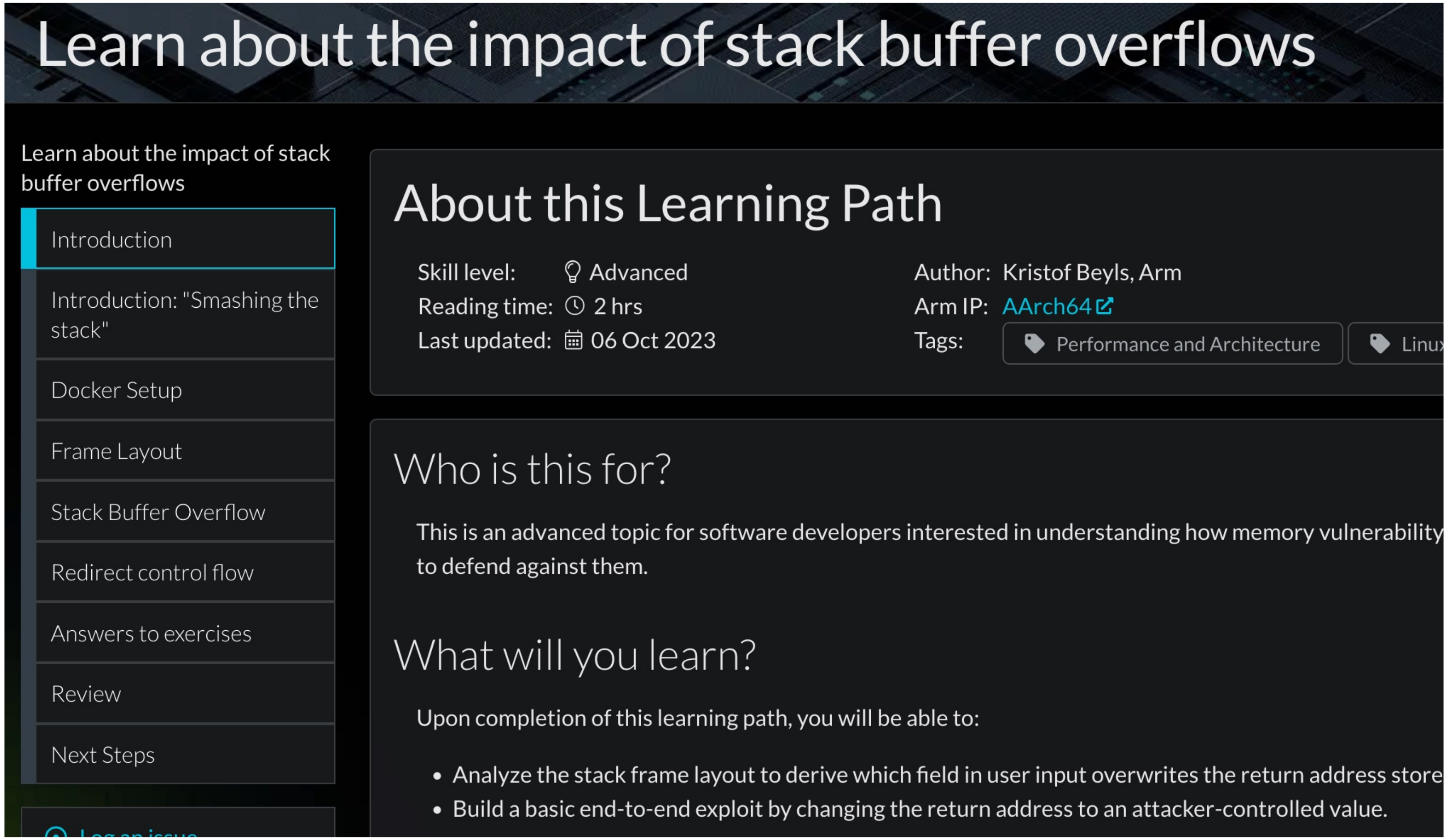
Furthermore, with the ever-expanding range of techniques implemented, it's very hard to gain a basic understanding of all security features implemented in typical compilers.

This poses a practical problem: compiler developers must be able to work on security hardening features, yet it's hard to gain a good, basic understanding of such compiler features.

Learn to think like an attacker, hands-on

https://learn.arm.com/learning-paths/servers-and-cloud-computing/exploiting-stack-buffer-overflow-aarch64/

- Helps to identify weakest spots in a hardening feature
- Start with “smashing the stack for fun and profit” 1996
- Arm Learning Path(s)
- Very hands-on: create a stack buffer overflow attack in less than 2 hours



Testing security hardening implementations

Standard testing practices don’t test hardening well…

1. **Regression and unit tests**, check if generated assembly is exactly as expected…
   … but only for a **very small number of test cases**
2. **Test-suites** cover more code…
   … but only test if program generates expected output
   … **does not test** if program became **more resistant to attack**
3. Sometimes **ad-hoc binary analyzer** gets created
   e.g. x86 stack clash. https://blog.llvm.org/posts/2021-01-05-stack-clash-protection/
   … not widely available, not integrated in CI loops => **no protection against regressions**

- Could we create an **open source binary analyzer** to check for the properties at binary level that should be there?
  - Make **category 2 (test-suite)** useful for testing **effectiveness of hardening features**.

What would a production-quality static binary analyzer enable?

1. Check correctness of hardening features **during implementation**.
2. Add the scanner to compiler CI loops, to detect **regressions**.
3. Integrate in a **fuzzing** setup to verify hardening remains correct with **non-default compiler options**.

Compiler development

1. Hardening feature correctly applied across an entire **distribution**, no matter how binary code was produced.
2. Integrate into a **distribution build process** to verify that there are no **regressions**.
3. For some mitigations, there are few specific contexts where they cannot be applied. Often this is only known to a hand-full of implementers working in this area.
   Use analyzer to enumerate and **document those intended gaps**.

Packaging/distro building

1. Could also use analyzer to check for **other binary properties that do not affect output of generated program**, e.g. are frame pointer chains created correctly?

Could we create such a binary analysis tool?

- **”How hard could it be?”**
  - Let’s build a few prototype binary scanners for AArch64 binaries.
- Start with one relatively easy one: **pac-ret** hardening
  - pointer authentication on return addresses; mitigating ROP attacks
  - Enabled by default on a number of linux distributions
- Then a harder one: **stack-clash**
  - Requires reverse engineering how stack grows, shrinks, gets accessed -> in theory intractable?
  - But maybe in practice, doable?
  - Could give an indication of how hard other stack-related hardening features such as stack canaries might be to scan for?

Pac-ret hardening

a.k.a. “pointer authentication”

Assumed Threat model

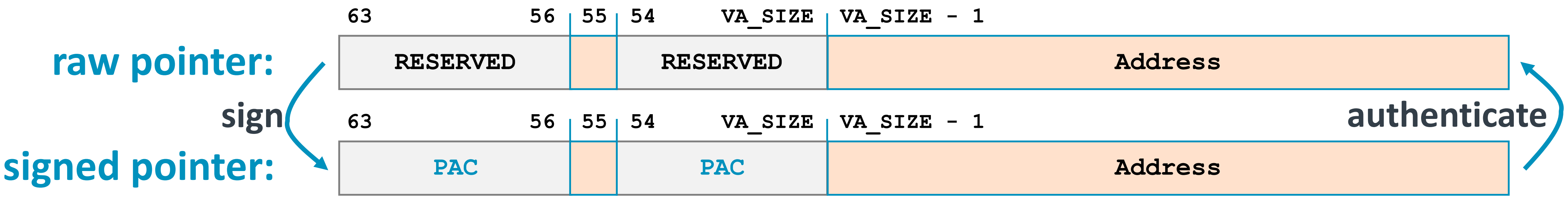
- Attacker uses one or more memory vulnerabilities to **overwrite data memory**.
  - Assumption is code can not (easily) be overwritten, cannot write “new code” to running process.
- Typical attacks then are so-called **code-reuse attacks**:
  - Attacker overwrites a “**code pointer**” in the data memory, e.g. return addresses stored on the stack.
    When code follows such a code pointer, the attacker controls where execution continues.
    By stitching together snippets of code ending in an indirect control flow, attacker can sometimes achieve “turing-complete”/arbitrary code execution.
    - E.g. opening a network port for the attacker to connect to the running process; …
  - **ROP** (return-oriented programming), **JOP** (jump-oriented programming) attacks

Armv8.3: PAuth signed pointers

Detect unintended overwrites of pointer values in memory

- Pointer Authentication aims to make such attacks harder by trying to detect pointer overwrites.

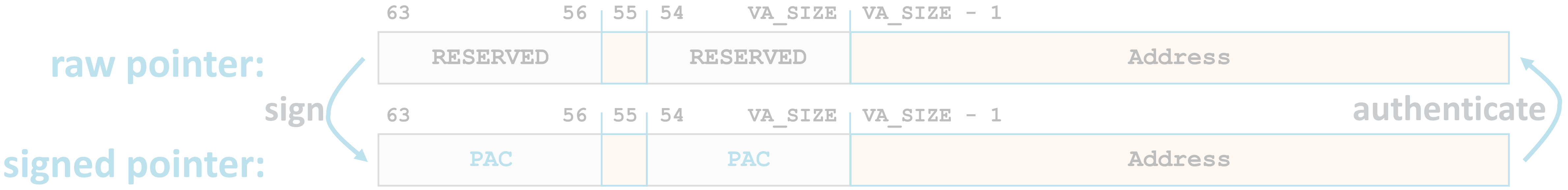
- Use otherwise-unused upper bits in the pointer to store a cryptographic hash (**PAC**).
- Between loading the pointer in a register and using it, **authenticate** the signed pointer



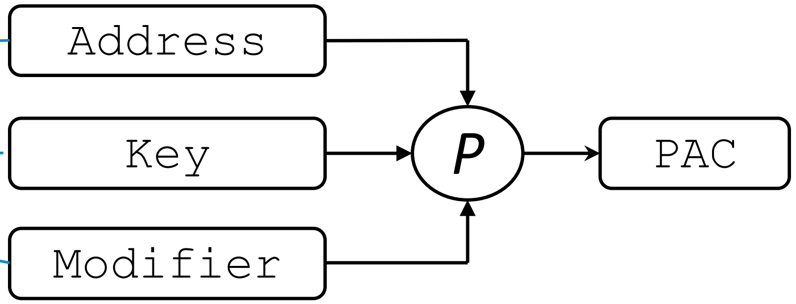
Armv8.3: PAuth signed pointers

Detect unintended overwrites of pointer values in memory

- Use otherwise-unused upper bits in the pointer to store a cryptographic hash (**PAC**).
- Between loading the pointer in a register and using it, **authenticate** the signed pointer



- What input should go into the PAC, so that attackers cannot produce valid signed pointers in memory?
  - The address
  - The attacker should not be able to compute the PAC offline
  - The attacker should not be able to substitute a valid signed code pointer



Typical use of Pauth instructions in pac-ret hardening

```
bl f // sets x30 to point to next_instruction
next_instruction
```

Typical use of Pauth instructions in pac-ret hardening

```
bl f // sets x30 to point to next_instruction
next_instruction
```

```
f:

    stp x29, x30, [sp, #-16]! // return address stored to memory

    bl function_processing_attacker_controlled_data

    ldp x29, x30, [sp], #16 // attacker-controlled return address

    ret x30 // Instead of returning to next_instruction, attacker
            // takes over control
```


Typical use of Pauth instructions in pac-ret hardening

```
bl f // sets x30 to point to next_instruction
next_instruction
```

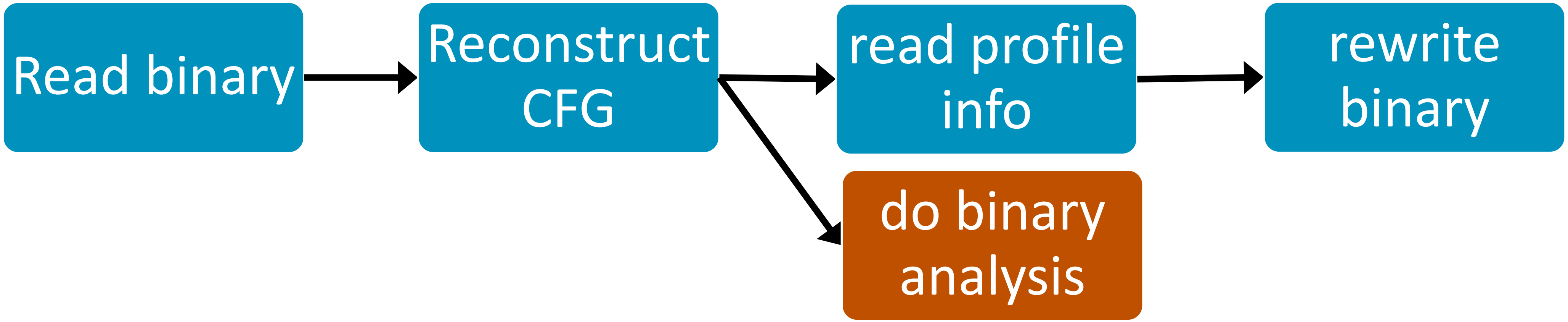
```
f:
    paciasp // PAC IA SP (x30)
    stp x29, x30, [sp, #-16]! // return address stored to memory

    bl function_processing_attacker_controlled_data

    ldp x29, x30, [sp], #16 // attacker-controlled return address
    autiasp // AUT IA SP (x30). Detects if x30 was tampered with.
    ret x30 // Instead of returning to next_instruction, attacker
            // takes over control
```

What is the “binary property” to check for pac-ret hardening?

- Goal: avoid checking specific compiler implementation
  So: what is the bare minimum invariant to check?
- I came up with:
  - When you have a return instruction (e.g. RET x30)
  - The register with the address to return to (e.g. x30)
  - Should either:
    1. not be written to in the function
    2. Or last be written to be an authenticating instruction.

Llvm-bolt-gadget-scanner

Why build a binary scanner in BOLT?



1. Works at the MCInst layer, i.e. exactly mirror what is in the binary, no loss of accuracy.
2. Familiarity for LLVM developers: can implement both a mitigation and the associated analyzer in the same framework.
3. Actively used by large organizations to achieve great benefits; framework most likely will be maintained for a long time.
4. Development cost for lifting binary to CFG can be shared between optimization and analysis use cases.

Implementation and evaluation strategy for a prototype

- `/usr/lib64` Fedora 39: 1981 libraries; 261M instructions.
- Iteratively:
  - Fix issues with BOLT unable to read binaries
  - Investigate root cause for reported pac-ret issues; fix implementation if false positive
- Making use of BOLT's built-in dataflow analysis
- What kind of issues with BOLT unable to read binaries?
  - Avoid crashing on unrecognized jump table sequence.
  - DWARF OpNegateRAState not supported (issue #74833)
  - Not being able to reconstruct CFG for many functions (23%)
  - Therefore, also implemented scanner for when CFG isn't reconstructed

Results from experiment

- Total analysis time is 667s on a single core => 391K instr/s. More than fast enough.
- Number of lines of code to implement/complexity?
  - Pac-ret-specific gadget scanning: O(700 lines)
  - Kloc for general “new tool based on BOLT”: O(400 lines)

Pac-ret “gadgets” found

- Total 2.5M returns.
  Pacret gadgets: 46K.
  About 1.8% of returns not protected.
- Why are there non-protected returns when pac-ret is enabled Fedora-wide?
  - True positives:
    1. Some libraries written in languages for which compilers do not yet support pac-ret hardening, e.g. Rust, Haskell, Go, …
    2. One or a few C/C++ libraries have quirks in their build system, meaning distro-wide default does not propagate through.
    3. A few in assembly-written code doing “special stuff” and ”known gap” by implementers.
  - False positives:
    1. analysis not yet aware that BRK instructions end execution flow

```
doesnotreturn:
        brk 1

f_call_noreturn:
        bl doesnotreturn
        ret
```


Conclusion on experiment building scanner for pac-ret

- Implementation and tool running cost very reasonable.
- Results from diagnostics are actionable and useful:
    1. Prioritize which toolchains for which language to implement pac-ret support in based on data.
    2. Fix build system for packages not respecting distro-wide default.
    3. Document accepted gaps in hardening, so knowledge becomes accessible.
- Some general remaining work left on:
    - enabling BOLT to reverse engineer CFG on more functions
    - recognizing more “no-return” functions
    - recognizing more jump table binary patterns

Stack-clash

Stack-clash attack: sketch of how it works

```
long f(int N) {
  long A[N];
  g(A, N);
  return A[N-1];
}
```

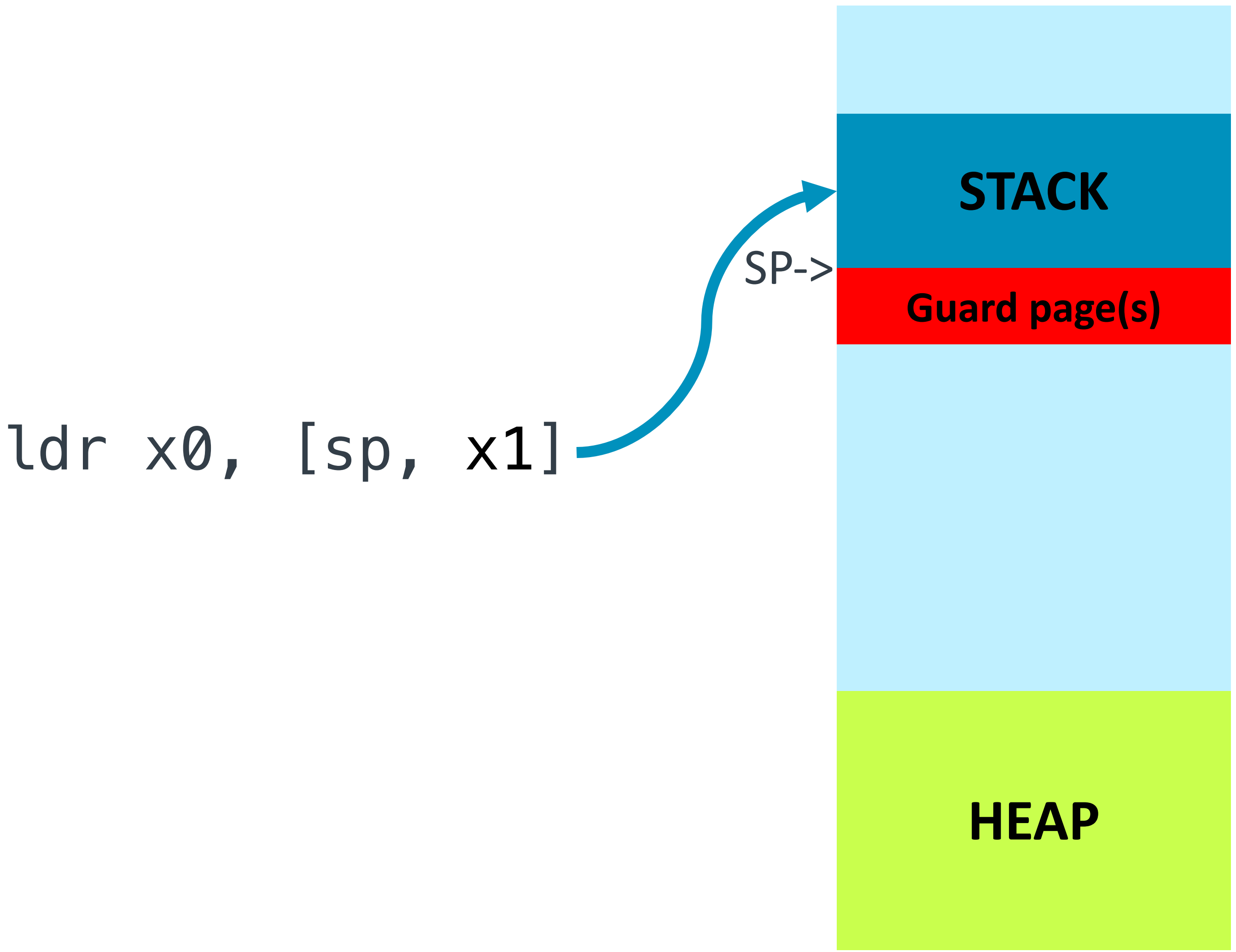

Stack-clash attack: sketch of how it works

```
long f(int N) {
  long A[N];
  g(A, N);
  return A[N-1];
}
```

```
ldr x0, [sp, x1]
```


Stack-clash attack: sketch of how it works

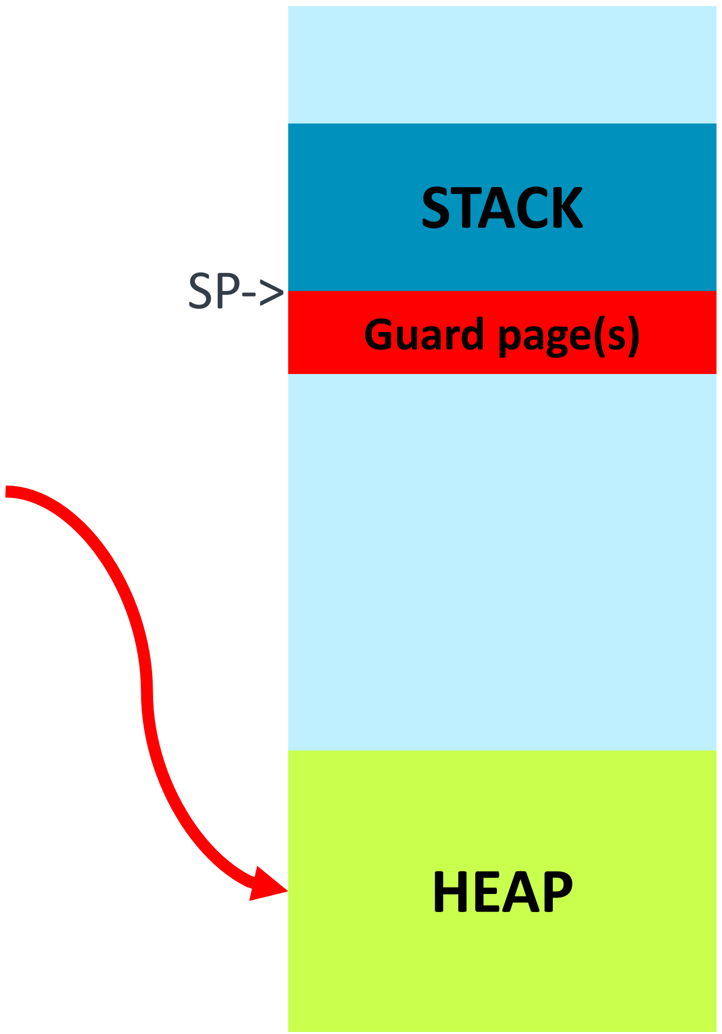
```
long f(int N) {
  long A[N];
  g(A, N);
  return A[N-1];
}
```



Stack-clash attack: sketch of how it works

```
long f(int N) {
  long A[N];
  g(A, N);
  return A[N-1];
}
```

```
ldr x0, [sp, x1]
```



What does stack clash protection aim to achieve?

1. Only grow stack at most one page at a time,
2. and do at least one memory access on every new page as it grows.
   … to ensure when the stack grows, there's always an access to the guard page

- See https://blog.llvm.org/posts/2021-01-05-stack-clash-protection/ for more information.

- A gadget scanner will need to keep track of stack pointer changes and stack accesses. Is that even tracktable?

Stack pointer evolution tracking: gcc stack protector loop

```
 sub x2, sp, x2
 cmp sp, x2
 beq .L3
```

```
.L7:
   sub sp, sp, #65536
   str xzr, [sp, 1024]
   cmp sp, x2
   bne .L7
```

```
.L3:
   and x1, x1, 65535
   sub sp, sp, x1
   str xzr, [sp]
```

1. Need to track known maximum values of registers

Stack pointer evolution tracking: spilled stack pointer value

```
f_spoffset_spilled:
  stp x29, x30, [sp, #-16]!
  mov x29, sp
  sub sp, sp, #16
  mov x0, sp
  str x0, [x29, #8]
  prfm pstl1keep, [x29, #0x0]
  ldr x1, [x29, #8]
  mov sp, x1
  mov sp, x29
  ldp x29, x30, [sp], #16
  ret
```

2. Need to track which registers have the same value as the stack pointer+offset

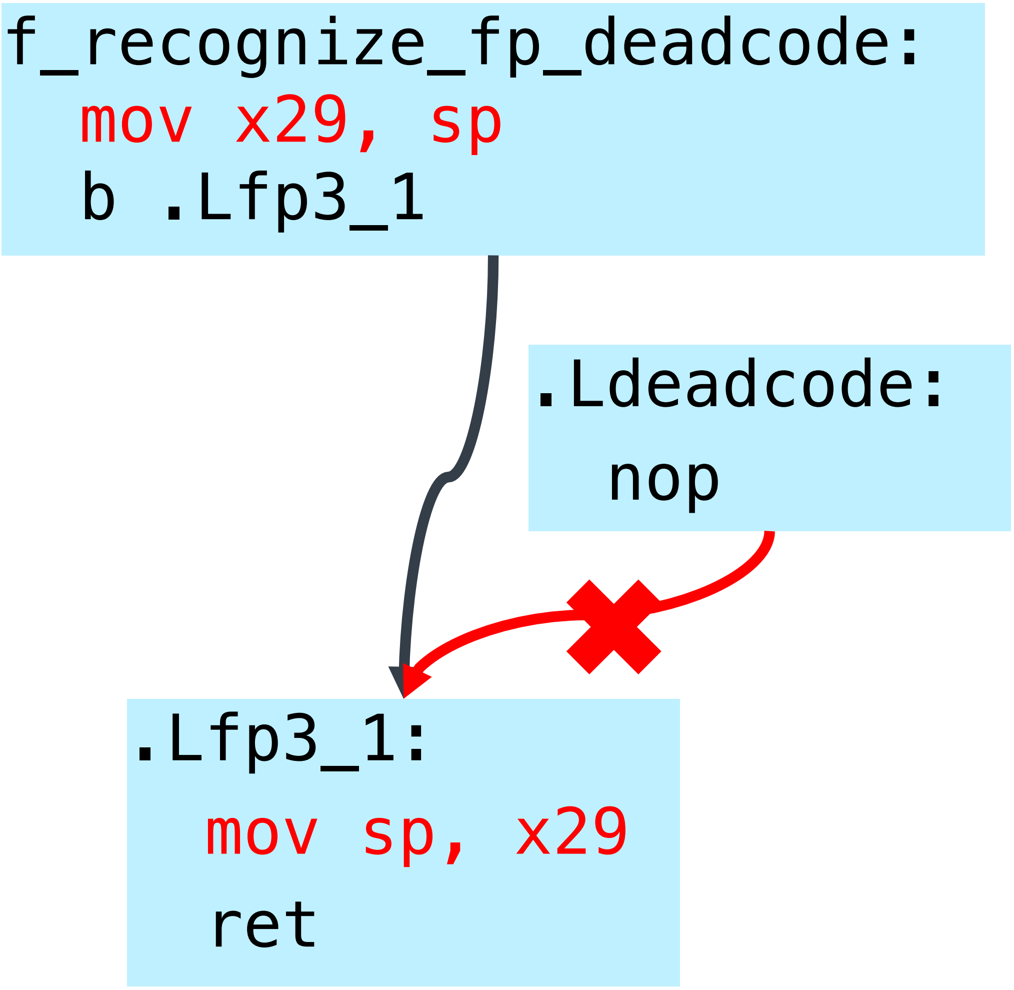
3. Need to track spill/fill of such registers

Stack pointer evolution tracking: constant values in registers

```
mov x12, #40000
sub sp, sp, x12
```

4. Need to track which registers contain a constant value

Stack pointer evolution tracking: dead binary code



5. Need to recognize dead basic blocks and no flow is possible from them

6. Need to recognize no-return functions

Stack pointer evolution tracking: aligning stack pointer

```
sub x9, sp, #0x1d0
and sp, x9, #0xffffffffffffff80
```

7. Need to recognize masking on sp-offset values

Prototype implementation experience

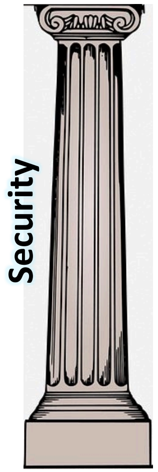
- Also implemented using dataflow framework.
- Like for the pac-ret scanner, iteratively:
  - Investigate root cause for reported stack-clash gadget
  - if false positive: improve pattern recognizer
  - That's how the stack change patterns in previous slides were recognized and implemented
- Ongoing work, current state: still stack clash gadgets reported in 39 out of 1920 libs.
  - Presumably most remaining ones are still false positives and a few more stack manipulation patterns need recognizing?
- Avg analysis speed 391K instr/s. More than fast enough.
- Core dataflow implementation O(1000) lines
  - O(1000) lines for improving tablegen to enable querying offset and size of memory access for all LD/ST instructions.

Stack-clash "gadgets" found

- Total 1920 libs, about 2M functions.
- Still stack clash gadgets identified in 39 out of 1920 libs.
- Smaller experiment on LLVM test-suite rather than /usr/lib64:
  - Build it with gcc, both with and without `-fstack-clash-protection`
  - LLVM test-suite built with gcc: 101 stack-clash gadgets reported.
  - LLVM test-suite built with gcc: 1 stack-clash gadget reported (not yet clear if true or false positive).
- Conclusion:
  - Bringing false positive rate down far enough seems feasible,
    requires some more iterating on analyzing false positives and improving pattern recognizer.

Summary

Summary



- Security is becoming the third pillar of compiler design and implementation, next to correctness and optimization.

- Security hardening features are regularly added to compilers.
  - Ability to test their implementation is limited
  - A significant number of reported security issues relate to security hardening features.

- Is a binary analysis tool that checks correct hardening across a binary feasible?
  - Reusing BOLT, as that already has binary analysis capabilities.
    Win-win with optimization use case.
  - Prototype implementation shows its absolutely doable for pac-ret, most likely doable for stack-clash.
- Conclusion: yes, it seems worthwhile to implement such a binary scanner in BOLT.

Summary (2): llvm-bolt-gadget-scanner

- Llvm-bolt-gadget-scanner would be useful to:
  - Better test correct implementation of security hardening in compiler
    - During development; integrated in CI; integrated in fuzz testing
  - Better check proper application across a whole distribution
  - Could be useful for checking other binary properties too (e.g. correct frame chain creation, …)
- Prototype implementation available at https://github.com/kbeyls/llvm-project/tree/bolt-gadget-scanner-prototype
- Cannot turn prototype into a quality upstream implementation fully on my own.
  - Please reach out if you think this is interesting. Even more so if you could provide help ☺
  - Round table later this conference.
  - RFC: https://discourse.llvm.org/t/rfc-bolt-based-binary-analysis-tool-to-verify-correctness-of-security-hardening/78148

arm

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה
ధన్యవాదములు