Rethinking Linux Tools with Compact Debuginfo Formats

Stephen Brennan
Oracle

Changing the way we think about debuginfo can bring about interesting new capabilities.

What is debuginfo?

```
[stepbren@stepbren-plc ~]$ gcc main.c -o main
[stepbren@stepbren-plc ~]$ strip main
[stepbren@stepbren-plc ~]$ gdb -q main
Reading symbols from main...
(No debugging symbols found in main)
(gdb) break main
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (main) pending.
(gdb) run
Starting program: /home/stepbren/main
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
hello
world
[Inferior 1 (process 11620) exited normally]
(gdb)
```

```
[stepbren@stepbren-plc ~]$ gcc main.c -o main
[stepbren@stepbren-plc ~]$ gdb -q ./main
Reading symbols from ./main...
(No debugging symbols found in ./main)
(gdb) break main
Breakpoint 1 at 0x40112a
(gdb) run
Starting program: /home/stepbren/main
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, 0x000000000040112a in main ()
(gdb) next
Single stepping until exit from function main,
which has no line number information.
hello
world
__libc_start_call_main (main=main@entry=0x401126 <main>, argc=argc@entry=1,
    argv=argv@entry=0x7fffffffe258)
    at ../sysdeps/nptl/libc_start_call_main.h:74
74        exit (result);
(gdb) q
```

```
[stepbren@stepbren-plc ~]$ gcc -g main.c -o main
[stepbren@stepbren-plc ~]$ gdb -q ./main
Reading symbols from ./main...
(gdb) break main
Breakpoint 1 at 0x401135: file main.c, line 5.
(gdb) run
Starting program: /home/stepbren/main
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffe258) at main.c:5
5               printf("hello\n");
(gdb) n
hello
6               printf("world\n");
(gdb) n
world
7               return 0;
(gdb) c
Continuing.
[Inferior 1 (process 11379) exited normally]
(gdb)
```

The Classical Approach

- Distributions compile packages with "-g" (or their preferred debuginfo settings)
- Distributions strip DWARF info into a separate package (or discard it)
  - Why? DWARF is big.
- Runtime binaries have minimal debuginfo
- Encounter a crash? Install debuginfo!

Yes, this is simplified and rather outdated.

Debuginfod! Crashpad! ABRT! systemd-coredump!

But it's still the prevailing mental model.

“Debuginfo” is not a great word

- Misleading
  - “Debuginfo” need not be only for debugging
  - Applications could use certain info for a variety of cases
- Non-specific
  - Types? Symbols? Variables? Source Mappings?
  - Stack Tracing / Unwinding?
  - Macro Definitions?
- Traditionally, “debuginfo” tends to be shorthand for DWARF

Reflection, Runtime Introspection

- Most high-level languages have “reflection” or some other runtime code inspection
  - Java: “java.lang.reflect”
  - Python: `globals()`, `code`, `traceback`, `object.__dict__`, `object.__type__`
  - These languages are using “debuginfo” but they don’t call it that!
- C does not, with a few exceptions:
  - glibc does provide backtrace(3)
  - compile time assertions & macro stuff
- The kernel is especially unusual:
  - Can unwind its own stacks (FP, ORC)
  - Can lookup its own symbols (kallsyms)
  - Can reason about its own types (BTF)

Types of Data (Including, but not limited to...)

- Type definitions
  - What is the size of an int?
  - Members of task_struct?
- Function, variable, symbol addresses
- Function, variable types
- Stack tracing (functions only)
- Stack unwinding (full caller state)
- Source code locations
- Inline functions
- Macro definitions

Common Limitations:

- Only "exported" symbols
  - Kernel: `EXPORT_SYMBOL*()`
  - Shared library APIs
- Core kernel vs modules
- Only functions (common w/ tracing)

Things I Consider Debuginfo

- DWARF
- Compact Type Format (CTF)
- BPF Type Format (BTF)
- SFrame
- ORC
- ELF Symbol Tables

- .gnu_debugdata (a Fedora-ism)
- Kallsyms

- Last Branch Record (LBR)
- Frame Pointers

Normal things to consider debuginfo

Weird things to consider debuginfo

Some Linux “Debuginfo” Formats & What They Can Do

Capability	DWARF	CTF	BTF	SFrame	ORC	.gnu_debugdata	Kallsyms	LBR	FP
Var/Sym Address	✅		⚠️			✅	✅		
Var/Sym Type	✅	✅	⚠️						
Type Definition	✅	✅	✅						
Source Code Mapping	✅								
Stack Tracing	✅			✅	✅			⚠️	⚠️
Stack Unwinding	✅								
Macro Definitions	⚠️								

Some Linux “Debuginfo” Formats & What They Can Do

Capability	DWARF	CTF	BTF	SFrame	ORC	.gnu_debugdata	Kallsyms	LBR	FP
Var/Sym Address	✅		⚠️			✅	✅		
Var/Sym Type	✅	✅	⚠️						
Type Definition	✅	✅							
Source Code Mapping	✅								
Stack Tracing	✅			✅	✅			⚠️	⚠️
Stack Unwinding	✅								
Macro Definitions	⚠️								

Currently, only per-CPU variables have address & type info.

Not ideal for asynchronous stack unwinding (e.g. debugger/perf)

GCC only generates `.debug_macro` section with `-gdwarf -g3`, uncommon!

Requires privileged hardware access; rather limited stack “depth”

Observations

- Many interesting capabilities do not require all features of DWARF.
- Combine a few non-DWARF debuginfo types and you can power a debugger!
  - We've been working on this in drgn!
- The Linux kernel is unusual in having several types of debuginfo available already
  - Can they be unified more?
- I wish this was my innovation, or even an "innovation," but I can't claim that.

Case Studies & Future Ideas

Historical Case Study

- Many system utilities, like ps(1), originally worked by opening /dev/kmem and inspecting kernel data structures.
  - They needed to run as root or setuid.
  - They also generally needed to be rebuilt on kernel changes. (**lack of type info**)
- Over time, kernel APIs have been introduced to remove the need for this.
- /dev/kmem was removed in 2021 after being disabled by most distros for many years
- ps(1) *should* have proper kernel API/ABI, but not everything should!

BTF CO-RE

- By using BTF, structure offsets of BPF code are adjusted

```
struct dentry *parent;
bpf_core_read(&parent, 8, &dentry->d_parent);
```

- Resulting BPF code can run on multiple kernel versions.
- Eliminates the need for compiling BPF on the target system.

drgn: “DWARFless” debugging

- drgn is a programmable debugger with a focus on Linux kernel support
- DWARF is useful for interactive, but “DWARFless” support can power other use cases
- Kallsyms symbol tables (kernel & modules)
  - Supported in v0.0.30 (Dec 2024)
- Unwinding with built-in ORC (no debuginfo files required)
  - Supported in v0.0.31 (April 2025)
- CTF type & variable lookup (with CTF from Oracle UEK)
  - In review upstream
- BTF type & variable lookup
  - Pending global variable info in BTF -- I’m working (slowly) on this!
  - In-progress branch available

Fedora: .gnu_debugdata & .eh_frame

- Rationale: enable userspace stack traces after a crash, without debuginfo
  - Users can upload a small crash report with stack traces, rather than a core dump
- .gnu_debugdata: a second ELF file compressed and stored as an ELF section
  - Contains additional symbol table entries
- .eh_frame: DWARF CFI stack unwind info
  - Runtime binaries on Fedora, Ubuntu, Arch, and likely others retain this section.
  - Frequently it is necessary for C++ exceptions, but is also good for runtime stack traces.
- Result: great stack tracing support on GDB, ABRT, and some other tools (e.g. drgn)

Makedumpfile: VMCOREINFO as a compact format

- Makedumpfile is used to create compact kernel core dumps
  - Filters unnecessary data from /proc/vmcore and writes a compressed output
- In order to work, it needs some basic symbols, type info, and other data
  - Can be provided by DWARF, but this is not ideal
  - VMCOREINFO exists to remove that requirement

```
OSRELEASE=6.17.9-300.fc43.x86_64
BUILD-ID=825d7bffa209f1b6919a685fa6eb2da5b7472e37
PAGESIZE=4096
SYMBOL(init_uts_ns)=ffffffff9dbf4b20
OFFSET(uts_namespace.name)=0
…
```

Future: BTF & kallsyms for makedumpfile

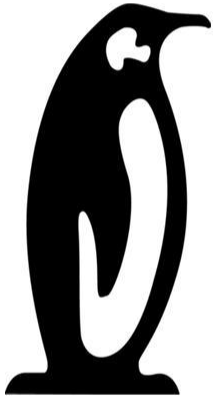
- Multiple implementations ongoing!
  - Tao Liu - posted v2 in Oct 2025
  - Stephen Brennan (me) - mistakenly implemented before checking the mailing list!
- Allow makedumpfile to access the full symbol table & all types
- Enables new features:
  - Exclude GPU buffers from vmcore
  - Include userspace stack memory in vmcore, enabling user stack traces from panic kernel

Future Ideas

- Printing memory formatted as structures, like GDB & drgn
  - New printk format specifier?
  - Kernel - via BTF & printk? Could be useful for development & debugging.
  - Similar for userspace via libc & BTF or CTF
- Converting enum values to strings
- Perf - perf mem, perf c2c - highlight structure members, rather than symbol/addresses.
- ABI checking - this is already being worked on with BTF & CTF for libabigail
- Tightly coupling kallsyms & BTF
  - They reference similar strings, but have separate string tables

Conclusion

- Non-DWARF formats are a useful source of info for debugging
- Combining the available types can form a useful, minimal alternative to DWARF
- Together they aren't just for debugging, but all forms of introspection
- There is a lot of room to rethink many different tools from this perspective



東京 2025 LINUX PLUMBERS CONFERENCE

TOKYO, JAPAN / DECEMBER 11-13, 2025