Making Write BPF Programs More Pleasant

More Abstraction At Source Level

```
for (auto item : objects) {
   func(item); // processing items
}
```

- Objects can be anything kind of collections, map, array, dynamic allocated array, kernel internal data, etc.
- Explicit parallel loops to show there are no inter-loop dependencies.
- Explicit parallel loops to allow generate explicit special BPF instructions to shorten verification time

- More C++ style abstraction to make dynptr allocation/free implicit at source level?

Macros (1)

- Current tracing programs, including *vmlinux.h*.
- Current networking programs, including a bunch of uapi headers.

Sleftest bind4_prog.c:

```
#include <string.h>
#include <linux/stddef.h>
#include <linux/bpf.h>
#include <linux/in.h>
#include <linux/in6.h>
#include <sys/socket.h>
#include <netinet/tcp.h>
#include <linux/if.h>
#include <errno.h>
…
```

Mostly used for their macros.

In the future, when networking program is added with certain tracing capabilities, it is possible to use vmlinux.h for CO-RE relocation like

```
#include <vmlinux.h>
#include <string.h>
#include <linux/stddef.h>
#include <linux/bpf.h>
#include <linux/in.h>
#include <linux/in6.h>
#include <sys/socket.h>
#include <netinet/tcp.h>
#include <linux/if.h>
#include <errno.h>
```

But Possible redefinition errors!

Macro (2)

- Possible solutions.
- llvm bpf_accept_identical_def attribute to ignore identical type definitions.

```
struct s { int a; } __attribute__((bpf_accept_identical_def));
struct s { int a; }
```

Possible extension to CO-RE based type definitions.

- Encoding macros into dwarf and then convert it a vmlinux_macro.h file and then not including *string.h* etc.
- Possible BTF space overhead ~5MB to encode all macros in elf binary, but would include kernel internal macros as well.