Stack unwinding in AArch64 processors: what is it and how it works

Nikita Popov

KernelCare
Stack unwinding in AArch64 processors:
What is it & How it works
AARCH 64

For the past nine months, KernelCare's Linux kernel live patching software has supported ARMv8 (AArch64) in addition to x86_64 (Intel IA32/AMD AMD64). To get KernelCare running on Arm, we needed a *stack frame unwinder*.

This article explains what they are, what they're used for, and why we had to write our own.

1
Stack Unwinders
What are they, What are they used for & history
2
How stack unwinding works?
Stack primer, How it works in AArch64 processors
3
Instructions
Instruction to jump, address to jump to & troubleshooting

Part A - Stack Unwinders

What is a stack unwinder?

A stack unwinder is a software that lists the addresses of every function currently in the calling stack. It shows you where you are in a program's execution, but more importantly, how you got there.

An effective unwinder must have all of these characteristics:

- **Fast**: so that processing can quickly resume (if possible).

- **Cheap**: so that it doesn’t drain system resources.
- **Accurate**: so that memory addresses and name-spaces are accurately reported.

What is a stack unwinder used for?

1. To provide stack traces when a program crashes. (In the Linux kernel world, such crashes are called ‘Oops’.)
2. To help with performance analysis, showing the route (which functions called which) a program takes within a program.
3. To enable Linux kernel live patching, the act of fixing kernel bugs without stopping (rebooting) the system.

History of stack unwinding

Historically, stack unwinding came to suit the needs to debug software. Whoever has written at least a few programs knows that programs may contain errors.

Some errors are easy to spot while other errors go almost unnoticeable. The bigger the program is the more difficult it becomes to debug such program: really huge programs

are almost impossible to debug by using only source code analysis as sole debugging technique. So, there were proposed several secondary techniques which aim to facilitate the debugging process.

These techniques are:

1. **Logging** (i.e. debugging output) -- programmer uses printing operators of his language of choice to display contents of specific variables at specific points in program flow. This allows to spot

where have the program flow diverged from the expected scenario. Modern software vastly relies on logging but uses different logging levels - debug, notice, warning, error and so on - to be able to disable some less important logging messages in production. Typically, logging messages go to dedicated logfile.

2. **Debugging via breakpoints.** In order to facilitate the debugging process, almost every modern CPU architecture includes special instruction called "breakpoint" instruction (for instance bkpt for arm, int3 for x86). The purpose of this instruction is to cause a special processor interrupt upon executing this instruction. The hardware then saves the current program counter and may save a few general-purpose registers in order to help interrupt service routine to successfully start. Control is then transferred to such interrupt service routine.

   What this routine does is to extract saved contents of general-purpose registers to help a programmer examine certain program variables at the current point where the program has been stopped. Just before executing special interrupt service routine return instruction this interrupt handler typically restores original instruction and after returning to the interrupted thread, the program runs as if it has never been touched at all.

   Modern-day usage of this technique relies on support from the operating system kernel since all interrupt service routines are part of it. Usually, OS kernel provides special system calls (for instance, ptrace on Linux) that userspace processes may use to perform debugging functions. Popular Linux open-source debugger, named GDB, uses ptrace to perform step-by-step (via breakpoints) debugging.

Also, some CPU architectures (x86-64 for instance) define additional capabilities which are based on the original idea. For example, there may be special system registers containing the address of (virtual) breakpoint: when the program counter hits this address, interrupt service routine is called immediately without any need to patch (and consequently restore) program code.

3. **Debugging via assertions.** The assertion is a special function which checks the given condition and if this condition is false, causes the program's termination. Assertions can be used by programmers to ensure that internal variables are sane. Assertions are typically disabled in production programs. Here, the most interesting case is when the assertion comes false. This state clearly signals that some program error took place. To help investigate a problem stack unwinding is performed. By unwinding the program's stack we can obtain "execution path" which lead our program to that point where it crashed.

Advanced unwinders allow seeing parameters for each function within the call chain.

So, stack unwinding originally came from software debugging. Now however it has a lot of applications one of which is in Kernelcare.

Why KernelCare teem needed an unwinder for Arm

To install a Linux kernel live patch, the patching software must know what functions are in the current calling stack. If a function currently in the calling stack is patched, the system can crash when returning. There is some stack unwinding functionality already in the Linux kernel. Here is a brief review of them and the reasons why they can’t be used for live patching.

- The ‘Guess’ unwinder: It guesses the contents of the stack. It is not accurate and so not useful for live patching. It is only available for x86_64 architectures.
- The ‘frame pointer’ unwinder: Available only for x86_64.
- The ‘ORC’ unwinder: Introduced in Linux kernel v4.14, it is only available for x86_64, and there are no stack reliability checks, making it unsuitable for live patching.

Part B - How stack unwinding works

Stack Primer

- When a function is called, a stack frame keeps track of the function’s arguments, and its entry and exit points.
- A processor register is assigned a stack point (‘sp’) that references the object most recently put on the stack. The memory implementing that stack expands downwards towards lower memory addresses (so-called ‘full-descending’).
- Stack memory must be aligned to byte boundaries (16 bytes for AArch64). This is enforced by the hardware (but can be deactivated on some Arm models).

Details

How stack unwinding works in AArch64 processors?

A special kernel function performs stack unwinding. When called, the function gets the frame pointer (FP) of the calling function. This FP refers to the stack frame which is represented by the struct stack_frame structure. It contains a pointer to the stack frame of the function that has called the calling function.

This means we have a linked list of stack frames that ends when the next obtained FP equals 0 according to AAPCS64 (the procedure call standard for AArch64). In each stack frame, we can retrieve a return address where the calling function should delegate control after it finishes its work. Using the fact that a return address should point out inside the calling function, we can get the symbolic names of all functions, up to and including the point where FP=0.

To do so, we keep the names of the functions and their begin and end addresses. This can be implemented using the Linux kernel subsystem called kallsyms.

The core problem can be boiled down to this: how do we move the program counter from one place to another (a jump) and resume processing without problems?

This action can be expressed in assembly language as:

- BL (branch with link)
- RET (unconditional return)

Let's look at these in more detail.

1. Instruction to jump

Procedures are invoked with BL. The 32-bit instruction can be visualized as follows:

```
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |
```

| | | | | | | | | | | | |

31 30 29 28 27 26 25 24 23 22 21 20 19 18

17 16 15 14 13 12 11 10 09 08 07 06 05 04

03 02 01 00
| 1 | 0 0 1 0 1 | imm26

|

Here, imm26 is a 26-bit PC offset.

2. address to jump to

We calculate where to jump to using:

bits(64) offset = SignExtend(imm26:'00', 64)

The offset shifts by two bits to the left and converts to 64 bit (i.e. the high bits fill with 1 if imm26 < 0, and with 0, otherwise).

The address to jump to is then:

Address = PC + offset

The offset is labeled and used in the BL instruction. (Instructions in AArch64 always take 4 bytes, which is an advantage compared to x86.) The register X30 (also known as the Link Register) is set to PC+4. This is the return address for RET (defaults to X30 if not specified).

So, for the complementary instruction RET, it is enough to retrieve the saved LR value and transfer control onto it and return into the calling function.

The problem: What happens if the called function calls another function itself?

And here we have a problem: what happens if the called function calls another function itself? If we do nothing, then the value saved in LR will be replaced with a new return address—it will not be able to return to the initial function and the program will most likely abort.

Solving the problem

There are some ways to solve this problem:

1. Save the LR value into some other register.
2. Save the LR value into RAM.

The first case is very restrictive, as the number of available registers is limited (to 31 registers). Arm uses the RISC architecture load/store architecture philosophy which says that memory calls are done via LD (LOAD) and ST (STORE) instructions, whereas arithmetical and logical operations are performed on registers. Therefore, empty registers are needed for program execution, and we are left with option 2, save the LR value into RAM.

Saving called functions to return addresses in the stack

A frame structure implemented in C looks like this:

struct stack_frame {

```
    unsigned long fp;
    unsigned long lr;
    char data[0];
};
```

In other words, each function allocates n bytes in the stack, reducing by n the stack pointer at the moment when it takes control with the BL instruction. The contents of registers x29 (FP) and x30 (LR) are saved according to the obtained stack pointer value — the calling function used by these values. After that, the new value SP is assigned to the register x29, called the frame pointer (FP). The remaining space in the stack frame is used by the function local variables. And the condition that the frame pointer (FP) and link register (LR) of the calling function are always located at the beginning of any stack frame, is always met. After finishing its work, the called function takes the saved values of FP and LR from the stack frame and increases the stack pointer (SP) by n.

How gcc cross-compiler deals with AArch64

Using the frame pointer requires the Linux kernel to be compiled with the --fno-omit-frame-pointer gcc option. This option tells gcc to store the stack frame pointer in a register. (NOTE: The default for gcc is --fomit-frame-pointer, so this option must be explicitly set.)

For AArch64, the register is X29. This is reserved for the stack frame pointer when the option is set. (Otherwise, it can be used for other purposes.) The cross-compiler GCC used to compile Linux under AArch64 sets the following instructions before the function body:

```
ffffff80080851b8 <arch_align_stack>:
```

```
ffffff80080851b8: a9be7bfd stp x29, x30, [sp, #-32]!
ffffff80080851bc: 910003fd mov x29, sp
```

Here, so-called indirect addressing with pre-increment where the stack pointer (SP) is decreased by 32 at the beginning and then x29, x30 are sequentially saved in the memory by the value obtained in the first instruction.

Usually, the function finishes as follows:

```
ffffff80080851fc: a8c27bfd ldp x29, x30, [sp], #32
ffffff8008085200: d65f03c0 ret
```

The indirect addressing with the post-increment where the saved values x29, x30, are taken from the memory on the stack pointer (SP) and then SP increases by 32. The code examples above are called the prologue and epilogue of the function respectively. GCC always generates such prologues and epilogues if the flag -fno-omit-frame-pointer is set. Linux on AArch64 is compiled with that flag so that stack frames look like regular code (except assembly code). This fact allows us to easily unwind the stack, i.e. track the call chain in the program.

Assembly Reference:

- STP (store pair of registers)
- MOV (reg to/from SP)
- LDP (load pair of registers)

Conclusion

Despite its usefulness, there is no common approach to stack unwinding that covers all architectures and systems. For Linux kernel live patching, a reliable and quick stack unwinder is essential. For Linux running on Arm, the need for a robust stack unwinding solution is even more pressing, as Arm gains traction in the IoT device and edge-cloud computing markets. With KernelCare pushing into both, we had to look into our own solutions for kernel stack unwinding.

Additional Reading

- "Using the Stack in AArch64: Implementing Push and Pop" – Jacob Bramley, November 2015 @ Arm Developer Community Processor Blog
- "Using the Stack in AArch32 and AArch64" – Jacob Bramley, November 2015 @ Arm Developer Community Processor Blog
- "The Linux x86 ORC Stack Unwinder" – Matt Fleming, July 2017 @ Code Blueprint
- "ORC unwinder" – Josh Poimboef, July 2017 @ LWN.net
- Linux kernel stack validation
- Linux kernel livepatch – A common consistency model for live patching.

About KernelCare

KernelCare makes patching your Linux kernels simple for servers on CentOS, Amazon Linux, RHEL, Ubuntu, Debian, and other Linux distributions, including the Yokto and Raspbian.

KernelCare maintains kernel security with automated, rebootless updates without any service interruption or degradation. The service promptly delivers the latest security patches for different Linux distributions applied automatically to the running kernel in just nanoseconds. The service works in both, live and staging environments, and for servers located behind the firewall, there is an ePortal to help you manage it.

KernelCare enhances compliance on over 300 thousand servers of various companies where the service availability and data protection are the most crucial parts of the business: financial and insurance services, video conferencing solution providers, companies protecting domestic abuse victims, hosting companies, and public service providers.

Flexible pricing for different server and devices fleets, a free trial for everyone, and bespoke Proof of Concepts for clients with custom infrastructures. Get your free 7-day trial here.