

Towards real-time ABI compatibility assurance with `libctf`

Nick Alcock

October 3, 2026

```
$ emacs  
ld.so warning: ABI incompatible!  
Linked against  
    int foo (char *bar)  
Run against  
    void foo (char *baz, enum foo)
```

End goal

Have ld.so warn about ABI-incompatible changes at runtime, rather than just crashing like we do now:

```
$ emacs
```

```
ld.so warning: ABI incompatible!
```

```
Linked against
```

```
    int foo (char *bar)
```

```
Run against
```

```
    void foo (char *baz, enum foo)
```

Asking libabigail and waiting 45 minutes (or 45 seconds with CTF) is OK for mass compatibility checks but not good enough for end users. We can do better, without making ld.so do anything difficult.

Getting there

- ▶ Brief CTF overview
- ▶ Symbol type info in CTF
- ▶ Compact type identity and the symtypehash
- ▶ ld.so changes (in progress)

Brief CTF history

- ▶ Models C type system, allows introspection
- ▶ Invented by Sun, used by DTrace and otherwise largely neglected, though it has obvious uses: e.g. `dlsym`
- ▶ People keep extending it; e.g. BTF
- ▶ CTFv3 has been upstream for years; new CTFv4 format (in late debugging phase now, in a branch) is a BTF superset
- ▶ Most of the stuff in this talk should work with `.BTF` sections as well

Rough structure of CTFv4

.ctf section with two components ('sections', confusingly: I'll call them 'tables' here):

- ▶ types (variable-length array, indexed by ID: also contains typed variables and function type signatures with names and 'datasecs', arrays of variable refs with some section info)
- ▶ string table used by type section
- ▶ that's it!

Rough structure of CTFv4

- ▶ In v4: multiple concatenated dicts are an 'archive' (with an optional trailing name table).
- ▶ BTF-compatible enough that we can often get away with writing it to a .BTF section.
- ▶ Usually stored in compressed sections (standard ELF). Will get section type assignments for all this before final upstreaming.

Symbol type info in CTF

Most likely starting place for CTF users: I have an ELF symbol, what type is it?

Symbol type info has been through several changes over the years:

- ▶ Sun-era: tables in the CTF section for function symbols and data object symbols, in symtab order; two tables with different formats (huge downsides): copies of parts of the type table
- ▶ CTFv3: upstream now. The same two tables, now with identical formats (array of type IDs), but in two distinct orderings: symtab order, or name order with separate index of symbol names. Still overcomplicated

New info

CTFv4: two *ELF* sections, after linking always sorted by symbol name.

- ▶ `.ctf.syntypetab`: straight type IDs
- ▶ `.ctf.syntypetab.all`: an array of (archive member index, type ID) for archive members other than the first

Symbol names obtained from CTF section's variable info.

In BTF-compatible mode, all symbol name strings are in the CTF string section: in BTF-incompatible mode, they come from the ELF dynstrtab (saving space).

Compact type identity and the symtypehash

- ▶ Deduplication cannot use type IDs to tell if types are identical! Existing upstreamed deduplicator uses SHA1 hashes internally to represent type identity.
- ▶ Try configuring binutils with `--enable-libctf-hash-debugging` and running with `LIBCTF_DEBUG=t` in the environment (*lots* of output)

Example: source:

```
#include <stdio.h>

typedef enum e { FOO, BAR} e_t;

struct foo
{
    enum e an_enum;
    int bar;
    struct foo *next;
};

int main (void)
{
    struct foo bar = {0};
    printf ("%i\n", bar.an_enum);
}
```

Example: enum

```
typedef enum e { FOO, BAR} e_t;hljfudjv
```

```
libctf DEBUG: 1: after hash addition of name: 36282fac116d9fd6b37cc425310e1a8510f08a53  
libctf DEBUG: 1: after hash addition of kind: 740906eeba668475d57ab19677d9b4a571145fb1  
libctf DEBUG: 1: after hash addition of enum size: ff262318d0234e9821aab21f13bb35ce0d080df4  
libctf DEBUG: 1: after hash addition of enumerator: Odd2810c8ae6fc3200fe07cc393a1daa80985271  
libctf DEBUG: 1: after hash addition of enumerand: a3b3308c11dd1bca33d24a174dbeb059357398cb  
libctf DEBUG: 1: after hash addition of enumerator: ff4f09ed9a9b20a5f68db66d04b340c18320a0a7  
libctf DEBUG: 1: after hash addition of enumerand: 99c9b9eec29834afa2d187444f108fd11ec7a6d1  
libctf DEBUG: Caching c, ID 0xc (e), 99c9b9eec29834afa2d187444f108fd11ec7a6d1 in final location
```

Example: struct

```
struct foo
{
    enum e an_enum;
    int bar;
    struct foo *next;
};
```

```
libctf DEBUG: 1: hashing thing with ID 0/d (kind 6): foo.
libctf DEBUG: 1: after hash addition of name: 2e774c1d2cfe7c32a48efba2b2c44caelf78cb45
libctf DEBUG: 1: after hash addition of kind: 112499d5b8257ee92ccba002bcfaa7b279eb0365
libctf DEBUG: 1: after hash addition of struct size: ad670951f3fe68fce79efea1e202671a8847e5d4
libctf DEBUG: 1: after hash addition of member name: 2c764ee6d805cd922b574694ca89c9d35a91bb56
libctf DEBUG: 1: Traversing to member an_enum
libctf DEBUG: 2: ctf_dedup_hash_type (0, c, flags 1)
libctf DEBUG: 2: Known hash for ID 0/c: 99c9b9eec29834afa2d187444f108fd11ec7a6d1
libctf DEBUG: 1: after hash addition of member hash: ebf0b119f09ffe4d32eafd9edb592471d8577fee
libctf DEBUG: 1: after hash addition of member offset: 9e9afab35ece6f31aee885c7ealcfcba95735ebc
libctf DEBUG: 1: after hash addition of member name: c1b6097c6b9a284720fdc687c116ef466994553d
...
libctf DEBUG: Caching d, ID 0xd (foo), 084d8d5ffbd52973f0a7166bcf59896750c7909f in final location
libctf DEBUG: Hash 084d8d5ffbd52973f0a7166bcf59896750c7909f, s foo, into output mapping for 0/d @ /home/n
```

'Binary-compatibility' hashes; suggestions wanted

All up in the air from now on: if you think of anything better, please let me know!

These hashes encode identity. Different types (by definition, with different SHA-1 hashes) may still be ABI-compatible: the deduplicator still considers them different.

Simpler version of hashes that only encodes ABI-affecting things would let us use hash changes to indicate ABI incompatibility, without looking at or decompressing `.ctf` at all!

- ▶ No function parameter names
- ▶ probably no names at all (just types, offsets, sizes)
- ▶ probably no enum info beyond name(?) and integral type size, since ABI-compatible updates often add members and change the identity of at least one trailing member.
- ▶ details will need careful review!

Binary-compatibility hashes: ELF section; improvements wanted

New ELF section, `.ctf.symtypehash`:

```
uint24_t symoffset (as in DT_GNU_HASH)
uint8_t  ent_bits
_BitInt(ent_bits) symhash[nsyms]
```

- ▶ Same length as `symtab`, less `symoffset`; laid out in `DT_GNU_HASH` order, like the `symtab`
- ▶ truncated hashes of symbols defined in this library, plus undefined symbols defined in other libraries: so, the entire `symtab`, really.

Each `symhash` entry is the binary-compatibility hash of that symbol, truncated to some number of bits.

Using binary-compatibility hashes; improvements wanted

- ▶ At link time, the linker copies the hashes of undefined symbols from the `.ctf.symtypehash` of the linked-against library to that of the library being generated, truncating to that library's `ent_bits` to fit, or zero-padding.
- ▶ Defined symbols get their entries from the `symtypetab` of the new library itself, truncating in the same way.

Example

```
ld -o foo.so foo.o -lbar
```

bar.so symtypehash (in part):

sym	type	recorded symtypehash (ent_bits 48)
a	SHN_OBJT	0xfa83a824a113

foo.so symtypehash:

sym	type	recorded symtypehash (ent_bits 32)
b	SHN_FUNC	0x3720c462
c	SHN_OBJT	0x4d720332
a	SHN_UNDEF	0xfa83a824 ;; copied in from -lbar
d	SHN_OBJT	0x00000000 ;; no type info known

Using binary-compatibility hashes

bar.so symtypehash (in part):

```
sym type          recorded symtypehash (ent_bits 48)
a  SHN_OBJT      0xfa83a824a113
```

foo.so symtypehash (matching portion)

```
sym type          recorded symtypehash (ent_bits 32)
a  SHN_UNDEF     0xfa83a824 ;; copied in from -lbar
```

- ▶ ld.so checks these at runtime, truncating both to the same length if ent_bits is different: if different and neither is 0, the library ABI has changed; invoke a helper binary that uses libctf to look up the actual type via the symtypetab and report the problem.
- ▶ Above, a (0xfa83a824a113) truncated to 32 bits is 0xfa83a824, which matches the recorded value: compatible!
- ▶ ld.so *does not* need to use libctf, do anything but straight comparisons, or even look anything up in a different order than the symbol lookups it already does.

Public header problems

- ▶ Type hashes pay no heed to type opacity: if a type in the public headers depends on an opaque one which changes, so does the public type. We can't even tell which input files will end up being public headers!
- ▶ Maybe: split the job in two!
- ▶ Linker fills out SHN_UNDEF entries from other libs, fills everything else in with 0; *does not hash types at all*.
- ▶ Another tool, run after installation, libgccjits all the relevant headers (provided on the command line), uses their names to constrain the types hashed in, and updates all the 0 entries it can; constrained types are hashed into types they contain as e.g. the string "struct foo"
- ▶ Cure worse than the disease?

Caveats

- ▶ Symbols with no types get empty entries in the table, even if the whole library they came from has no `.ctf` section at all: space-inefficient. One solution: tune `ent_bits` so that the probability of a false negative across all symbols is constant
- ▶ Hashing function, things hashed in, etc unchangeable without ABI breaks

Future work

- ▶ Use in `libabigail` (idea largely invented in chat with Dodji)
- ▶ C++ support (C ABI subset)
- ▶ C/C++ support for LLVM, to help with...
- ▶ ... Rust support
- ▶ ... others?

Thanks

- ▶ Elena Zannoni for API improvements, countless suggestions
- ▶ Indu Bhagat, Jose Marchesi and David Faust for the BTF and CTFv3 generators in GCC (I'll do the v4 one) and for lots of useful critiques
- ▶ Bruce McCulloch for a bunch of useful commits and suggestions
- ▶ Dodji Seketeli for ideas that led to this madness

Links

- ▶ libctf rewrite branch in progress (close to submission):
<https://sourceware.org/git/?p=binutils-gdb.git;a=shortlog;h=refs/heads/users/nalcock/road-to-ctfv4>

The idea in this talk is still vapourware and has no code yet.