Guarded Control Stack (FEAT_GCS) for Debian

Steve Capper steve.capper@arm.com

Debian Miniconf – 2024-10-12

Introduction

- I’ll be talking about a new Arm architectural feature, Guarded Control Stack (FEAT_GCS),
- FEAT_GCS is OPTIONAL from Armv9.3,
- The target Debian distro for this will be Forky (not Trixie! ☺),
- We’ll start with a brief intro behind the architecture and then move on to how we’d like to enable this in the distros,
- Support for FEAT_GCS is still going through the upstream process, so there may be small variations in what the deployed solution looks like (f.e. names of flags/etc),
- Content warning: this talk does contain some assembler.

Quick intro to the architecture

Let’s dive in!

Let’s take a look at a motivating example: x29 = frame pointer, x30 = link register

```
#include <stdio.h>

void world(void)
{
        printf("world!");
}

void hello(void)
{
        printf("Hello ");
        world();
}

int main(void)
{
        hello();
        printf("\n");
        return 0;
}
```

```
world:
        stp     x29, x30, [sp, -16]!
        mov     x29, sp
        adrp    x0, .string_world
        add     x0, x0, :lo12:.string_world
        bl      printf
        ldp     x29, x30, [sp], 16
        ret
hello:
        stp     x29, x30, [sp, -16]!
        mov     x29, sp
        adrp    x0, .string_hello
        add     x0, x0, :lo12:.string_hello
        bl      printf
        bl      world
        ldp     x29, x30, [sp], 16
        ret
main:
        stp     x29, x30, [sp, -16]!
        mov     x29, sp
        bl      hello
        mov     w0, 10
        bl      putchar
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
```

Two big takeaways from the assembler example

1. The control flow could be influenced by modifications to the stack,
   - I mentioned PAC + BTI in my previous mini-conf talk which help mitigate against return oriented and jump oriented programming style attacks.
2. It is fiddly to unwind the call frames, and it becomes a lot fiddlier with custom assembler and different runtimes being involved
   - I've omitted the Call Frame Information (CFI) directives,
   - (Try a sneaky `gcc -s hello.c` to reveal more details),
   - Profiling tools (f.e. `perf record`) have to do a lot of call stack unwinding.

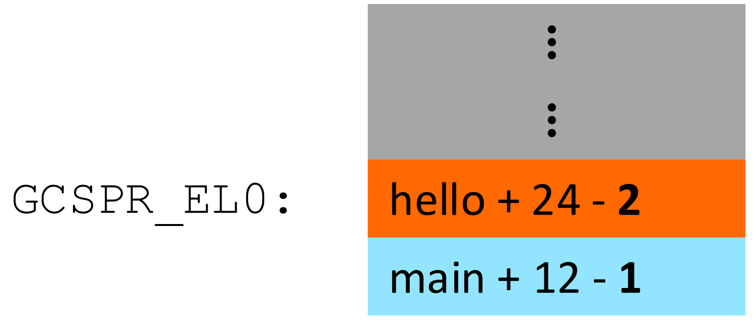
+ With FEAT_GCS, the return addresses are placed in their own area of memory (in addition to being on the stack) where they can be parsed quickly but not be modified maliciously/accidentally.

Our example running under FEAT_GCS

Big takeaway from this slide: no instructions have changed!

```
world:
        stp     x29, x30, [sp, -16]!          Break
        mov     x29, sp
        adrp    x0, .string_world
        add     x0, x0, :lo12:.string_world
        bl      printf
        ldp     x29, x30, [sp], 16
        ret
hello:
        stp     x29, x30, [sp, -16]!
        mov     x29, sp
        adrp    x0, .string_hello
        add     x0, x0, :lo12:.string_hello
        bl      printf
        bl      world
        ldp     x29, x30, [sp], 16         2
        ret
main:
        stp     x29, x30, [sp, -16]!
        mov     x29, sp
        bl      hello
        mov     w0, 10                     1
        bl      putchar
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
```

- Let's imagine we have set a breakpoint in `world` and started our program,
- Program execution starts at `main`, which then calls `hello`, which then eventually calls `world`.
- Our guarded control stack is a table of return addresses which looks like this when we hit the breakpoint:



So how did our example work?

- FEAT_GCS, when activated, subtly changes how some instructions work,
- In our example the branch with link (`bl`) instruction would additionally “push” the return address onto the guarded control stack memory space,
- A return (`ret`) instruction would compare the contents of the guarded control stack with the return address and, if equal, “pop” from the guarded control stack and return.
- The guarded control stack (pointed to by `GCSPR_EL0`) can be read as normal by userspace, but normally can’t be written to explicitly.

More formal (and correct!) terminology can be found in the Arm Architecture Reference Manual:

https://developer.arm.com/documentation/ddi0487/ka/?lang=en

(Chapter D11 The Guarded Control Stack)

Getting FEAT_GCS into a distro
“How do I make it go?”

A subtle change to deployment strategy for FEAT_GCS

- With Pointer Authentication (PAC), we "baked in" prologue/epilogue code sequences into functions to sign and authenticate return addresses.
  - The user could disable PAC globally via `arm64.nopauth` on the kernel command line,
- For Branch Target Identifiers (BTI), we placed BTI instructions into functions, where needed, and annotated compatible binaries – custom assembler without annotations would result in the final executable not advertising BTI.
  - The user could disable BTI globally via `arm64.nobti` on the kernel command line.
- With Guarded Control Stack (GCS) most userspace code is unchanged. Binaries are annotated and again custom assembler needs to be reviewed and manually annotated for an executable to be advertised as GCS compatible.
- Big change: **The end user needs to opt-in per-process for GCS via glibc tunable**.
- This means we can phase in GCS support as a low risk feature.

Running a program with FEAT_GCS

(Patches still under review, naming of these tunables may change)

- To run with GCS enabled, one sets the tunable as follows:

```
GLIBC_TUNABLES=glibc.cpu.aarch64_gcs=1:glibc.cpu.a
arch64_gcs_policy=2 ./hello
```

- This will cause the runtime loader to then call the relevant kernel APIs to enable "shadow stacks" (borrowing the x86 terminology),

```
prctl(PR_SET_SHADOW_STACK_STATUS,
        PR_SHADOW_STACK_ENABLE, 0, 0, 0);
```

- Any GCS errors would be picked up as SIGSEGV with an si_code of SEGV_CPERR (control protection error).

Explanation of the glibc tunables:

`glibc.cpu.aarch64_gcs=<x>`

0	GCS disabled (default)
1	GCS enabled depending on policy selected below:

`glibc.cpu.aarch64_gcs_policy=<x>`

0	GCS enabled regardless (default)
1	GCS enabled where executable is marked as compatible, any later `dlopen` to an incompatible binary is an error
2	GCS enabled, but any incompatible binary is an error

Getting FEAT_GCS into a test distro

- At the moment GCS is still under review upstream,
- Whilst various upstreams were discussing overall design we have staged a Yocto Project GCS layer that allows one to build and run a small test distro on an arm64 or x86 host.
- Quick start steps can be found here:
  https://git.yoctoproject.org/meta-arm/tree/meta-arm-gcs/README?h=testing/gcs
  (I used the above to test the code sequences in this talk)
- The Fixed Virtual Platform – "Armv-A Base RevC AEM", is used:
  https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms

Getting FEAT_GCS into Debian

- Once the binutils and gcc patches are upstream and merged into Debian Unstable…
- To annotate binaries as being GCS aware, one needs to employ the following CFLAG:

```
-mbranch-protection=standard
```

- (This CFLAG is already used for PAC + BTI deployment in Debian Trixie),
- One can check existing binaries via:

```
$ readelf -n ./hello | grep AArch64
      Properties: AArch64 feature: BTI, PAC, GCS
```

- Then it is a case of hunting for stragglers, most of which will likely be due to hand-coded assembler being present, the following LDFLAG can be employed to aid diagnostics:

```
-z experimental-gcs=always -z experimental-gcs-report=error
```

- (flag naming may change as patches are still under review)

I need to write GCS aware code; how do I detect it?

- For any manual manipulation of the stack (f.e. for implementing something like `longjmp/setjmp`), code needs to detect whether or not GCS is enabled,
- One can use the Linux kernel API:
  ```
  prctl(PR_GET_SHADOW_STACK_STATUS, &status, 0,
  0, 0);
  ```
- There may be circumstances where one is unable to make a syscall, in which case, we have an instruction to aid with detection of GCS,
  ```
       CHKFEAT x16
  ```

- From the Arm Architecture Reference Manual: (D1.10 Check Feature)

```
MOV X16, #0x1 ; X16 has bit [0] set to select GCS
CHKFEAT X16   ; Updates X16 with the status of GCS

; Skip over GCS code if GCS is not enabled
TBNZ X16, #0, skipgcs
...           ; GCS related code
skipgcs:
              ; no more GCS code
```

- `CHKFEAT` above runs as a NOP on cores that don't implement FEAT_GCS (or where GCS hasn't been enabled),
- so can be used safely in general purpose code.

Enabling GCS in assembler

The first rule of assembler is… not to use assembler!

ABI still under review!

- Advertising GCS compliance in assembler is done in a very similar way employed for BTI, we have an extra flag to set in the NOTE section:

```
.pushsection .note.gnu.property, "a"
.balign 8
.long 4                 /* size of the name - "GNU\0" */
.long 0x10              /* size of descriptor */
.long 0x5               /* NT_GNU_PROPERTY_TYPE_0 */
.asciz "GNU"
.long 0xc0000000        /* pr_type - GNU_PROPERTY_AARCH64_FEATURE_1_AND */
.long 4                 /* pr_datasz - 4 bytes */
.long 7                 /* pr_data - GNU_PROPERTY_AARCH64_FEATURE_1_BTI | GNU_PROPERTY_AARCH64_FEATURE_1_PAC
                                     | GNU_PROPERTY_AARCH64_FEATURE_1_GCS */
.long 0                 /* pr_padding - bring everything to 8 byte alignment */
.popsection
```

- The GCS flag is documented here: https://github.com/ARM-software/abi-aa/blob/853286c7ab66048e4b819682ce17f567b77a0291/sysvabi64/sysvabi64.rst#process-gnu-property-aarch64-feature-1-gcs

Current status and enablement strategy

- GCS support for upstream is under way for:
  - Linux kernel, binutils, GCC, Glibc, gdb
- GCS support is available already in LLVM-18,
- Our enablement strategy for Forky is to:
  1. Put up a GCS page on the Debian wiki,
  2. Once gcc gets upstream support and makes its way into Debian…
     any packages built in Debian using `–mbranch–protection=standard` (which is what we're doing already for PAC + BTI), will start also annotating binaries with GCS support,
     (This is the bit that will take most time as it's a rebuild everything job)
  3. Once glibc has GCS support, it should naturally translate over,
  4. Once the Linux kernel gets GCS support, we'll check that it's enabled in Kconfig
     - additional work may be required to ensure that GCS works with uprobes
  5. We can then hunt down stragglers and test incrementally
     - As the feature is opt-in, we do not believe that incremental enablement poses a risk to users.

Thank you for your attention!
Any Questions/Comments?

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

References

- Shadow stacks for 64-bit Arm systems
  - https://lwn.net/Articles/940403/
- Procedure Call Standard for the Arm 64-bit Architecture(AAPCS64)
  - https://github.com/ARM-software/abi-aa/releases
- The Arm Architecture Reference Manual (Chapter D11 The Guarded Control Stack)
  - https://developer.arm.com/documentation/ddi0487/ka/?lang=en
- What is new in LLVM 18? (part 2) – discusses clang + gcs
  - https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/p2-whats-new-in-llvm-18
- Arm A-Profile Architecture Developments 2022
  - https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2022

Gratuitous Recruitment Spam

Apologies, nothing for Debian directly this time.

- We are recruiting software and hardware engineers at Arm, for roles including:
  - Linux kernel,
  - Firmware,
  - Performance optimisation and profiling,
- Many of the software roles are geared towards Open Source Software development,
- Should anyone here be interested (or know anyone who may be interested); please do get in touch with me: steve.capper@arm.com
- There is a careers page which I can help folk navigate too:
  - https://careers.arm.com
- **I would be more than happy to answer any recruitment queries.**