GNU Toolchain and CTF

Elena Zannoni
elena.zannoni@oracle.com
July 2019

Background

- Debugging information:
  - generated by the compiler to help debuggers and other tools gather information about a program and its behavior
  - encodes types and structure of a program (variables, functions, data structures, line numbers, etc.)
  - several debugging formats exist (stabs, DWARF, coff...)
- DWARF is the most commonly used one

A bit of DWARF

- Very complex
- Debug info stored in multiple .debug_<...> sections
- Dwarf uses DW_TAG_<name> for naming types
- Each type has multiple attributes (DW_AT_<name>) to describe it
- Backtrace / unwind information: in .debug_frame section (CFI)
- Each subroutine has a virtual unwind table
  - describes at each code location (instruction) how to recover the values of the registers and its frame address
- Tables are described indirectly by a set of operations / instructions (stored in the debug info)
- Instructions: used to generate the table when needed, operate as a stack machine.
- Size of .debug_* sections is quite large

Motivation

- Perform debugging when debuginfo not available (stripped executable and/or debug info not installed)
- Analyze stack traces in the absence of debug info
- Do so in a fast way, as opposed to off-line processing like done for DWARF used with debuggers
- Do not expose unnecessary information via the debuginfo in production binaries
- No ad-hoc solutions (hard to maintain)
- Must work also in the presence of always changing compiler optimizations

CTF

- CTF stands for "**Compact C Type Format**"
- Describes type information
- CTF originates from Solaris, but extended significantly for Linux
- More compact and easier to use/parse than DWARF
- Used by DTrace on Linux for the kernel since 2012 (dwarf2ctf tool, libdtrace-ctf RPM)
  - Libdtrace-ctf:
    - https://github.com/oracle/libdtrace-ctf
  - Dwarf2ctf script:
    - https://github.com/oracle/dtrace-linux-kernel
- **General framework**, not DTrace specific
- Used also on FreeBSD, Solaris and MacOS

CTF vs DWARF

- CTF doesn’t store its own encoding
  - No extra info used to describe the fields.
  - Description is implicit in the representation
- To decode DWARF the “key” is in the debuginfo itself, to decode CTF the key is the specification
- DWARF: model everything in C and everything to do with the mapping between C and the hardware
- CTF: Type identifiers are derived by array offsets
- CTF: Space saving (for instance reuse strings from ELF string table)
- CTF: only model the type system and mapping from symtab entries to types
- CTF: no location lists, expressions, stack machines

Additions to the GNU Toolchain

- CTF generation in GCC (new switch -gt)
- GDB support for debugging
- Binutils includes:
  - CTF handling in objdump and readelf
  - Linker modifications
  - Libctf library

How to use it

- Compile with -gt with level either 0, 1 or 2
  - Level 0 : turns off the CTF generation
  - Level 1 : (reserved for later use) generate backtrace info only
  - Level 2 : complete CTF generation (default)
- Objdump with
  - `--ctf=SECTION`
  - `--ctf-parent=SECTION`
- Readelf with
  - `--ctf=SECTION`
  - `--ctf-symbols=SECTION`
  - `--ctf-strings=SECTION`

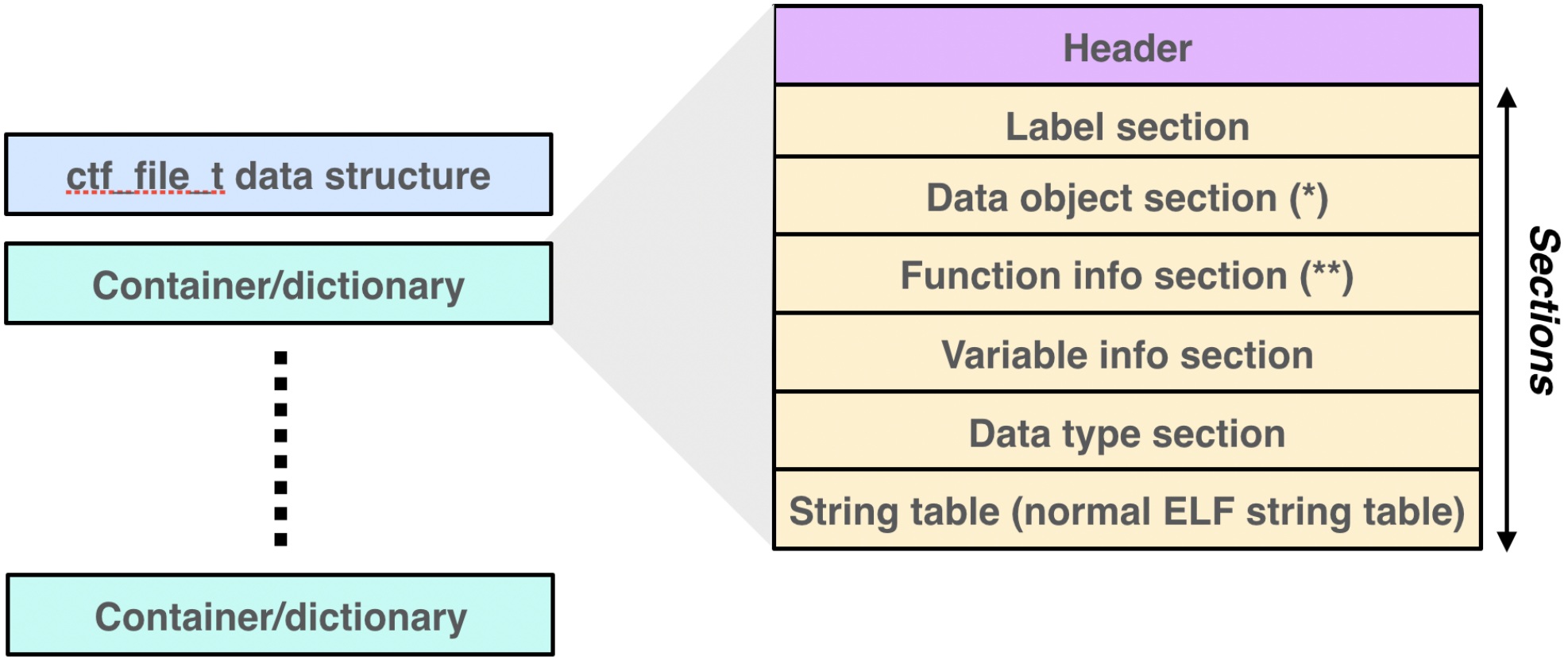
% **size -A /tmp/gcc/bin/ld**

```
/tmp/gcc/bin/ld  :
section                size      addr
.interp                  28   4194984
.note.gnu.build-id       36   4195012
.note.ABI-tag            32   4195048
.gnu.hash               172   4195080
.dynsym                3264   4195256
.dynstr                1116   4198520
.gnu.version            272   4199636
.gnu.version_r          144   4199912
.rela.dyn               216   4200056
.rela.plt              2808   4200272
.init                    23   4206592
.plt                   1888   4206624
.text                949825   4208512
.fini                     9   5158340
.rodata              1447296   5160960
.eh_frame_hdr          19172   6608256
.eh_frame             122760   6627432
.init_array                8   6757888
.fini_array                8   6757896
.dynamic                 480   6757904
.got                      16   6758384
.got.plt                 960   6758400
.data                  25136   6759360
.bss                   22976   6784512
.comment                  68         0
.debug_aranges          6320         0
.debug_info          4323894         0
.debug_abbrev         144836         0
.debug_line           750267         0
.debug_str            231568         0
.debug_loc           2069864         0
.debug_ranges         179760         0
.ctf                  212598   6815680
Total               10517820
```

More Details

- CTF is Compressed:
  - When size of CTF is above threshold (currently 4 KBytes)
  - Done at writeout time
  - Compress type table, string table, not the header
  - Flag in header indicates if it’s been compressed
- CTF can coexist with DWARF
- CTF is not stripped by default
- RPMs:
  - DWARF info included in debuginfo rpms
  - CTF info included in binary rpm
- Standard ELF symbol table must exist, CTF uses the structure and data of the symbol table to avoid storing redundant information.

Structure of CTF Information



*) Map 1:1 to the symbols of type STT_OBJECT
**) Map 1:1 to the symbols of type STT_FUNC

Structure of CTF Information

- ctf_file_t data structure
- “Containers” (or “Dictionaries”) are collections of types.
- A CTF "Container" (or “Dictionary”) has a header and a number of sections:
  - Header
  - Label section
  - Data object section: map 1:1 to the symbols of type STT_OBJECT
  - Function info section : map 1:1 to the symbols of type STT_FUNC
  - Variable info section
  - Data type section
  - String table: same format as a normal ELF string table

Header of a CTF Dictionary

- Preamble
  - Magic number (determines endianness)
  - CTF Version number (an integer)
  - Various Flags
- Reference to the parent dictionary
  - Used in case of conflicting types
  - Dictionary name (from the name of corresponding translation unit)

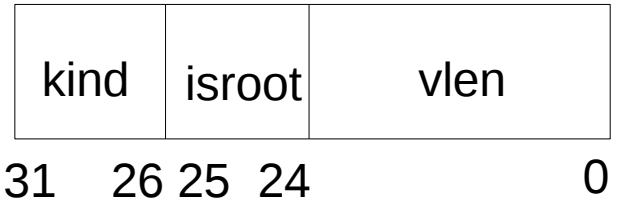
Data Type Section

- An array of variable length entries, each is a struct ctf_stype (or struct ctf_type) followed by optional variable-length data.
- Each type has an ID derived from its index in the array
- Two types of elements in the array depending on the size of the type
- The name of the type is represented by either its offset in the ELF string table, or its offset in the local (CTF) string table (if not present in the ELF string table)

```
typedef struct ctf_stype
{
   uint32_t ctt_name;    /* Reference to name in string table.  */
   uint32_t ctt_info;   /* Encoded kind, variant length.  */
   union
   {
      uint32_t ctt_size;    /* Size of entire type in bytes.  */
      uint32_t ctt_type;   /* Reference to another type.  */
   };
} ctf_stype_t;
```

Representing Types

ctt_info:



- **`kind`** is a constant with value CTF_K_* (one per type), such as CTF_K_INTEGER, CTF_K_FUNCTION, CTF_K_STRUCT, etc
- **`isroot`**: it is 1 if type is a named top-level type
- **`vlen`** (variable length): size of the type-kind-specific properties that follow.
- Type descriptions (analogous to DWARF attributes):
  - For functions: it is a list of argument types, with the ctt_type being the return type;
  - For integer and floating-point types use flags packed into a single uint32_t in the variant data encoding things like format, etc

CTF Versioning

- V1: Original version from Solaris
- V2: Ported to Linux for DTrace
  - increase the max number of types
  - Increase max number of struct and union members and enumerated values
  - Increase the number of type kinds to 64 (for future expansion).
  - No ABI change
  - Versioning of some constants
- V3: Still in flux
  - Header changes
  - Additional CTF_K_* values
  - Other…
- V4: in initial planning stage
- Maintain compatibility

```
% PATH=/tmp/gcc/bin:$PATH objdump --ctf=.ctf /tmp/gcc/bin/ld

/tmp/gcc/bin/ld:     file format elf64-x86-64
Contents of CTF section .ctf:
  Header:
    Magic number: dff2
    Version: 4 (CTF_VERSION_3)
    Flags: 0x1 (CTF_F_COMPRESS)
    Variable section:   0x0 -- 0xedf (0xee0 bytes)
    Type section:       0xee0 -- 0x133db3 (0x132ed4 bytes)
    String section:     0x133db4 -- 0x14cbfc (0x18e49 bytes)
  Labels:
  Data objects:
  Function objects:
  Variables:
    _xexit_cleanup -> a7e: void (*)() (size 0x8) -> a7d: void () (size 0x0)
    bfd_x86_64_arch -> 53ee: const struct bfd_arch_info (size 0x50) -> 238: struct bfd_arch_info (size 0x50)
    iamcu_elf32_vec -> afe9: const struct bfd_target (size 0x370) -> 286: struct bfd_target (size 0x370)
    bfd_last_cache -> c9b6: struct bfd * (size 0x8) -> 1f4: struct bfd (size 0x6)
    _CTF_NULLSTR -> 39bf: const char [0] (size 0x0)
[...]

  Types:
     1: long int (size 0x8)
        [0x0] (ID 0x1) (kind 1) long int  (aligned at 0x8, format 0x1, offset:bits 0x0:0x40)
     2: ptrdiff_t (size 0x8) -> 1: long int (size 0x8)
        [0x0] (ID 0x2) (kind 10) ptrdiff_t  (aligned at 0x8)
[...]

  Strings:
    0:
    1: A
    3: AOUTHDR
    b: AOUTHDR64
    15: AddressOfEntryPoint
    29: Age
    2d: B
```

Cross reference for ld .ctf section dump in previous slide (for clarity)

- void (*_xexit_cleanup) (void);
- static const bfd_arch_info_type bfd_x86_64_arch
- extern const bfd_target iamcu_elf32_vec;
- typedef long ptrdiff_t;
- #define CTF_K_TYPEDEF 10

Multiple Translation Units – Linker (ld)

- GCC generates one raw .ctf section per object file
- It is the linker's job to take a bunch of object files with one .ctf each and emit a unified type listing (dictionary) removing duplicates.
- ld walks through each translation unit (TU)’s dictionary and adds every newly encountered type in turn to a new single Dictionary.
- If there is a type conflict, it creates a child Dictionary for that TU and adds the type there instead.
- Conflicting types are types that have the same name but different definitions is separate TUs. Such as:
  - `struct foo {int bar;}` and `union foo {char *baz;}`
  - `typedef int foo_t` and `typedef long foo_t`
- In the end, ld produces one large shared dictionary (parent) and a few tiny sub-dictionaries (children).

Libctf

- Used to write and read ctf data
- Used by debugger and linker
- ctf_add_<type> (): build an element of that type
- ctf_open(), ctf_close(): open and close a ctf container/dictionary
- Lookup and iterator functions
- Header File: include/ctf-api.h (used by the debugger)

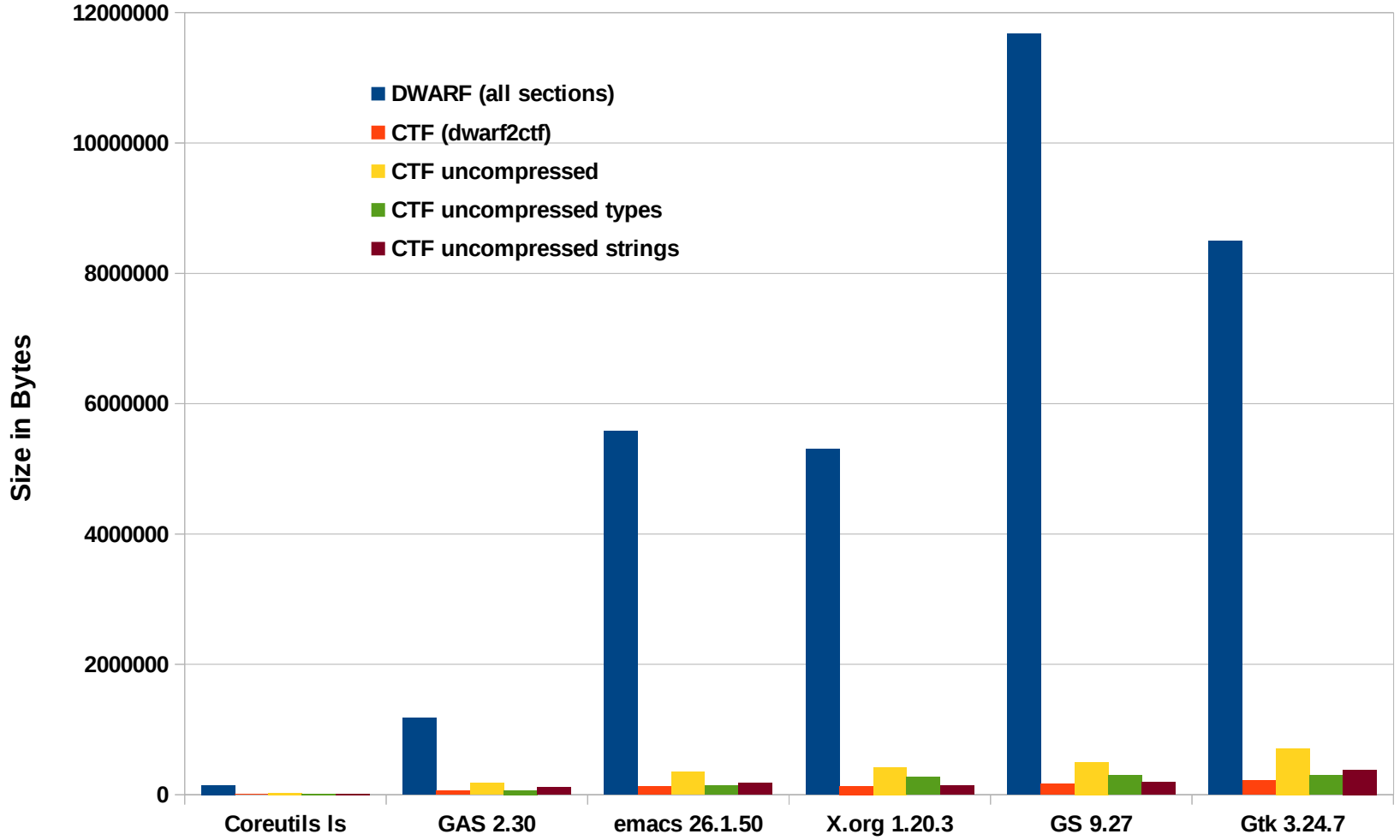
Upstreaming Status

- Still very much a work in progress
- Binutils:
  - readelf, objdump modifications (May 2019)
  - Libctf (May 2019)
  - Linker work: posted (July 2019). More work needed.
  - Few bugfixes and improvements: posted/committed (June & July)
  - Hopefully in next release of binutils (2.33) (for reference, binutils 2.32 released Feb 2019)
- GCC:
  - Initial set of patches for CTF generation: posted few revisions (May / June)
  - Undergoing more modifications based on reviews
  - New version of patch under testing now
  - Link time optimization (LTO) WIP to be posted next
  - Hopefully in gcc 10 (in 2020) (for reference gcc 9.1 released May 2019)
- GDB:
  - Posted, under review (July 2019)
  - Hopefully in next release of GDB (8.4) (for reference, gdb 8.3 released May 2019)

Observations

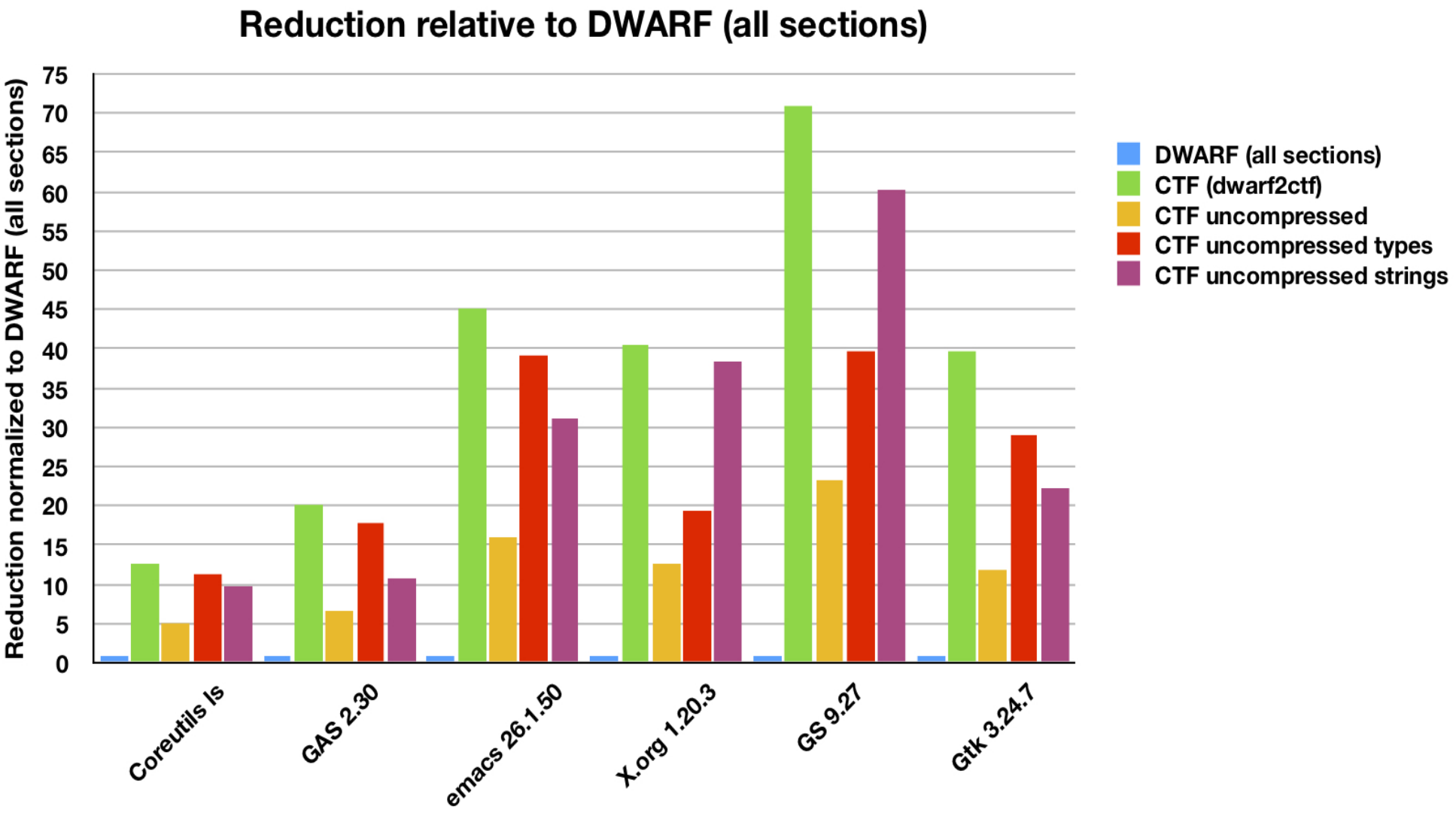
Program	DWARF (all sections) (bytes)	CTF (dwarf2ctf) (bytes)	CTF uncompressed (bytes)	CTF uncompressed types (bytes)	CTF uncompressed strings (bytes)
Kernel 5.2 + DTrace	1624364740	6677225	13910391	5984032	7138583
Coreutils ls	146456	11567	29174	13028	14939
GAS 2.30	1180929	58162	177774	66612	107851
emacs 26.1.50	5582440	123902	349295	142232	179184
X.org 1.20.3	5305506	131314	418567	272336	138096
GS 9.27	11679906	164675	502545	293216	193514
Gtk 3.24.7	8499254	213839	713263	292712	382680

Observations (*)



(*) Kernel Omitted from graph

Reductions relative to DWARF
(higher is better for CTF)



Future Development

- Discussions planned at Linux Plumbers Conference in September 2019 at the Toolchain MC
- Add 2 new index sections (objects and functions), for cases when the order of the symbols in the symtab is not known
- Link Time Optimization (LTO)
- More compactness and optimizations
- Write up the specification document (!!) (right now ctf.h)
- Backtracer
- Expand to other languages beyond C