CTFv4, the next generation

`Nick Alcock <nick.alcock@oracle.com>`

or, CTF == B++TF

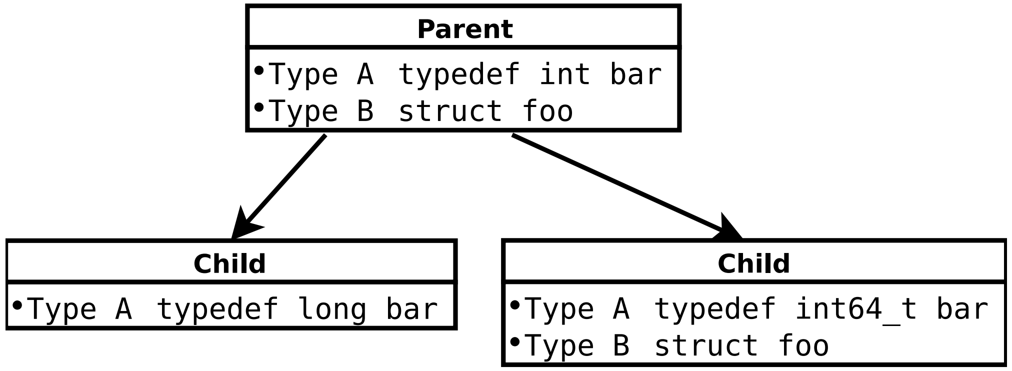
What is CTF?

- Provides a table of types in a C program, emitted by GCC, deduplicated by GNU ld, with read and write access provided by libctf in binutils.
- This table is known as a *dictionary* or *dict* because it largely contains definitions
- Also lets you map from ELF symbols to types
- Can represent ambiguously-defined (clashing) types.
- Useful for debugging, tracing, ABI analysis and reflection.
- The current version is CTFv3 (dating back to 2019).
- Spec: https://sourceware.org/binutils/docs/ctf-spec.html

Why?

- CTFv3 is pretty stable by now
- But so is BTF, and a lot of tools can manipulate that.
- They are very similar!
- Making the two mutually compatible will save effort for both sides
- This will be CTFv4: BTF, with a few extensions

Parents, children, conflictingness



- CTF has a notion of *conflicting types*
  - The deduplicator detects types with different definitions but the same name
  - The least utilized types get moved into child dicts named after the translation unit
  - These are all stored in a single *archive*, which is deduplicated together.

- For the kernel, these are named after kernel modules instead
  - Conflicting types within one module are stored but marked *hidden*

Invisible format differences

- The trickiest format differences are changes in the *distribution* of type IDs and strings in child dicts
- Type IDs
  - CTF: type IDs in children have their high bit on
  - BTF: type IDs in children run continuously from the parent
- Strings
  - CTF: child dicts can only refer to strings in child strtabs
  - BTF: strtabs are deduplicated against the parent

Invisible format differences

- Strtab improvement saves a *lot* of space (~20% after compression)
- Type ID change has no real benefits but is more or less harmless
- *However*, this makes the format more fragile: types and strings can no longer be added to parents after children are populated, and if you import children into the wrong parent absolute disaster results.
- In practice this is harmless, and CTF archives make it much less likely

More visible format differences: headers

- CTF has a bunch of features useful for ELF (but less so for the kernel) which we want to preserve.
- The most important of these is *typetabs*
- This needs several sections in the CTF file which are not in BTF: how to add this compatibly?

Adding headers compatibly

btf_header
• __u16 magic
• __u8 version
• __u8 flags
• __u32 hdr_len
• __u32 type_off
• __u32 type_len
• __u32 str_off
• __u32 str_len

ctf_header
• __u16 magic
• __u8 version
• __u8 flags
• __u32 hdr_len
• __u32 type_off
• __u32 type_len
• __u32 str_off
• __u32 str_len
• __u32 cu_name
• __u32 cu_parent_name
• __u32 cu_parent_ntypes
• __u32 cu_parent_strtab_len
• __u32 objt_off
• __u32 objt_len
• __u32 func_off
• __u32 func_len
• __u32 layout_off
• __u32 layout_len

- CTF header has extra members after the BTF header
- Header entries are all offsets
- BTF tools almost fully support dicts with larger hdr_len than sizeof(btf_header) (only endian-swapping doesn't work)
- so CTF has the same header, with a bigger hdr_len, and extra fields (which no BTF tools will consult)
- If BTF adds more fields, we bump the CTF format version and adapt our header

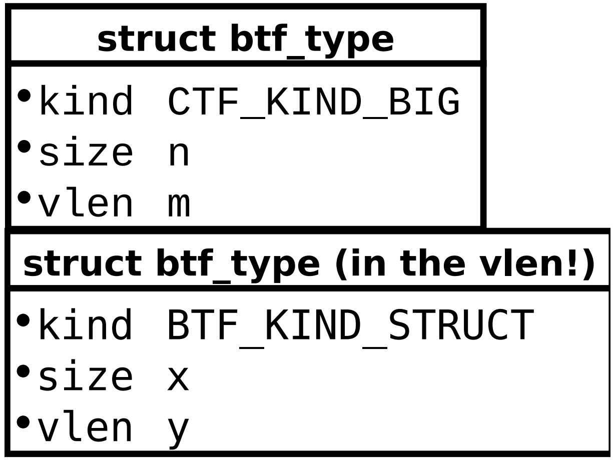
Smaller format differences

- The BTF type header differs from the CTFv3 one in three major ways
  - Fewer bits for the *vlen* (2^16 structure fields rather than 2^24)
  - Shorter max type size (types > 2^32 bytes unrepresentable
  - No hidden bit (problematic for conflicting types in modules)
- Mostly these don’t matter for kernels, so a reasonable decision for BTF.
- ... but we handle userspace too.

Smaller format differences

- We can compensate for all of these problems without changing the btf_type_t
- We introduce two new type kinds (BTF allows up to 32 and only 19 are in use: no shortage).
- These type kinds are a new sort of type kind, a *prefixed kind*.

Prefixed kinds



struct btf_type

```
•kind CTF_KIND_BIG
•size n
•vlen m
```

struct btf_type (in the vlen!)

```
•kind BTF_KIND_STRUCT
•size x
•vlen y
```

- Prefixed kinds have a btf_type immediately followed by *another* btf_type: the variable-length portion of this type is another type header! (But both types are one entity, with one type ID.)
- The ultimate type of the kind on the left is BTF_KIND_STRUCT, but its size is obtained by (n >> 32) | x, and its vlen by (m >> 16) | y.
- The BTF vlen is 16 bits, giving a total vlen of 32; the BTF size is 32, giving a total size of 64. This is bigger than CTFv3!
- Hidden types are handled via a CTF_KIND_HIDDEN kind that just hides the type it prefixes

Smaller additions

- CTFv3 implements *slices*, which change the encoding of a type (e.g. shrinking a bitfield).
- CTFv3 supports mapping ELF symbols to their types (including conflicting types found in child dicts).
- CTFv3 string offsets with their high bit on refer to an *external string table*, usually an ELF dynstrtab.
- All these features are carried into CTFv4 unchanged (but slices might in future be dropped: they have relatively marginal benefit).

BTF improvements over CTF

- We can adopt a number of BTF improvements and smaller changes into CTF unchanged:
  - 64-bit enums
  - data sections
  - BTF-specific representation for variables and forwards
- Data sections and variables are encoded as if they were types even though they’re not
- This is a bit ugly, but does not lead to loss of expressiveness

libctf API changes

- What does all this churn mean for the libctf API? Almost nothing!
- There is a new set of ctf_dict_set_flag() flags that control writeout and let callers say “only emit BTF, fail if CTF is called for” or “emit CTF always” (passed by ld). If neither is passed we emit BTF if it would lose no information, CTF otherwise
- We gain new functions to insert and look up the contents of data sections and insert and follow type and decl tags

libctf API changes

- Enums are... tricky, because the API currently treats their size as ints: new API functions for enum64s will probably be needed, if we don’t just bump the soname to change the prototypes to int64_t (I’d rather not).

- The API changes are nonetheless essential, because we eat our own dogfood: the deduplicator *uses* the public API functions to populate deduplicated dictionaries

Deduplicator changes

- Again, almost none: the new type kinds are mostly trivial, like existing ones or are types no other types can refer to
- There is one implication for existing BTF clients, though – the deduplicator can emit forwards to *anything*, including types that in C cannot be forwards (e.g. forwards to arrays): this is a signal that “this type is conflicting, look in children for the multiple definitions of it”: consumers must adapt if BTF decides to allow conflicting types
- We might just add a flag to cause ctf_link to drop conflicting types, and then consumers don’t need to worry about this or CTF archives. (See my upcoming talk at LPC.)

Kernel-side changes

- The ultimate goal of all of this is to allow the kernel (and pahole) to drop all their deduplicator and just use the toolchain's!
- The CTF approach is to link each module using the normal userspace approach, then have a separate program that calls the ctf_link API used by ld to dedup all these modules and the core kernel together into one big archive
- There's no reason BTF can't do the same thing, handing the result on to pahole to do the further transformations it wants to do to it. There are other possible approaches, but this one has the advantage that more or less all the code already exists.
- The open question is whether the kernel people *want* any of this or think it's useless frippery. My opinion is obvious :)

Questions?

- What else would be helpful?
- What other things would people like?
  - Better API functions?
  - More speed?
  - Free ice cream with every ld invocation?