LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Type deduplication in the CTF linker

Nick Alcock <nick.alcock@oracle.com>

Very quick CTF basics

- Model of the C type system, without scopes (effectively, one translation unit, global scope)
- Many sections: this talk will focus on the types section, which almost all other sections reference
- Each TU gets a `.ctf` section generated by GCC, but nearly all these types come from headers, so are shared between TUs: but C being C, shared types can refer to non-shared types via opaque structs, etc.
- It is up to the linker to deduplicate these.
- Doing this efficiently without producing a terrible type graph presents some interesting challenges!
- Spec: http://www.esperi.org.uk/~oranix/ctf/ctf-spec/index.html, http://www.esperi.org.uk/~oranix/ctf/ctf-spec.pdf

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Some numbers!

- The algorithm we came up with is fairly efficient
- These are .ctf section sizes, so *larger* than the type section size
- These sizes will shrink!

Program	Input size	Largest TU	Output size	Compressed	Time cost
GNU ld	4941163	167931	93815	52028	0.5s
libbfd	7960173	164000	262637	81523	1.5s
emacs 26.3	16672247	86267	303306	228780	5.5s

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

CTF type section basics

- An array of variable length entries: each is a `struct ctf_stype` (or `struct ctf_type`) with a *kind* recorded in it, followed by optional kind-dependent variable-length data (struct names and offsets, array bounds, etc).
- Each type has an ID derived from its index in the array: types can refer to other types by this ID.
- The deduplicator can create parent/child relationships between CTF dictionaries (with type IDs in distinct halves of the ID space: a dict is either a parent or a child, never both). You open a child then attach its parent to it. Types in a child dict can refer to types in the parent, but not vice versa (because you can open parent dicts on their own).
- Types form a graph, so it would seem that dedupping would be horrifically slow and painful (since everything is painful with graphs).

Linking CTF: The basics

- Nearly all the job of linking CTF is done by reusable code in binutils's libctf.so: see in particular the functions starting ctf_link* in include/ctf-api.h.
- Most of the work this does (in libctf/ctf-link.c) relates to sections other than the types section.
- Deduplicator interface is very narrow: all it has to provide back to the rest of the CTF linker is a mapping from a type ID in an input TU to an ID in the output. So we can ignore the rest of the machinery and focus on the deduplicator.

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Avoiding allocation nightmares

- We keep lots of references to deduplicated types (each of which is an SHA-1 hash value) and types in input TUs. We don't want to malloc for either of these if possible. (It makes the code too ugly!)
- Hash values are easy: store them only once as hashtab keys: `intern` them as atoms
- Identifiers for input types are harder. There are a *lot* of them, of the form "this type in this TU". We use them as hashtab keys all over the place.
- Number input TUs (whether in .o files or in archive members) in the link line, and pack the number of the TU containing each type together with a 32-bit `ctf_id_t` into a 'global ID', or 'GID', which can usually get stuffed into a pointer. (The TU number is literally an array index.)

The algorithm in three short steps

1) Hash every type in every TU (“input type”), recursively, hashing subtypes’ hashes into those of parents (a little simple caching means we only have to hash any given input type once): consider the hashes to be (hashes of) the *output types*.
2) Look across the set of named types for those where one name corresponds to multiple hashes: these types are *ambiguous*
3) Emit everything by traversing the set of type hashes from leaf to root, emitting each hash only once

This much is easy. The devil, as ever, is in the detail.

The problem in one word: cycles

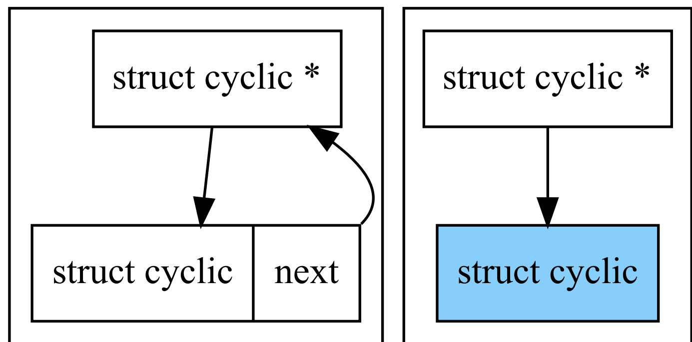
```
struct foo;

struct bar
  {
    struct foo *foo;
  };
```

```
struct foo
  {
    struct bar *bar;
  };
```

- We would like users to be able to get from the type `foo*` to `foo` at all times: also from forwards and opaque references to `struct foo` or `union foo` to the real thing: ideally we want to eliminate opaque references completely. This is not *mandatory*, but obviously highly desirable.
- But this makes the C type system cyclic!

What doesn’t work

- The original plan was to preserve the cycles, detect them, and stabilize them so that the same cycle’s types have the same hashes across all TUs they appears in.
- This fails because of opaque structs: they can be in a cycle in one TU but acyclic in another!

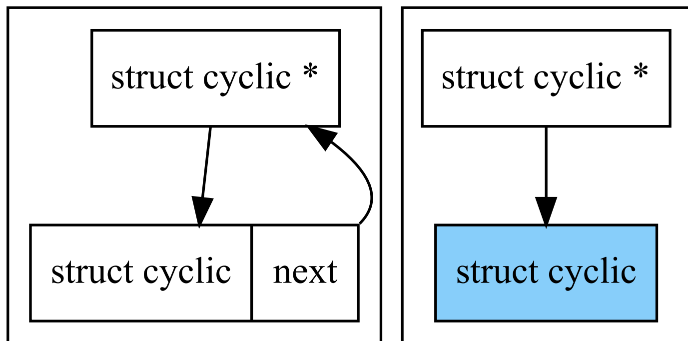


Cycles: the right way

- Don’t do too much damage to the type graph
- We must break cycles, but where?
- If we break at every type (don’t recurse to subtypes at all), all types become identical! We must discriminate.
- Break at the one thing all cycles must have: tagged structs (or unions).

Cycles: the right way

- When we see a tagged struct or a forward to one while hashing types to mix them in to a parent type, hash it as a *stub*. Stubs are basically just the *name* of the struct. Stubs do not hash the same as anything but other stubs.
- Now the graph on the right works! `struct cyclic *` has the same hash value in both cases.
- This can turn even the most complicated cyclic type graph back into something acyclic. (e.g. ‘struct bfd’).



LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

struct cyclic	yes_cyclic_t *bar
	struct cyclic *baz
	also_cyclic_t quux

typedef struct also_cyclic_t

yes_cyclic_t *

typedef yes_cyclic_t

struct also_cyclic	also_cyclic *cycle
	struct cyclic *another_cycle

struct (yes_cyclic_t)	size_t bar
	struct cyclic *foo

struct also_cyclic *

struct cyclic *

typedef int size_t

struct also_cyclic (stub)

struct cyclic (stub)

int

LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Ambiguity resolution

- What if we have types in two TUs that refer to a struct with the same name, but that struct is different in each TU? We only consider the name, so references to it hash to the same value!
- We can fix this by finding places where one C type name has multiple hashes (*ambiguous* types), and put all but one of those into (per-TU) child dicts (where they must be unambiguous).
- All types that cite ambiguous types also have to go into the same dict as the type they cite. All these child types together are known as *conflicting* types.

Cutting down conflicting forests

- Of course this means we can get stuck in cycles while marking things conflicting! We don’t want to emit whole cycles into child dicts anyway (50MiB of CTF for GNU ld because of three conflicting types deep under struct bfd: no!).
- Do not mark types that cite tagged structs or unions as conflicting. At emission time, emit a *synthetic* forward into the shared dict instead of conflicting structs.
- This is safe because if a structure T is ambiguous, opaque forwards to T are ambiguous too. So it’s fine to just give the user an opaque forward.
- We save a bit more space with a popularity contest; the most highly-cited type in an ambiguous set is shared.

The easy part: emitting types

- Walk over all type hashes (deduplicated types) from leaf to root, emitting each of them in turn, emitting conflicting types into multiple dicts and unconflicting types into only one.
- Forwards to the same type all hash to the same value as the type they refer to, so they collapse into their referents without our having to do anything
- We remember the input GID -> output ctf_id mapping for every type we emit, and use this to link types together
- We don’t emit struct members at this stage, so all cycles vanish: structs are emitted empty (CTF lets you add members to structs after initial insertion). Struct members get emitted later, after all types are emitted.

Future work

- Reduce memory consumption of libiberty hashtab: we use hashtabs as sets and create two hash *tables* per input type. Patch in progress that lets hashtabs share almost all their 100+ bytes of copy constructors etc
- Multithreading is possible to speed it up, but only programs with a great many types even see a slowdown. Probably best done as part of C++ work.

Other languages?

- Haven't thought much about other languages. Languages in which many more sorts of things are cyclic might be problematic, but the general "hash all things that *must* be present in cycles as stubs when hashing child types" approach seems more or less valid regardless. In particular I think it should work fine in C++ (except adding classes to the set of things that constitute cycles).

References

- The algorithm: https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=libctf/ctf-dedup.c;hb=HEAD#l28
- Other linking machinery: https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=libctf/ctf-link.c;hb=HEAD (includes the entire obsolescent non-deduplicating linker)
- The API: https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=include/ctf-api.h;hb=HEAD#l466