Google

Compiling Android userspace and Linux Kernel with LLVM

Nick Desaulniers, Greg Hackmann, and Stephen Hines*

October 18, 2017

*This was/is a really **HUGE** effort by many other people/teams/companies. We are just the messengers. :)

Making large changes is an adventure

- Change via decree/mandate can work, …
- But we found it much easier to build up through sub-quests.
  - Initial Clang/LLVM work was not intending to replace GCC.
  - Eventually, a small group of people saw change as the only reasonable path forward.
  - Small, incremental improvements/changes are easier.
  - Got **partners**, **vendors**, and even **teams** from other parts of Google involved early.
  - Eventually, the end goal was clear:
    - “It’s time to have just one compiler for Android. One that can help find (and mitigate) security problems.”

Grow your support

A Brief History of LLVM and Android

- 2010 — RenderScript project begins
  - Used LLVM bitcode as **portable IR** (despite repeated warnings **NOT** to). :P
  - On-device bitcode JIT (later becomes AOT, but actual code generation is done on device).
  - Uses same LLVM on-device as for building host code with Clang/LLVM - we **<3** bootstrapping!
- March 2012 — **`LOCAL_CLANG`** appears (Gitiles).
  - Compiler-rt (for ASan), libpng, and OpenSSL are among the first users.
  - Other users appear as extension-related ABI issues spring up.
- April 2014 — Clang for platform != LLVM on-device (AOSP / Gitiles).
- July 2014 — All host builds use Clang (AOSP / Gitiles).

LOCAL_CLANG

- Flag for Android’s build system.
- If set to **true**, use Clang to compile this module.
- If not defined, use the regular compiler.
- Pretty simple, right?
- If set to **false**, use GCC to compile this module.

`LOCAL_CLANG := false`

- Need to retain some instances of GCC-specific testing.
  - Bionic (libc) needed to check that headers/libraries could still work for native application developers using GCC (NDK).
- Some tests were a little too dependent on GCC implementation details:
  - **`__stack_chk_guard`** explicitly **`extern`**-ed in and mutated in bionic (libc) tests!
- Other areas where we just didn't know how to fix bugs yet.
  - Valgrind was the last instance of this escape to be fixed in AOSP.
    - Wrong clobbers for inline assembly in 1 case.
    - ABI + runtime library issues (we'll chat about aeabi later).

Escape hatches are vital

Escape hatches are vital

- If we had to turn off Clang entirely each time we hit a bug, none of us would be here right now.
- We would be chained to our desk fixing bugs still.
- Lots of people working on this makes it parallel, so long as everyone can make progress — all or nothing is a bottleneck you can’t afford.

Two Builds for the Price of Two

- A simultaneous, obvious extension of `LOCAL_CLANG` was the concept of the **default** platform build.
- Original default was GCC.
- We were eventually able to set up a separate build target (actually multiple device targets) that used Clang as the default toolchain.
- Why didn't we do this first?
  - Because devices didn't boot with Clang...
  - And many things didn't even compile successfully with Clang!

Example: aeabi functions

```
void __aeabi_memcpy(void *dest, void *src, int size)  // Please ignore the 'int'. ;)
{
  memcpy(dest, src, size);
}
```

- Looks pretty harmless, but GCC and Clang treat Android ABI differently, at least for lowering calls to the runtime memcpy (**RTLIB::MEMCPY**).

```
void __aeabi_memcpy(void *dest, void *src, int size)
{
  __aeabi_memcpy(dest, src, size);  // Infinite loop!!!
}
```

- Discovered this in side-by-side builds after import of new third-party code.
- **LOCAL_CLANG** allowed us to ignore this issue for a short while.

Side-by-side builds are great

Side-by-side builds are great

- The ability to measure and “compare” things is why software engineering isn’t just an art*.
  - Correctness/Conformance Testing
  - Code size
  - Performance
  - …
- Helped **prevent** early regressions — compiler-dependent build breaks go to code submitters, and not just the wacky toolchain folks.

* not to be confused with Android’s managed runtime, otherwise known as ART.

Bugs happen ...

Sometimes it is the compiler

Assembly parsing is hard

- What does the following assembly code do?

```
and $1 << 4 - 1, %eax
```

- GCC assembler parses `(1 << n - 1)` as `((1 << n) - 1)`.
- LLVM assembler parses `(1 << n - 1)` as `(1 << (n - 1))`.
- Bionic hit this ambiguity in an optimized `strrchr()` (AOSP / Gitiles).
  - Compiler/assembler bug or regular code bug?
  - Why not both?

Undefined Behavior

- Signed integer overflow :(
  - -fwrapv makes this defined.
  - Can expose other bugs (in addition to harming performance).
- Nonnull manifested a few ways in Android:
  - Removing `this` checks in Binder. (AOSP / Gitiles)
    - ```
      sp<IBinder> IInterface::asBinder()
      {
        return this ? onAsBinder() : NULL;
      }
      ```
    - Except people had been calling `(nullptr)->asBinder()` in lots of places.
      - Further cleanup replaced this with a static method. (AOSP / Gitiles)
  - ```
    // src == nullptr
    if (!src || !dst) size = 0;
    memcpy(dst, src, size);
    ```

Inline Assembly Revisited

- Legacy wrapper functions:
  - Do some minor action up front.
  - Pass existing caller arguments through to another (possibly tail) call.
  - Maybe return a different value (always 0 in these cases).
- Input/Output/Clobber constraints might not matter until one day the compiler says that they do. (AOSP / Gitiles)
- SWEs work to make the compiler happy, even if it isn't correct (enough).
  - Clang stomped all the arguments/returns for the inline assembly, while GCC didn't bother touching any of the argument/return registers.
  - Nobody noticed until we tried to switch to Clang.
  - Even a GCC update or slight change to the source files (due to inlining) could have caused a bug that would likely be misattributed as a "miscompile".

Lots of empathy for other teams

Lots of empathy for other teams

- They are going to have undefined behavior.
- They are going to have general bugs that got exposed by the transition.
- They need support, not an adversary. C++ is a worthy enough adversary for all of us.
- You’re going to want their empathy/understanding when it **is** a compiler bug.

A Continued History of LLVM and Android

- 2012 - 2016 — Everything you just saw.
- December 2014 — First side-by-side (mostly) Clang build for Nexus 5.
- January 2016 — Android Platform defaults to Clang.
- April 2016 — **99%** Android Platform Clang (valgrind was the last!)
- August 2016 — Forbid non-Clang builds (AOSP / Gitiles).
  - Whitelist for legacy projects (started in AOSP / Gitiles).
- October 2016 — **100%** Clang userland for Google Pixel.

The Platform Numbers

- 597 git projects in aosp/master (10/18/2017).
  - 37M LOC C/C++ source/header files in aosp/master alone.
  - 2M LOC assembly additional!
  - 25.3M LOC of C/C++ is in aosp/master external/*.

    The above data was generated using David A. Wheeler's 'SLOCCount' on a fresh checkout of aosp/master. It does not include duplicates or generated source files either.
- >150 CLs alone to clean up errors that Clang **uncovered**.
  - Some of these were Clang bugs.
  - Many of these were actual user bugs.
  - Some were both.
- ~2 years from high-level decision to shipping!
- ~6 years if you count our early efforts!

BONUS - How to deprecate something in a short time!

- STLPort (a C++ runtime library) was a blocker for switching to Clang (and libc++).
- “Unbundled” Android 1st party apps didn’t want to switch to libc++/Clang.
- It’s hard to incentivize good behavior.
  - “Nothing really changes”, maintenance is viewed as “unnecessary churn”, ...
  - But we **want**/**need** to remove deprecated components in a reasonable timeframe.
  - Sound familiar yet? This story probably resonates with many of us here.
- Enter the “Sleep Constructor”.

The Sleep Constructor

```
__attribute__((constructor))
void incentivize_stlport_users() {
  ALOGE("Hi! I see you're still using stlport. Please stop doing that.\n");
  ALOGE("All you have to do is delete the stlport lines from your makefile\n");
  ALOGE("and then you'll get the shiny new libc++\n");
  sleep(8);
}
```

- Seriously, we added an 8 second sleep in May 2015! (AOSP)
- And then we doubled it to 16 seconds in June 2015!
- Deleted it in August 2015, because no one was left using STLPort!

Platform Takeaways

- Grow your support.
- Escape hatches are vital!
- Side-by-side builds are great.
- Bugs happen — Sometimes it is the compiler.
  - People are going to be upset when this happens, so ...
- Lots of empathy for other teams
  - s/other teams/everyone/ for when it is actually the compiler.
- When being nice fails — Sleep Constructor!

Linux Kernel in 2014/2015

- Patches provided by LLVMLinux (http://llvm.linuxfoundation.org)
- Some work upstreamed
- Large out-of-tree patchstack, last updated in January 2015

- Kbuild changes in fairly good shape
- Many architecture-specific patches labeled “DO-NOT-UPSTREAM” or “Not-signed-off-by”

Not shippable, but worth keeping an eye on.

Linux Kernel in 2016

- clang/LLVM continued maturing as a toolchain
- Many LLVMLinux patches no longer needed
- Got working on dev boards and qemu, but quickly ran into limitations:
  - Upstream Kbuild support for LLVM bitrotted
  - Couldn’t compile crypto code on x86 or ARM64
  - Misaligned stacks on x86
  - ARM64 EFI stub panicked before starting kernel
  - Core kernel module (futex) didn’t always assemble on ARM64
  - ...

Tantalizingly close. Several teams in Google interested in pushing this to completion.

Why Is the Linux Kernel Special?

23.2 million LOC codebase [0] that evolved simultaneously with GCC, and does things that most codebases can't:

- Act as its own dynamic linker, libc, and libcompiler-rt
- Directly access system registers and I/O memory
- Handle CPU faults
- Manipulate page tables
- Mix 16-bit, 32-bit, and 64-bit code in a single executable
- Simultaneously act like an ELF executable, COFF executable, and neither of the above

[0] https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.14-Code-Size

Why Isn’t the Linux Kernel *That* Special?

- Clang already builds lots and lots of diverse codebases.
- … including FreeBSD kernel!
- Tons of bugs already shaken out, relatively few unique corners of the C language.

- Most of the weirdest, kernel-y, low-level stuff *isn’t really meaningful in C anyway*.
- Linux falls back to assembly for things that need very precise semantics (i.e., most of the previous slide).

Sometimes It’s the Kernel ...

clang turns the **`llist_for_each_entry()`** macro into an infinite loop.

- Take a pointer **`node`**
- Walk node backwards **`offsetof(node, member)`** bytes to compute **`pos`**
- Reconstruct original **`node`** by computing **`&pos->member`**
- Terminate loop if **`&pos->member == NULL`**

```
#define llist_for_each_entry(pos, node, member)                         \
        for ((pos) = llist_entry((node), typeof(*(pos)), member);       \
             &(pos)->member != NULL;                                    \
             (pos) = llist_entry((pos)->member.next, typeof(*(pos)), member))
```

(source: include/linux/llist.h)

Sometimes It’s the Kernel ...

Loop only terminates if pointer underflow and pointer overflow cancel each other out. Not defined behavior!

Code first introduced in August 2011:

f49f23abf3dd lib, Add lock-less NULL terminated single list

Fixed in July 2017, by casting to **`uintptr_t`**:

beaec533fc27 llist: clang: introduce member_address_is_nonnull()

… But Sometimes It’s the Compiler

The **futex** module tests an API’s availability by asking it to dereference **NULL**:

```
/*
 * This will fail and we want it. [...] NULL is
 * guaranteed to fault and we get -EFAULT on functional
 * implementation, the non-functional ones will return
 * -ENOSYS.
 */
if (cmpxchg_futex_value_locked(&curval, NULL, 0, 0) == -EFAULT)
```

(source: kernel/futex.c)

… But Sometimes It’s the Compiler

Clang assigns the **NULL** constant to a register that can’t be loaded from:

```
  CC      kernel/futex.o
/tmp/futex-f1b216.s: Assembler messages:
/tmp/futex-f1b216.s:14498: Error: integer 64-bit register expected at operand 2
-- `prfm pstl1strm,[xzr]'
/tmp/futex-f1b216.s:14499: Error: operand 2 should be an address with base
register (no offset) -- `ldxr w12,[xzr]'
/tmp/futex-f1b216.s:14502: Error: operand 3 should be an address with base
register (no offset) -- `stlxr w13,w10,[xzr]'
```

https://bugs.llvm.org/show_bug.cgi?id=33134 (fixed in r308060)

Linux Kernel in 2017

State of the upstream kernel summarized at https://lkml.org/lkml/2017/8/22/912

- Kbuild, x86_64, and ARM64 support upstreamed

```
$ git diff --stat 3b61956a41a5..994d12e0b4bb
[...]
 28 files changed, 198 insertions(+), 145 deletions(-)
```

- One out-of-tree patch still needed for ARM64 (LLVM bug 30792)
- Backports to 4.4 and 4.9 available from Chromium and AOSP (android-{4.4,4.9}-llvm branches)
- Production ready?

Pixel 2


File Edit View Search Terminal Help

Benefits

- Consistent toolchain for kernel and userspace
- LLVM development beyond critical mass
- Better static analysis + dynamic analysis (sanitizers)
  - Sanitizers developed first in LLVM, have significantly more features
  - KASAN+ramdumps helps A LOT, recommended for dedicated dogfooders
- Additional compiler warning flag coverage
- More tools planned in the future (control-flow integrity, LTO, PGO)
- Shake out undefined behaviors
- Improve both kernel and compiler code bases

LLVM bugs found/hit from Linux Kernel effort

- [AArch64] -mgeneral-regs-only inconsistent with gcc
- false(?) -Wsequence warning
- typeof(const members of struct), -std=gnu89, and -Wduplicate-decl-specifier vs gcc7.1
- Wrong relocation type in relocatable LTO link
- Clang integrated assembler doesn't accept asm macro defined in one asm directive and used in another
- Invalid LDR instruction with XZR

New warnings for our kernel (that found bugs)

- -Wlogical-not-parenthesis
- -Warray-bounds
- -Wunused-function
- -Wimplicit-enum-conversion
- -Wformat-extra-args
- -Wframe-larger-than=
- -Wignored-attributes
- -Wduplicate-decl-specifier
- -Wshift-overflow
- -Wself-assign
- -Wsection
- -Wtautological-pointer-compare
- -Wparentheses-equality
- -Wenum-conversion
- -Wliteral-conversion
- -Wheader-guard
- -Wnon-literal-null-conversion
- -Waddress-of-packed-member disabled :(

Test these with $(CC) -c -x c /dev/null -W<arning> (https://github.com/Barro/compiler-warnings seems neat)

Can LLVM compile a working Linux kernel?

Yes*†‡¶§. Compile vs run is a big difference, too.

* 4.4 and 4.9 LTS Chromium/Android forks, ToT (4.14-rc5) (assuming no one broke anything since this morning)

† Our device specific configurations

‡ Run on our specific hardware

¶ Cannot assemble or link, still deferring to binutils' ***as*** and ***ld***

§ ARCH=arm64 || ARCH=x86_64

Testing

- Presubmit (compile+boot tests)
  - Clang
  - GCC
  - KASAN
  - lint
- Postsubmit
  - fuzzing
  - regression testing

Try it today!

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git && \
      cd linux && make localmodconfig && make CC=clang


$ ARCH=arm64 CROSS_COMPILE=arm64-linux-gnu- make CC=clang HOSTCC=clang
```

Future Work

- Switch to LLVM tools from binutils.
  - Integrated assembler
    - Clean up existing assembly code.
    - Improve Clang assembly parsers.
  - LLD
  - control-flow integrity, LTO, PGO
- Continued community involvement both upstream and with our users.
  - Public Mailing List: https://groups.google.com/forum/#!forum/android-llvm
  - Android toolchain bugs can be filed at: https://github.com/android-ndk/ndk

Thank you

To our audience, the LLVM community, and our fellow adventurers for helping to make Android + LLVM a success!