模块十四：eBPF 入门

王炜/前腾讯云 CODING 高级架构师

目录

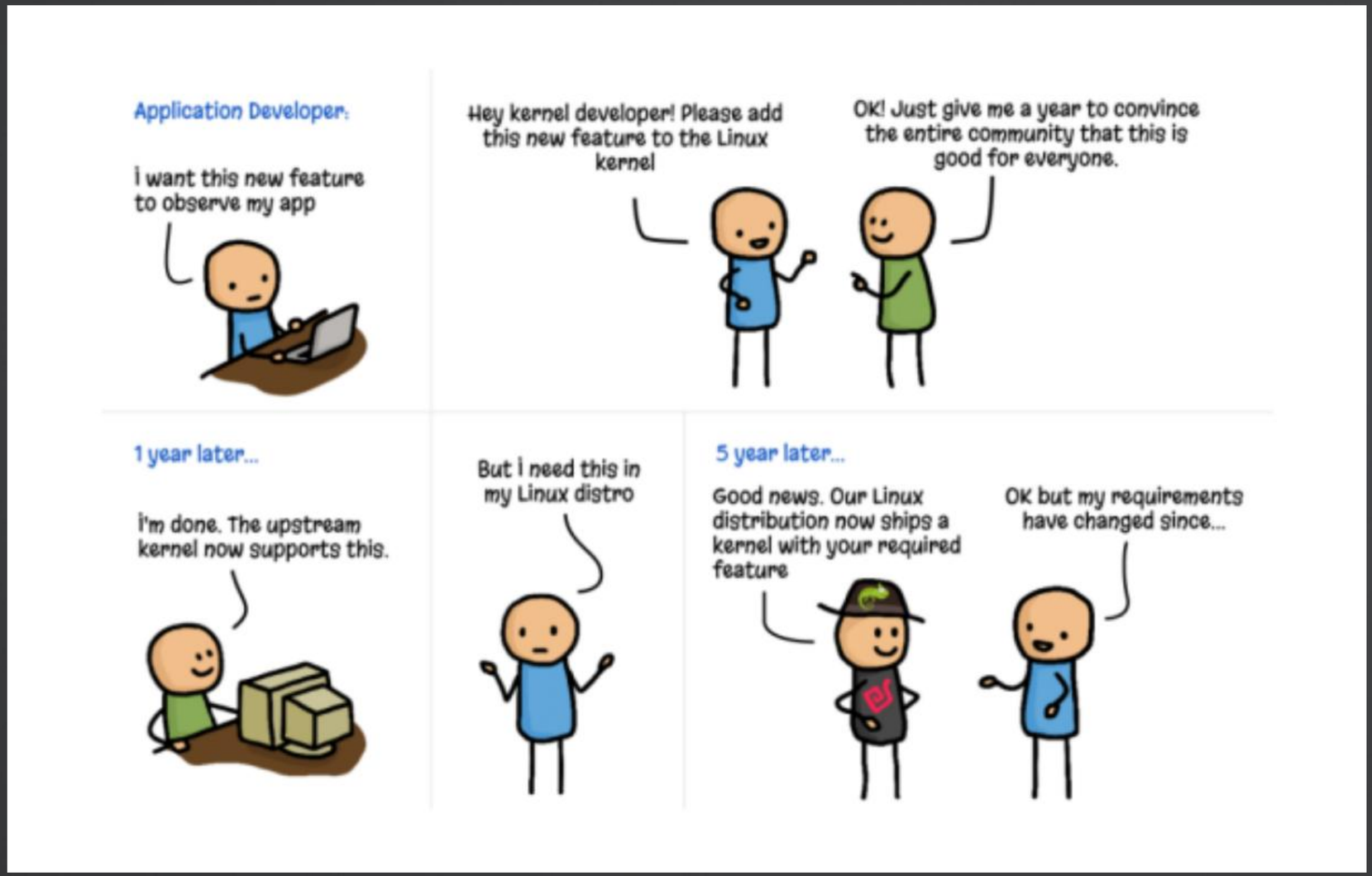
1. 什么是 eBPF

什么是 eBPF

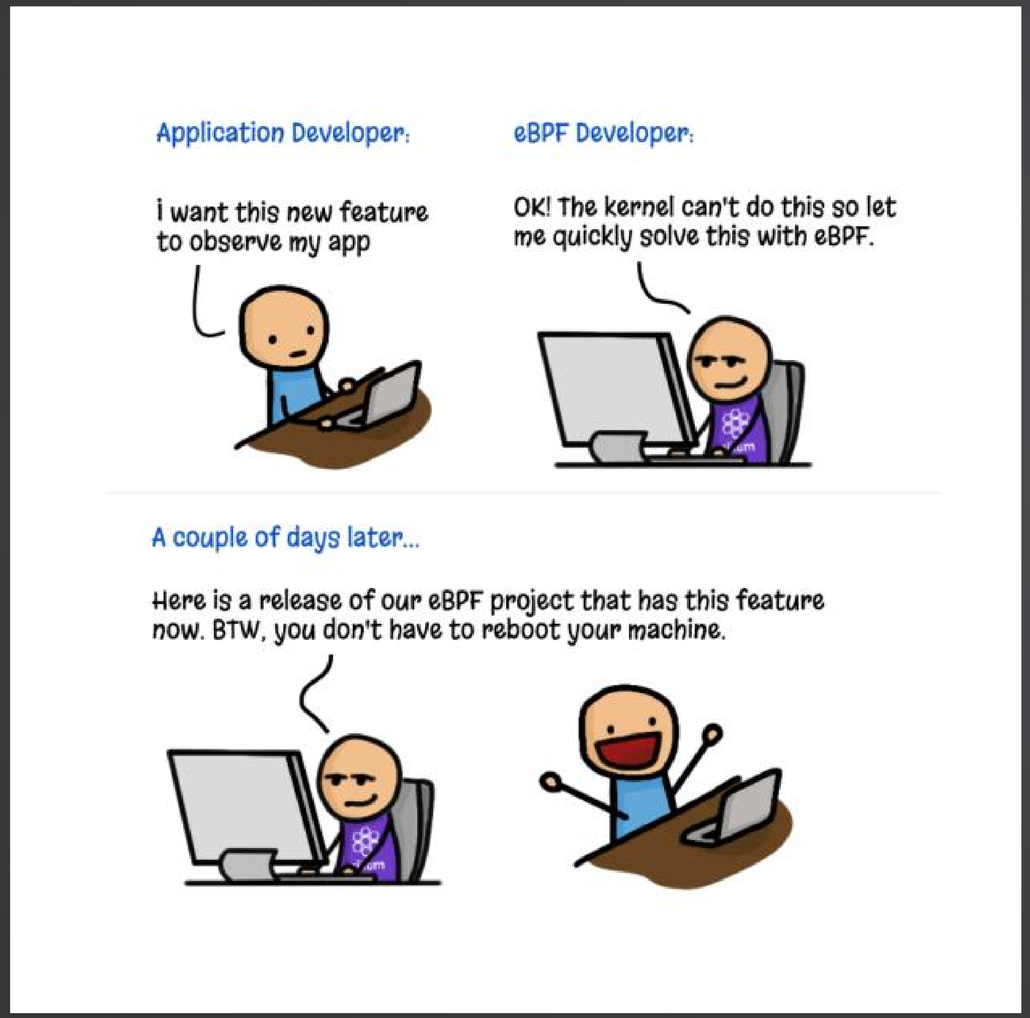
1. extended
2. Berkeley
3. Packet
4. Filter

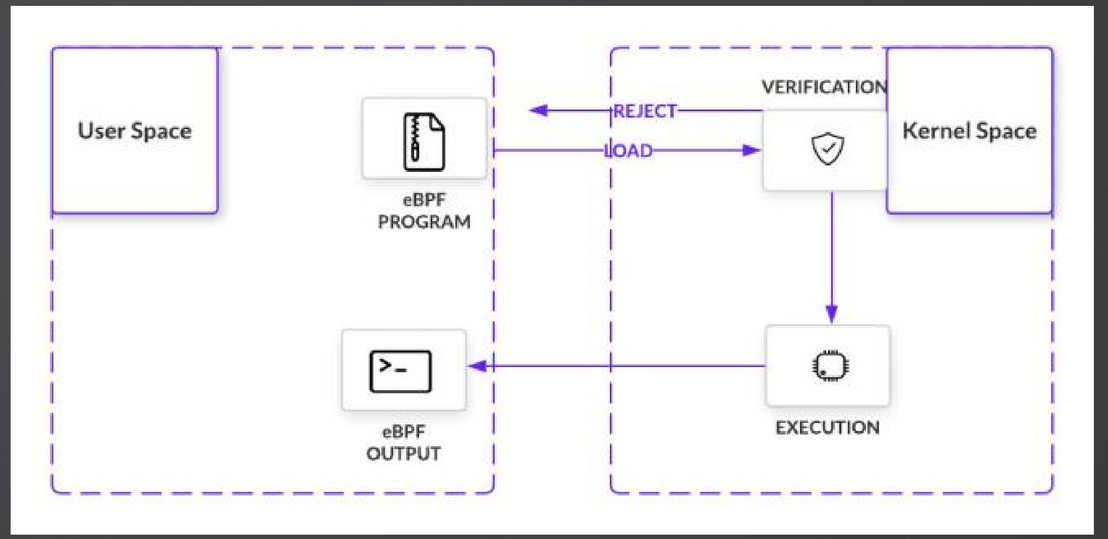
让 Kernel 内核实现可编程化

为什么需要 eBPF


Application Developer:
I want this new feature to observe my app
Hey kernel developer! Please add this new feature to the Linux kernel
OK! Just give me a year to convince the entire community that this is good for everyone.
1 year later...
I'm done. The upstream kernel now supports this.
But I need this in my Linux distro
5 year later...
Good news. Our Linux distribution now ships a kernel with your required feature
OK but my requirements have changed since...

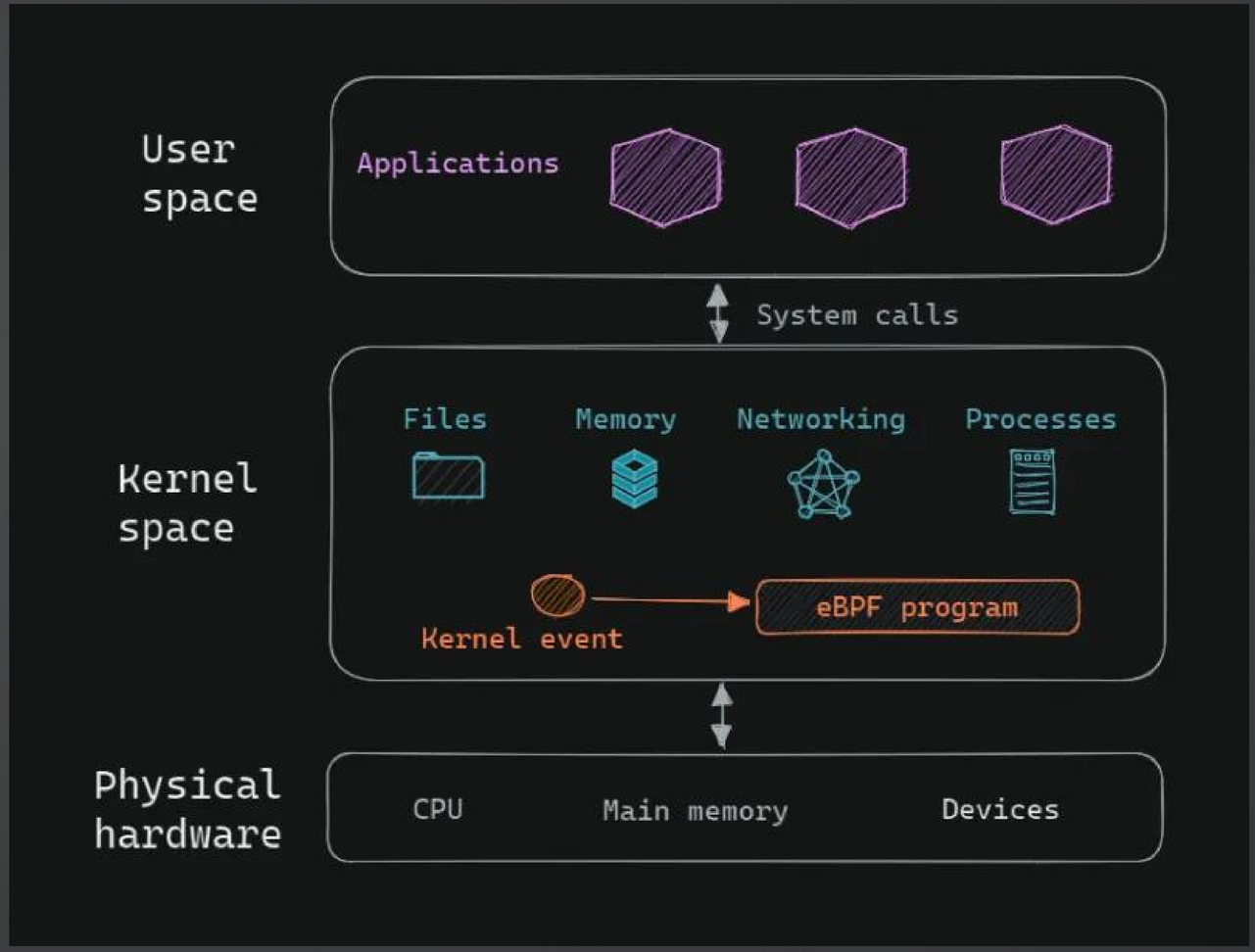
为什么需要 eBPF


Application Developer:
I want this new feature to observe my app
eBPF Developer:
OK! The kernel can't do this so let me quickly solve this with eBPF.
A couple of days later...
Here is a release of our eBPF project that has this feature now. BTW, you don't have to reboot your machine.


User Space
Kernel Space
VERIFICATION
REJECT
LOAD
eBPF PROGRAM
eBPF OUTPUT
EXECUTION

用户空间和内核空间

1. 用户空间是所有应用程序运行的地方
2. 内核空间位于用户空间和物理硬件之间。用户空间中的应用程序无法直接访问硬件，它们会向内核发出系统调用，然后由内核再访问硬件



eBPF 工作原理

	Predetermined/ manual	Dynamic
Kernel	Kernel tracepoints	kprobes
Userspace	USDT	uprobes

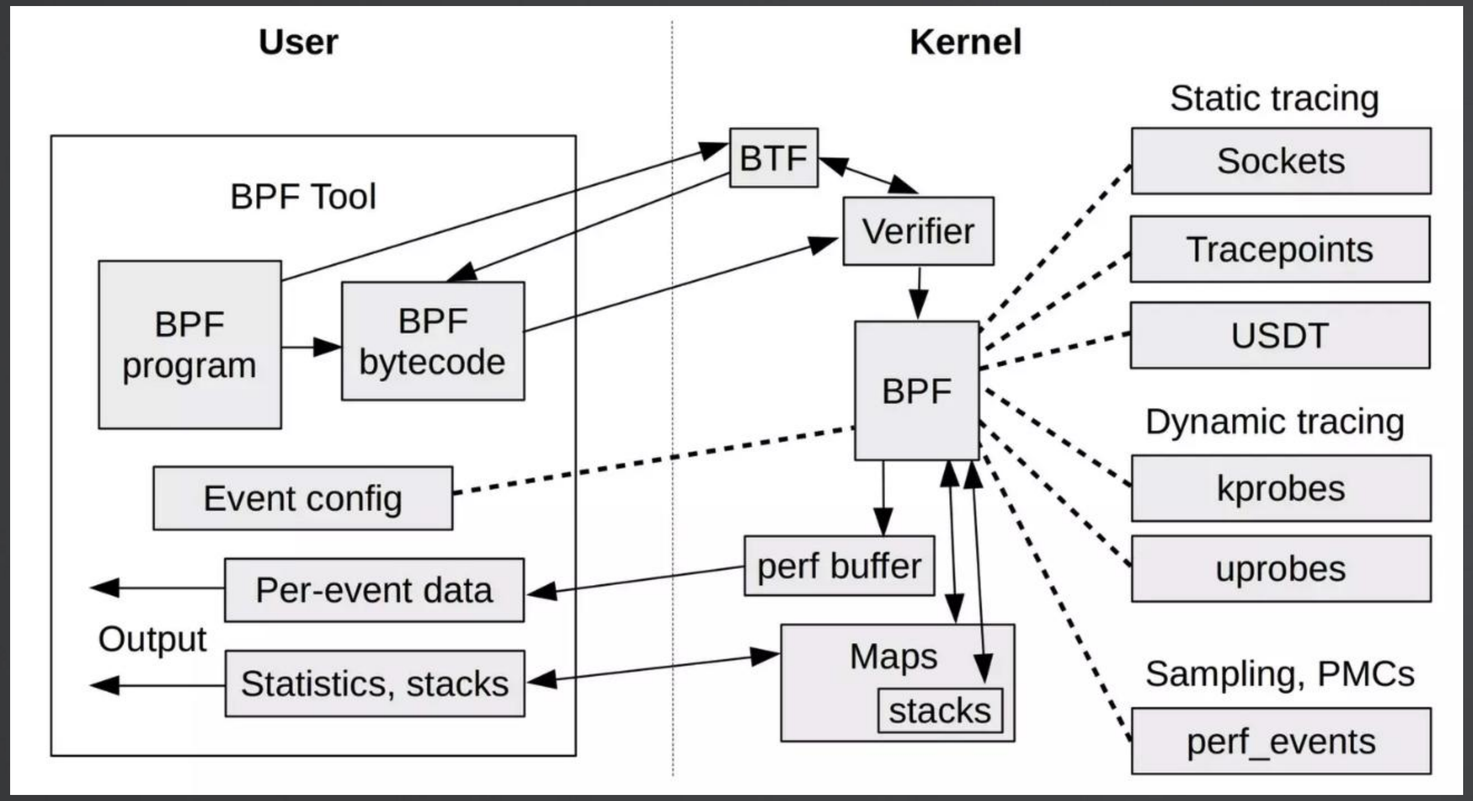
1. 基于事件驱动，在特定事件发生时触发 Hooks，调用 eBPF 程序来获取数据
2. Hooks 可以在用户空间和内核空间，也可以是静态和动态的
3. 静态的 Hooks 一般是内核开发人员预定义的一些事件
4. 动态的 Hooks 比如 kprobes 和 uprobes 是最常用的

eBPF 程序可挂载的事件

- Kprobes：在特定的内核函数开始或结束时挂载 eBPF 程序
  - do_sys_open
- Uprobes：在用户程序特定函数处插入 eBPF 程序
- Tracepoints：Tracepoints 是 Linux 内核中定义的预设点，适合对特定内核事件进行监控
  - sys_enter_read
- Network packets：eBPF 允许挂载在网络事件上，例如处理网络数据包的入口和出口
- Linux Security Module：eBPF 可与 Linux 安全模块 (LSM) 集成，监控和管理安全事件
- Perf event：Perf event 允许 eBPF 程序监听 CPU 或内核中的性能事件，通常用于性能分析
  - cpu-cycles

2. eBPF 技术细节

eBPF 技术总览


User
Kernel
BPF Tool
BPF program
BPF bytecode
Event config
Per-event data
Output
Statistics, stacks
BTF
Verifier
BPF
perf buffer
Maps
stacks
Static tracing
Sockets
Tracepoints
USDT
Dynamic tracing
kprobes
uprobes
Sampling, PMCs
perf_events

技术栈

- AST (Abstract Syntax Tree)：抽象语法树，是编译器在解析源代码时生成的一种数据结构，表示代码的结构。对于eBPF，AST通常是编译器（如LLVM）生成中间代码前的一个步骤。
- LLVM：一种开源编译器基础设施，支持将高层次代码编译为各种平台的机器码。eBPF程序通常用C语言编写，然后通过LLVM编译为BPF字节码。
- IR (Intermediate Representation)：中间表示，是编译器将源代码转换成目标代码之前生成的一种中间格式。对于eBPF，LLVM首先将代码转为IR，再转换为BPF字节码。
- JIT (Just-in-time compilation)：即时编译，在eBPF中是指将BPF字节码动态编译成机器码以提升执行效率。Linux内核支持对eBPF程序进行JIT编译，以提高其在内核中的运行速度。

动态追踪和事件输出的例子

```
bpftrace –e 'kprobe:do_nanosleep {

  printf("PID %d sleeping...\n", pid);

}'

# 输出

PID 10287 sleeping...

PID 2219 sleeping...

......
```

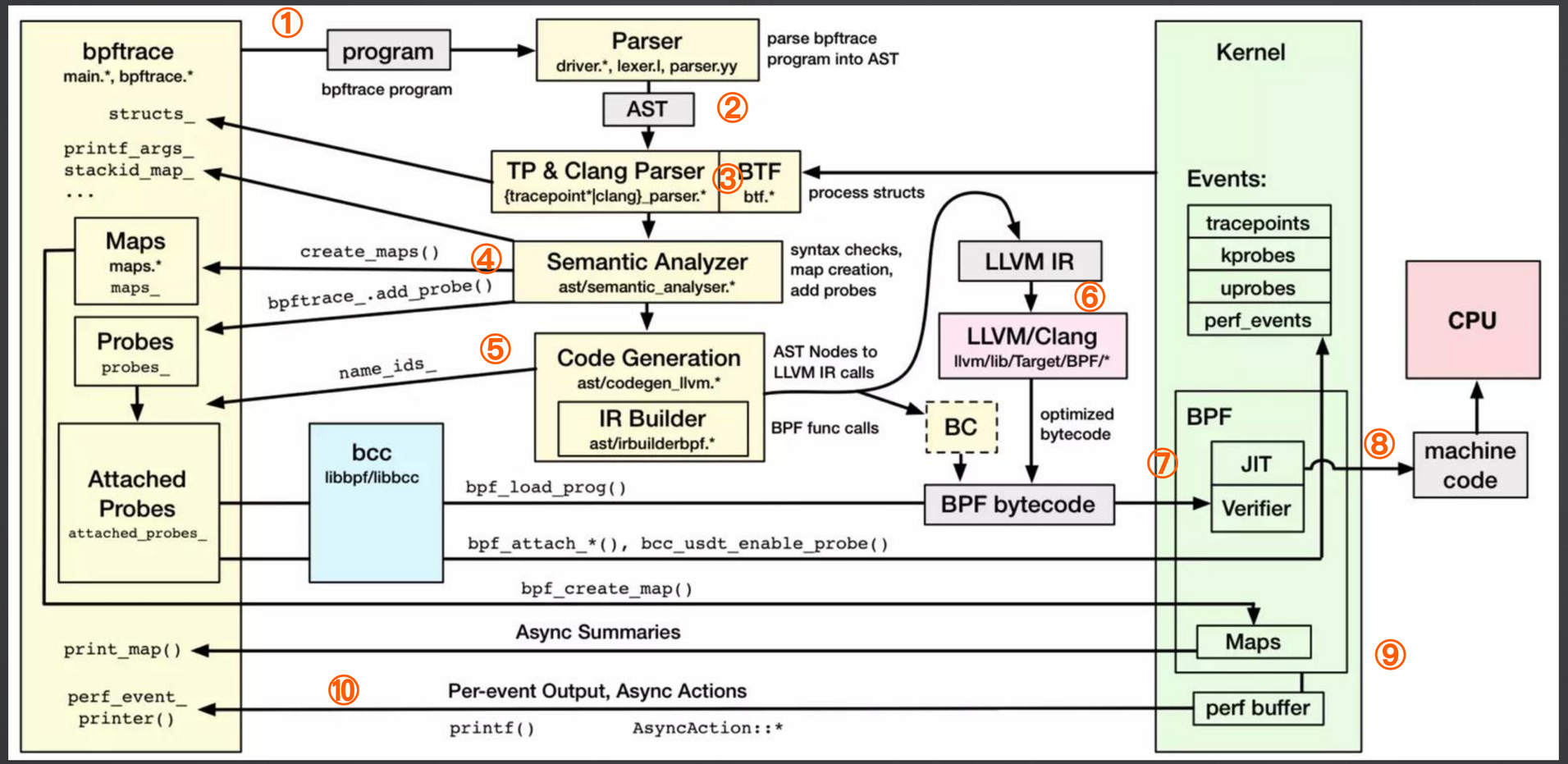
目标

```
bpftrace –e 'kprobe:do_nanosleep {
  printf("PID %d sleeping...\n", pid);
}'
# 输出
PID 10287 sleeping...
PID 2219 sleeping...
.....
```

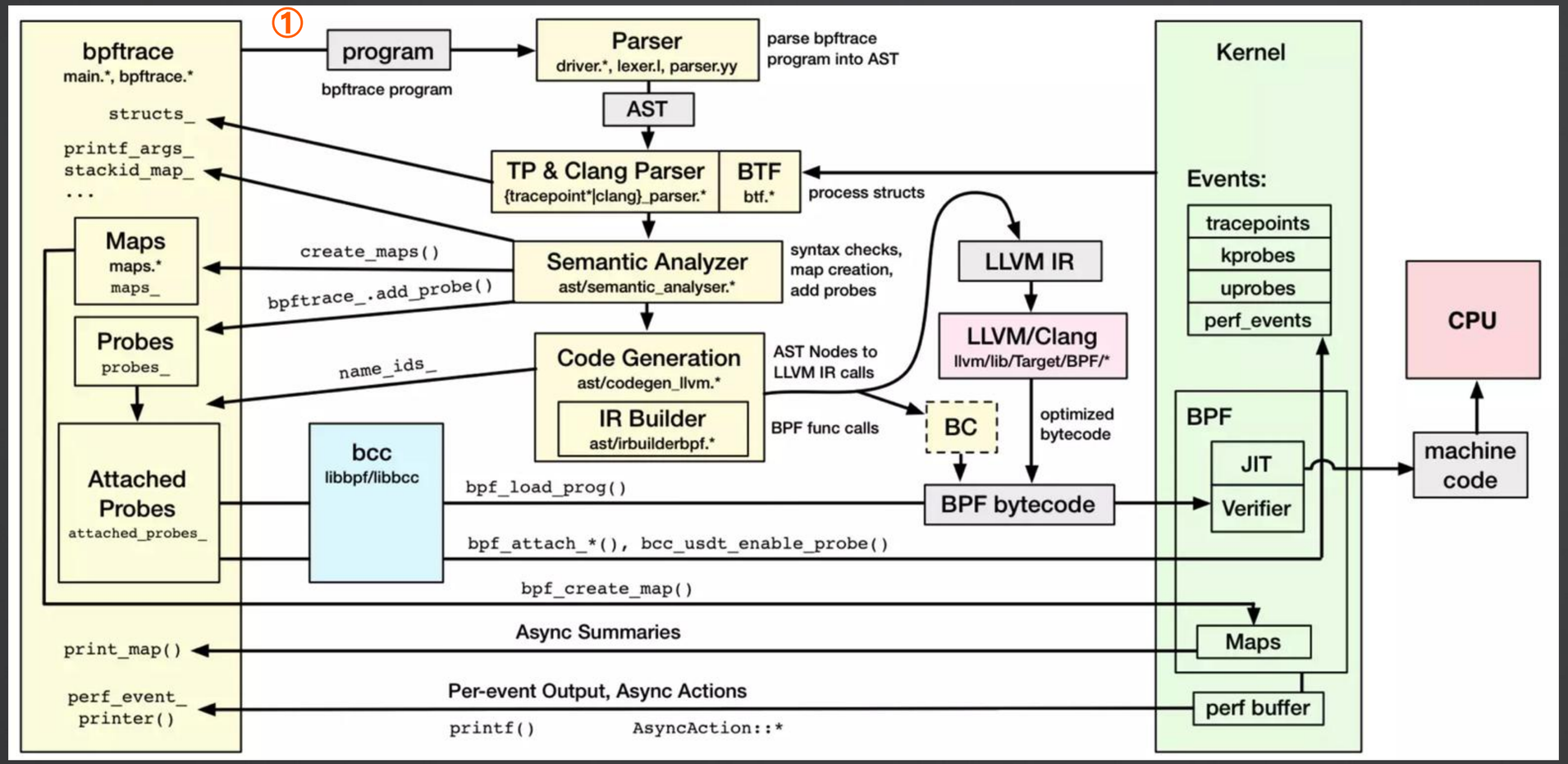
目标：

1. BPF bytecode：将高级语言转化成字节码
2. Kernel events mapped：将内核事件映射到字节码上
3. User space printing events：将内核中的事件传递给用户态程序，并打印相关信息

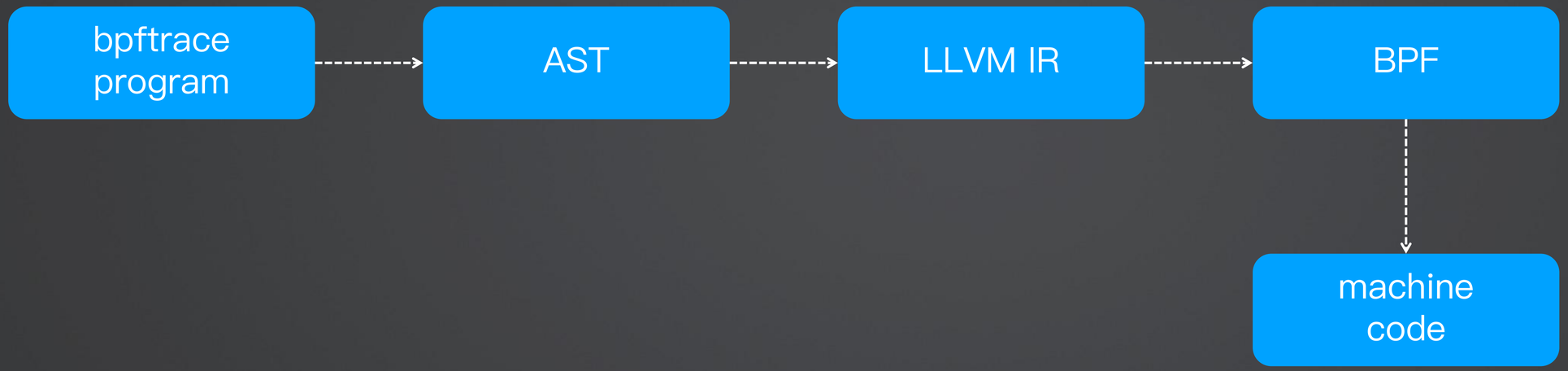
深入 bpftrace 原理



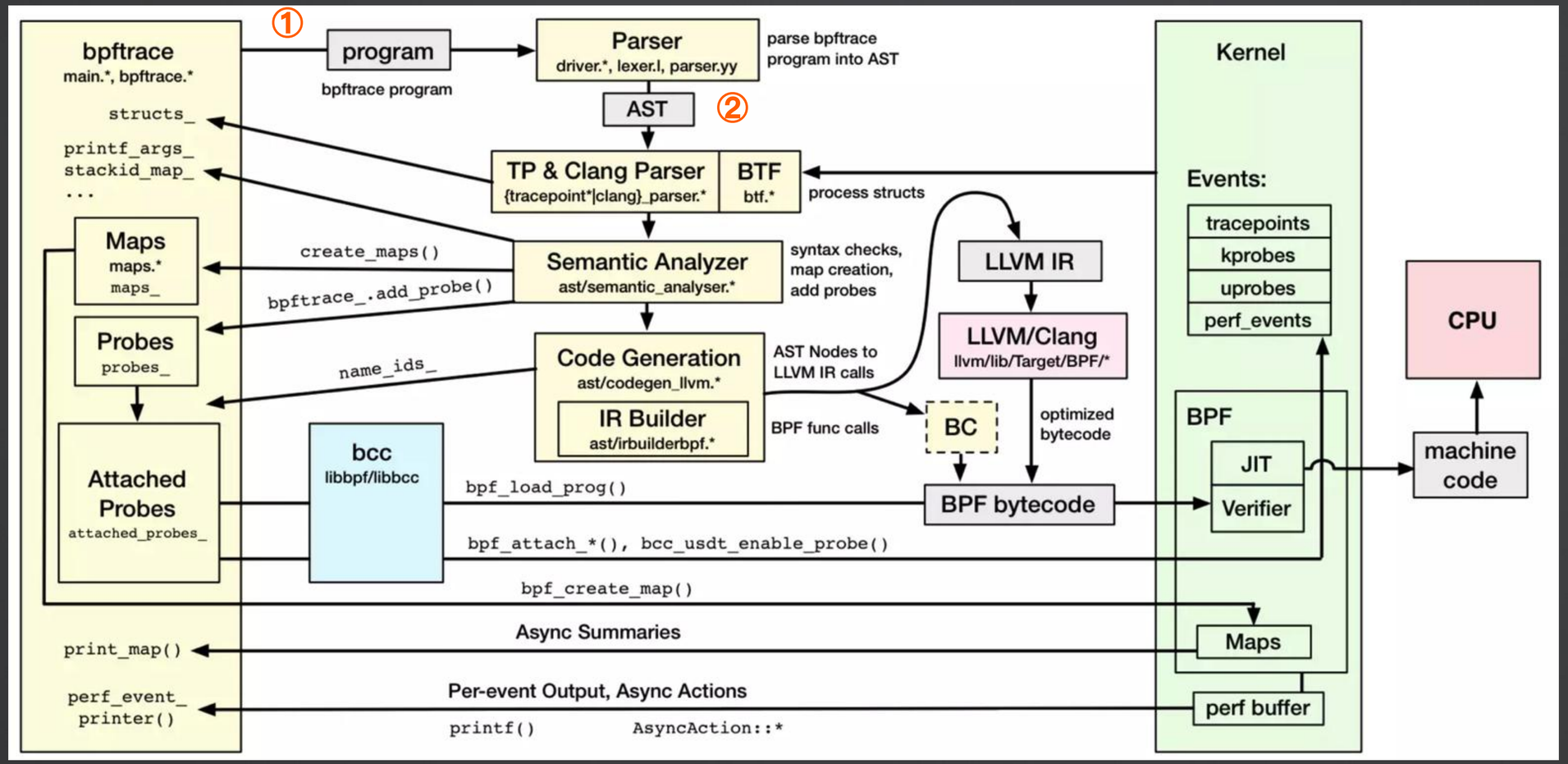
深入 bpftrace 原理



代码的转化



深入 bpftrace 原理

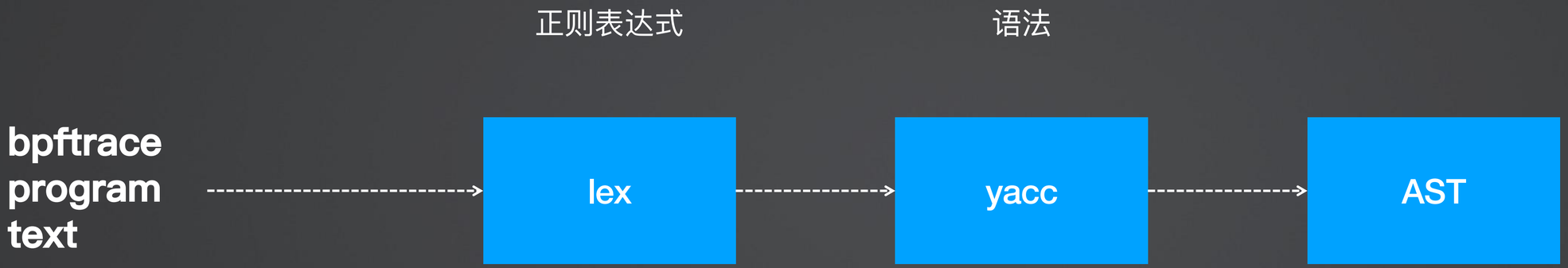


转成 AST

```
kprobe: do_nanosleep {
  printf("PID %d sleeping...\n", pid);
}
```

```
probe(kprobe:do_nanosleep)
    call(printf)
        string(“PID %d sleeping...\n”)
        builtin(pid)
```

AST 解析过程



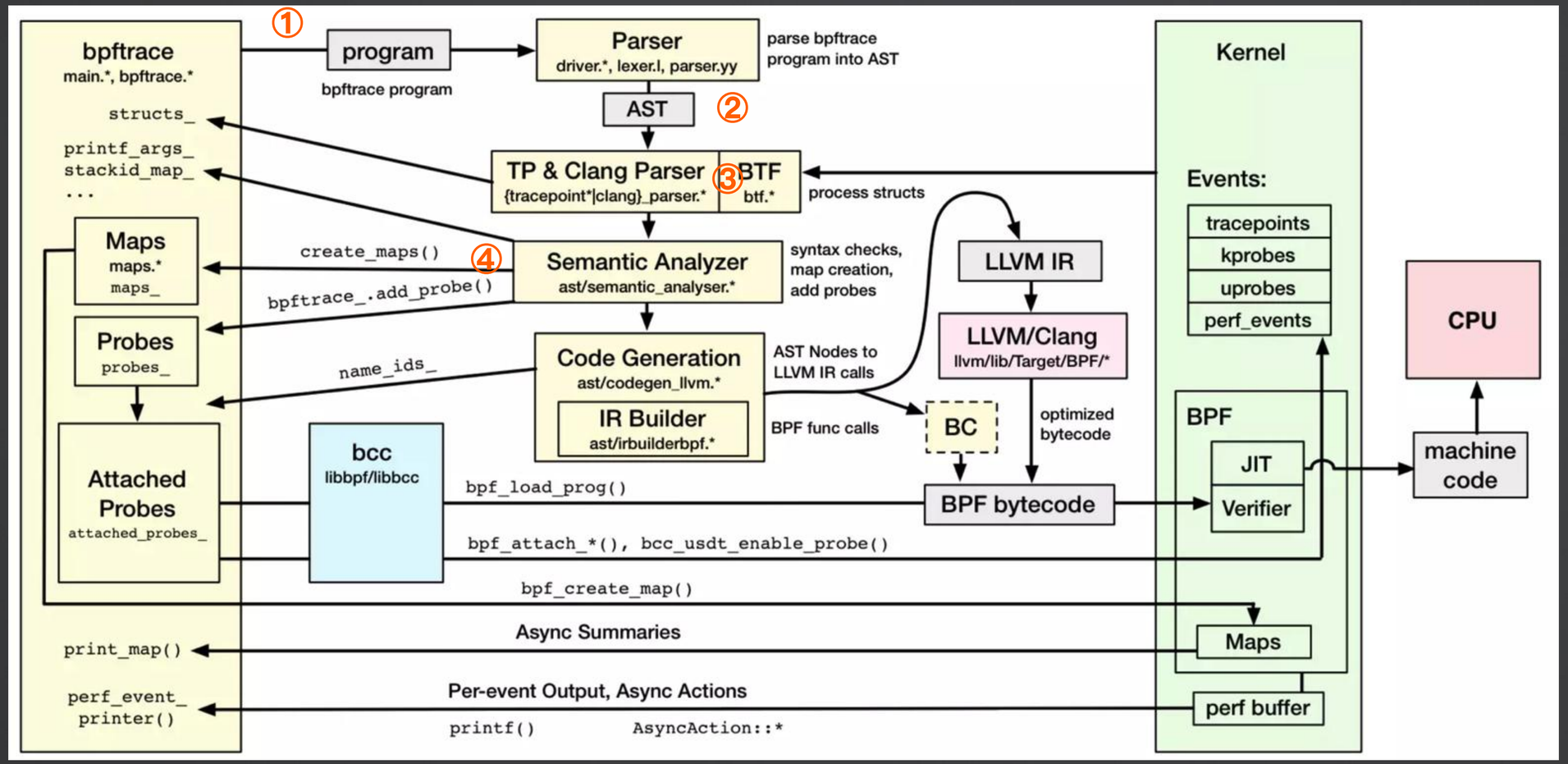
最终解析结果

```
# bpftrace -d -e 'kprobe:do_nanosleep {
    printf("PID %d sleeping...\n", pid);
}'

AST
--------------------
Program
 kprobe:do_nanosleep
  call: printf
   string: PID %d sleeping...\n
   builtin: pid

[...]
```

深入 bpftrace 原理

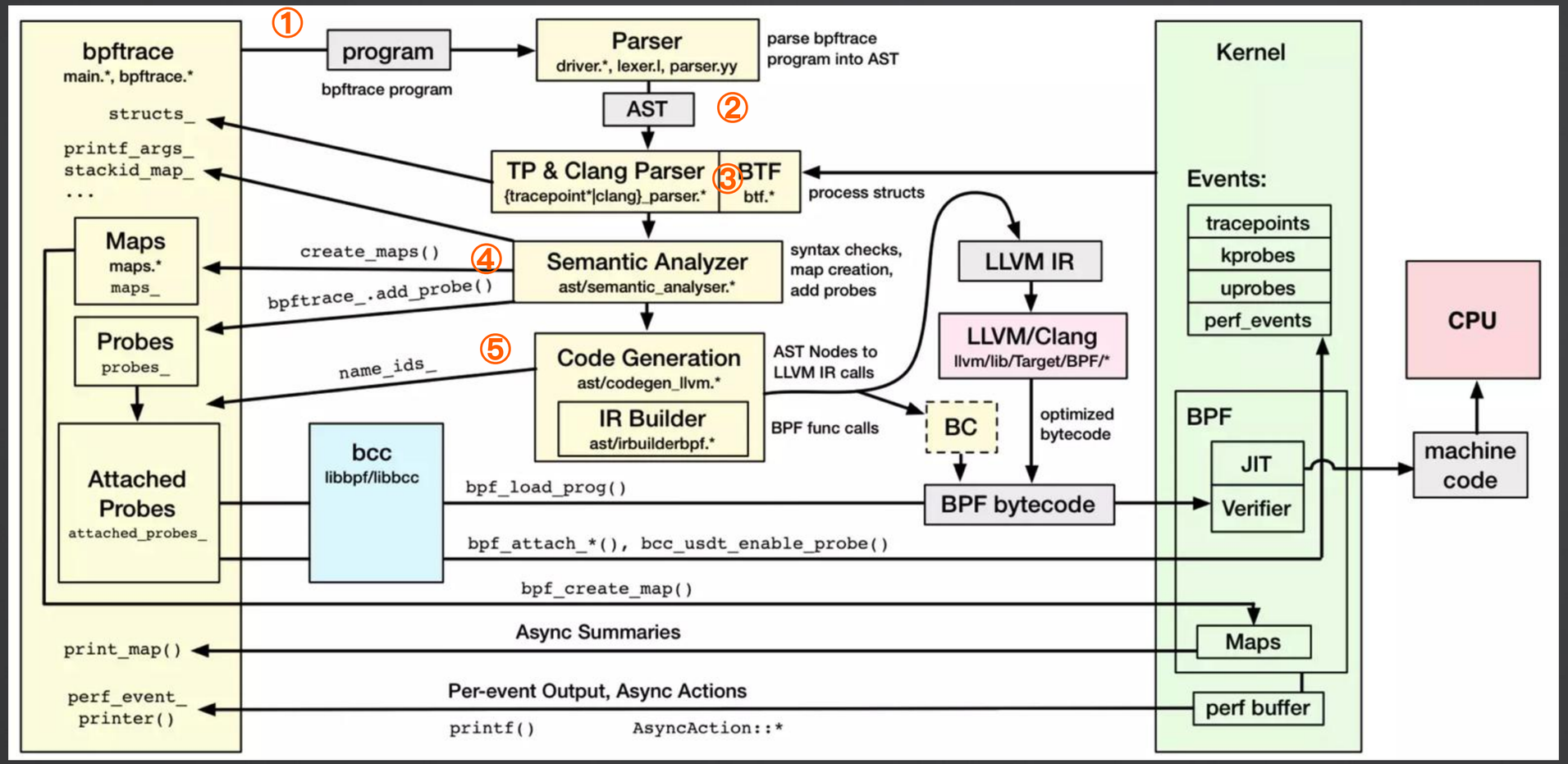


语义分析

```
kprobe: do_nanosleep {
  printf("PID %d sleeping...\n", pidd);
}
```

```
stdin: 2:36-38: ERROR: Unknown identifier: 'pidd'
printf("PID %d sleeping...\n", pidd);
```

深入 bpftrace 原理



AST -> LLVM IR

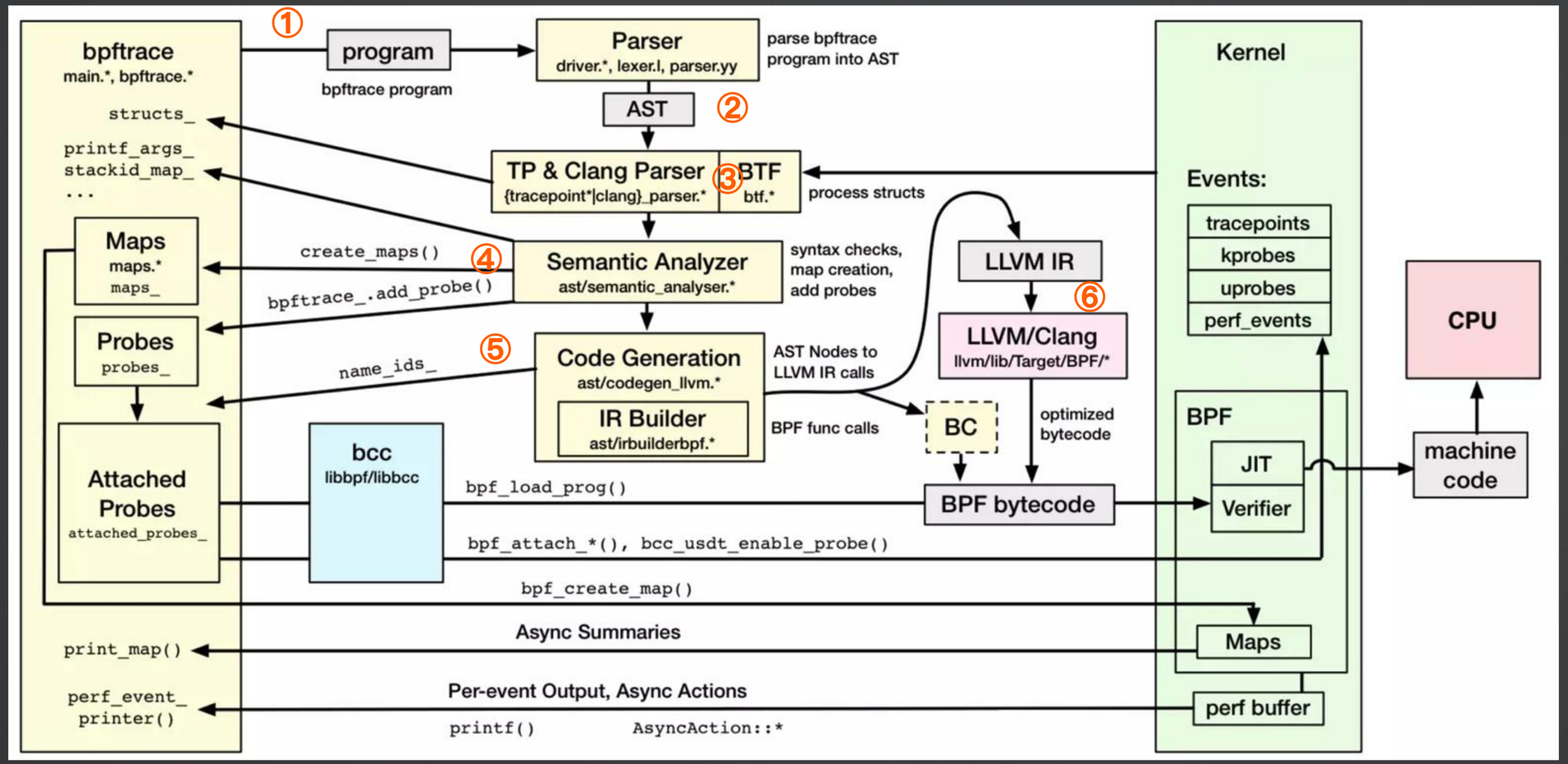
```
void CodegenLLVM::visit(Builtin &builtin)
{
[…]
  else if (builtin.ident == "pid" || builtin.ident == "tid")
  {
    Value *pidtgid = b_.CreateGetPidTgid();
    if (builtin.ident == "pid")
    {
      expr_ = b_.CreateLShr(pidtgid, 32);
```

```
CallInst *IRBuilderBPF::CreateGetPidTgid()
{
  // u64 bpf_get_current_pid_tgid(void)
  // Return: current->tgid << 32 | current->pid
  FunctionType *getpidtgid_func_type = FunctionType::get(getInt64Ty(),
false);
  PointerType *getpidtgid_func_ptr_type =
PointerType::get(getpidtgid_func_type, 0);
  Constant *getpidtgid_func = ConstantExpr::getCast(
      Instruction::IntToPtr,
      getInt64(libbpf::BPF_FUNC_get_current_pid_tgid),
```

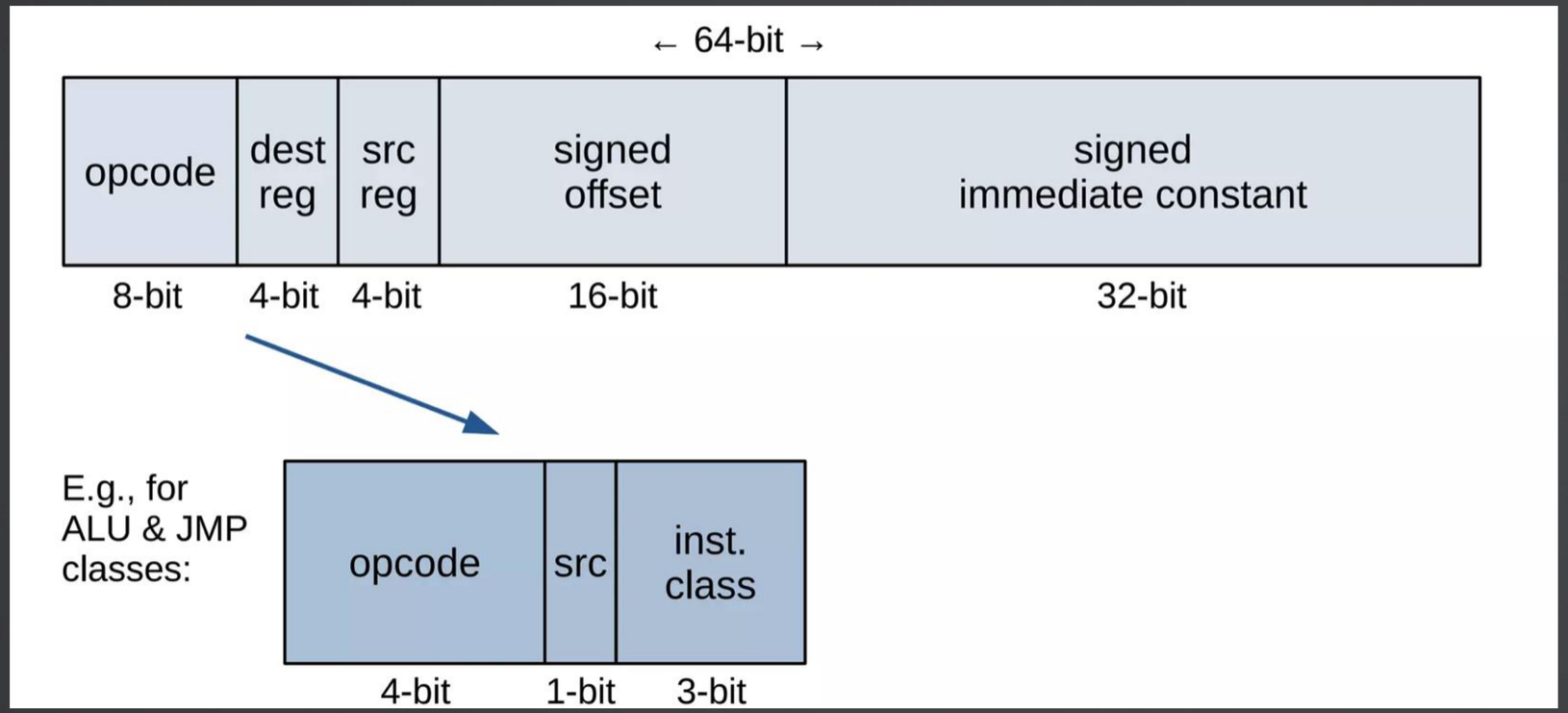
转化结果 LLVM IR

```
# bpftrace -d -e 'kprobe:do_nanosleep {
    printf("PID %d sleeping...\n", pid); }'
[…]
define i64 @"kprobe:do_nanosleep"(i8*) local_unnamed_addr section
"s_kprobe:do_nanosleep_1" {
entry:
  %printf_args = alloca %printf_t, align 8
  %1 = bitcast %printf_t* %printf_args to i8*
  call void @llvm.lifetime.start.p0i8(i64 -1, i8* nonnull %1)
  %2 = getelementptr inbounds %printf_t, %printf_t* %printf_args, i64 0, i32 0
  store i64 0, i64* %2, align 8
  %get_pid_tgid = tail call i64 inttoptr (i64 14 to i64 ()*)()
  %3 = lshr i64 %get_pid_tgid, 32
  %4 = getelementptr inbounds %printf_t, %printf_t* %printf_args, i64 0, i32 1
  store i64 %3, i64* %4, align 8
  %pseudo = tail call i64 @llvm.bpf.pseudo(i64 1, i64 1)
  %get_cpu_id = tail call i64 inttoptr (i64 8 to i64 ()*)()
  %perf_event_output = call i64 inttoptr (i64 25 to i64 (i8*, i64, i64, %printf_t*,
i64)*)(i8* %0, i64 %pseudo, i64 %get_cpu_id, %printf_t* nonnull %printf_args, i64 16)
  call void @llvm.lifetime.end.p0i8(i64 -1, i8* nonnull %1)
  ret i64 0
}
```

深入 bpftrace 原理



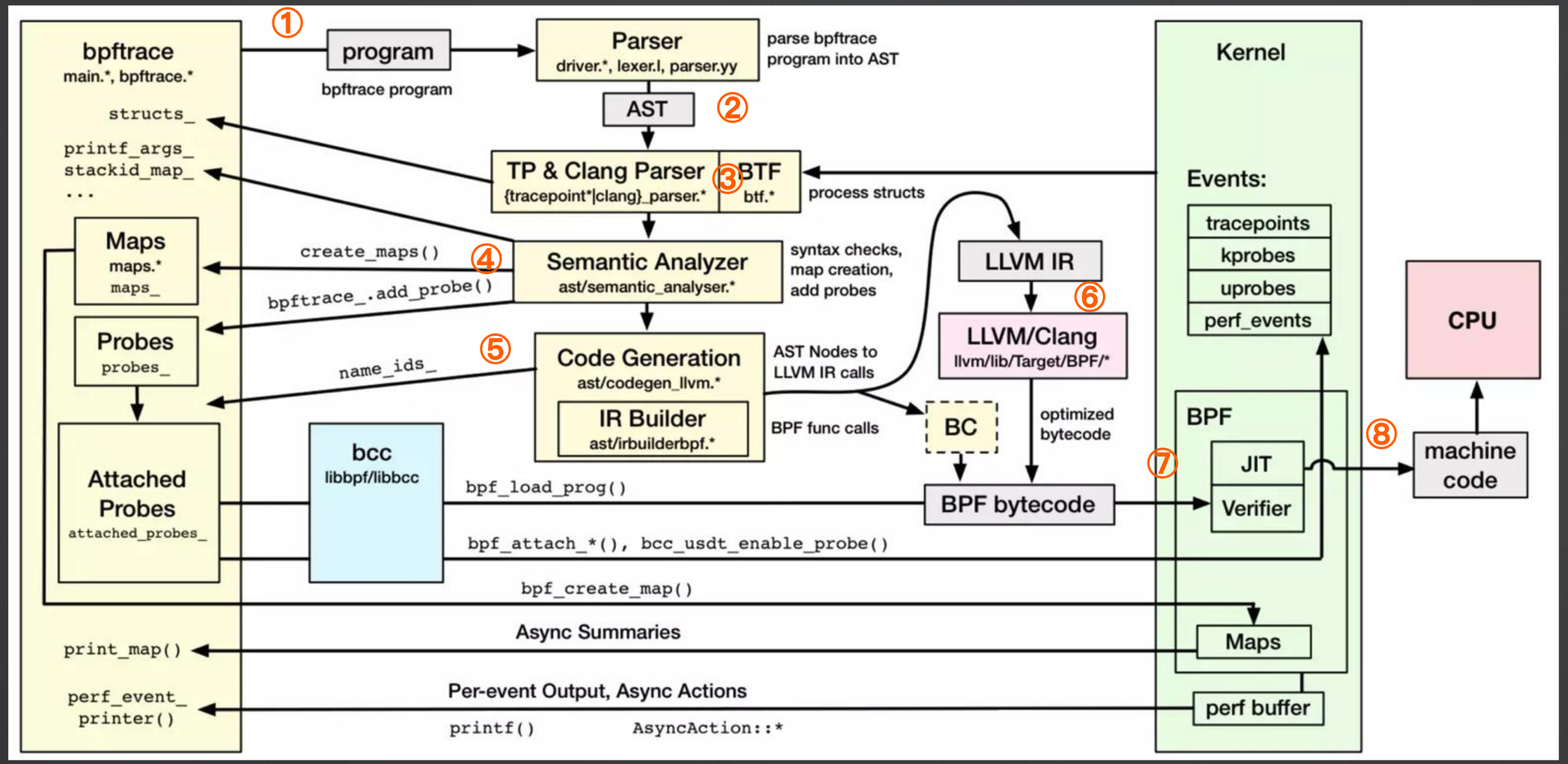
eBPF bytecode 格式



eBPF bytecode 格式

```
bf 16 00 00 00 00 00 00
b7 01 00 00 00 00 00 00
7b 1a f0 ff 00 00 00 00
85 00 00 00 0e 00 00 00
77 00 00 00 20 00 00 00
7b 0a f8 ff 00 00 00 00
18 17 00 00 30 00 00 00 00 00 00 00 00 00 00 00 00
85 00 00 00 08 00 00 00
bf a4 00 00 00 00 00 00
07 04 00 00 f0 ff ff ff
bf 61 00 00 00 00 00 00
bf 72 00 00 00 00 00 00
bf 03 00 00 00 00 00 00
b7 05 00 00 10 00 00 00
85 00 00 00 19 00 00 00
b7 00 00 00 00 00 00 00
95 00 00 00 00 00 00 00
```

深入 bpftrace 原理



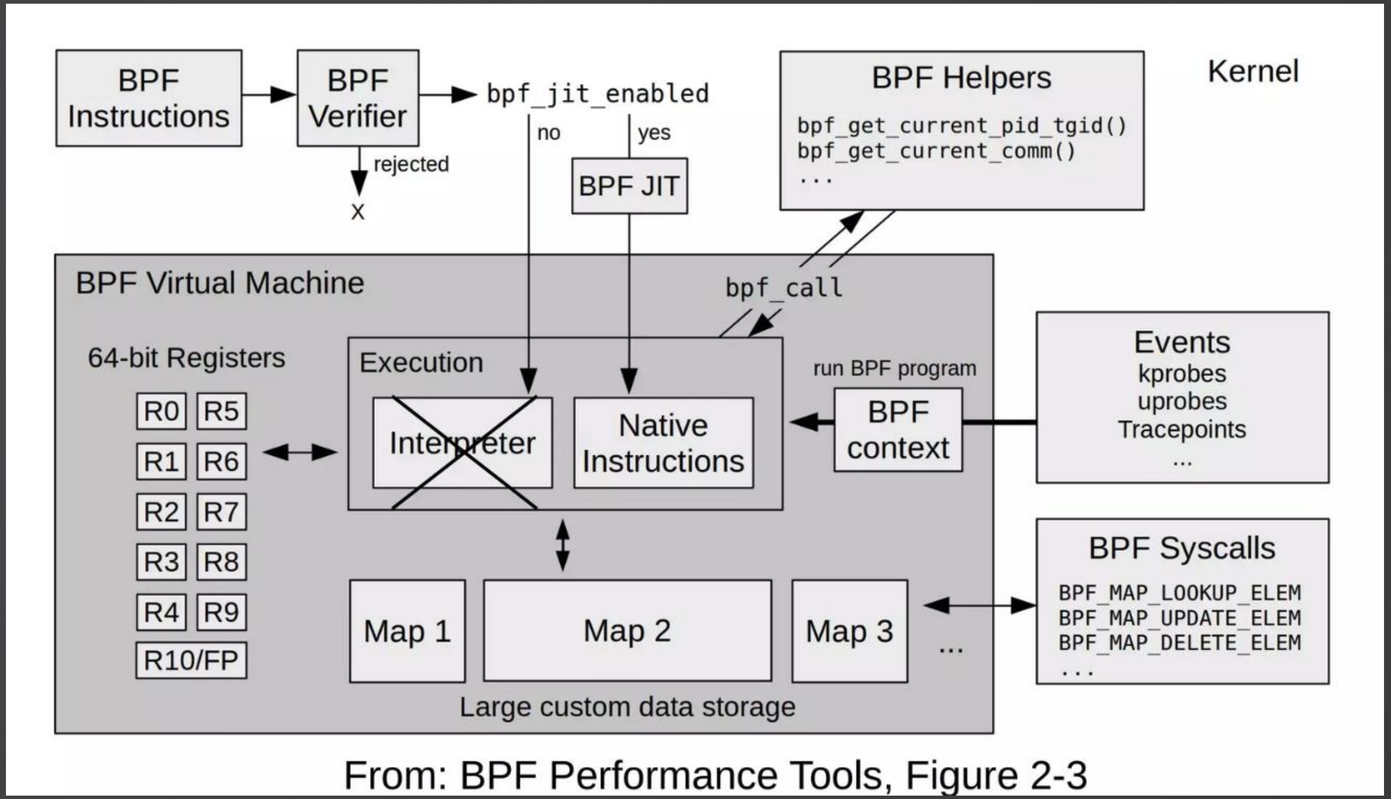
eBPF bytecode 验证和执行过程

验证器：

- 无限循环
- 非法内存访问（越界或对内核关键内存的写入）
- 未初始化的寄存器或变量使用

执行器

- JIT 路径：如果 bpf_jit_enabled 设置为 yes，BPF JIT 将 bytecode 编译为本地机器指令
- 解释器路径：未启用 JIT，则将由内核的 BPF 解释器逐条解释执行



From: BPF Performance Tools, Figure 2-3

验证器

- > 9000 行代码
- > 260 错误

```
check_subprogs                  check_helper_mem_access
check_reg_arg                   check_func_arg
check_stack_write               check_map_func_compatibility
check_stack_read                check_func_proto
check_stack_access              check_func_call
check_map_access_type           check_reference_leak
check_mem_region_access         check_helper_call
check_map_access                check_alu_op
check_packet_access             check_cond_jmp_op
check_ctx_access                check_ld_imm
check_flow_keys_access          check_ld_abs
check_sock_access               check_return_code
check_pkt_ptr_alignment         check_cfg
check_generic_ptr_alignment     check_btf_func
check_ptr_alignment             check_btf_line
check_max_stack_depth           check_btf_info
check_tp_buffer_access          check_map_prealloc
check_ptr_to_btf_access         check_map_prog_compatibility
check_mem_access                check_struct_ops_btf_id
check_xadd                      check_attach_modify_return
check_stack_boundary            check_attach_btf_id
```

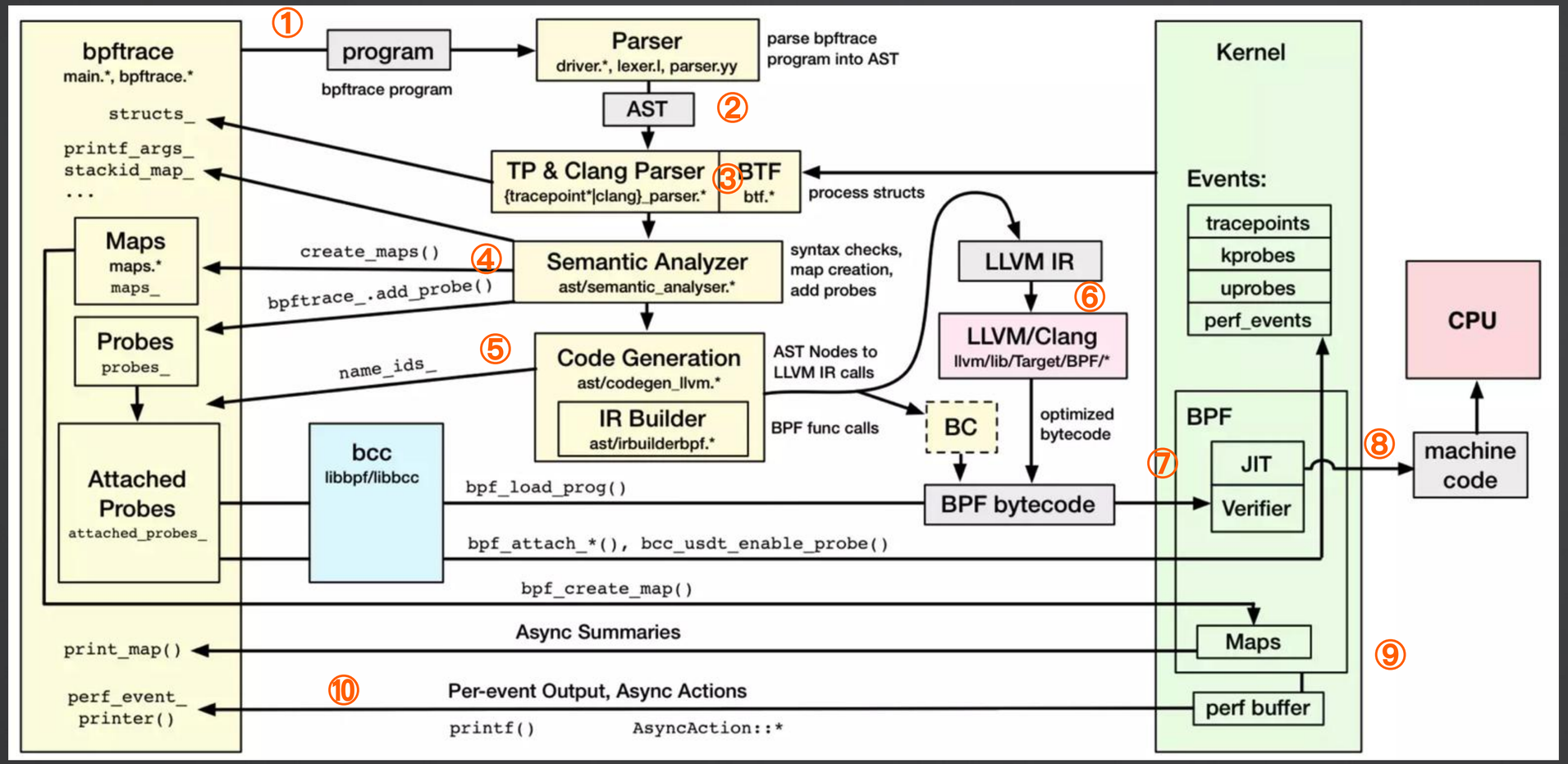
验证器细节

- 内存访问
  - 直接访问受到严格限制
  - 只能读取已初始化的内存
  - 内核内存访问必须通过 bpf_probe_read() 助手函数及其检查
- 参数类型检查
  - 函数的参数必须严格匹配指定的类型
  - 任何类型不匹配的函数调用都会导致 eBPF 程序在验证阶段被拒绝
- 寄存器使用检查
  - 禁止写入帧指针寄存器
- 没有写入溢出
  - 例如在使用 BPF Maps 时， eBPF 程序不能写入超出 Map 定义的数据结构

转化成机器码

```
# bpftool prog dump jited id 80 opcodes | grep -v :
    55
    48 89 e5
    48 81 ec 10 00 00 00
    53
    41 55
    41 56
    41 57
    6a 00
    48 89 fb
    31 ff
    48 89 7d f0
    e8 a0 8b 44 c2
    48 c1 e8 20
    48 89 45 f8
    49 bd 00 b6 7b b8 3a 9d ff ff
    e8 a9 8a 44 c2
    48 89 e9
    48 83 c1 f0
    48 89 df
[...]                                    31 instructions
```

深入 bpftrace 原理



3. eBPF Map

eBPF Map

- 用于存储不同数据的通用结构
- 允许在以下对象之间共享数据
  - eBPF 内核程序
  - 内核和用户空间
- 每一个 Maps 都有以下属性
  - 类型
  - 元素最大长度
  - Key 大小（bytes）
  - Values 大小（bytes）

```
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC,  /* Reserve 0 as invalid map type */
    BPF_MAP_TYPE_HASH,
    BPF_MAP_TYPE_ARRAY,
    BPF_MAP_TYPE_PROG_ARRAY,
    BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    BPF_MAP_TYPE_PERCPU_HASH,
    BPF_MAP_TYPE_PERCPU_ARRAY,
    BPF_MAP_TYPE_STACK_TRACE,
    BPF_MAP_TYPE_CGROUP_ARRAY,
    BPF_MAP_TYPE_LRU_HASH,
    BPF_MAP_TYPE_LRU_PERCPU_HASH,
    BPF_MAP_TYPE_LPM_TRIE,
    BPF_MAP_TYPE_ARRAY_OF_MAPS,
    BPF_MAP_TYPE_HASH_OF_MAPS,
    BPF_MAP_TYPE_DEVMAP,
    BPF_MAP_TYPE_SOCKMAP,
    BPF_MAP_TYPE_CPUMAP,
};
```

Hash Map

- Hash Map（哈希表）
  - 类似哈希表，支持高效的查找和更新
  - 最常用的 Map 类型之一
- 使用场景
  - 可以用来记录每个用户调用某个系统调用的次数，键为用户 ID，值为次数。

```
// 定义哈希表 BPF Map
struct bpf_map_def SEC("maps") my_hash = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(int),     // 键的大小
    .value_size = sizeof(int),   // 值的大小
    .max_entries = 128,          // 表的最大元素数量
};
```

Array Map

- Array Map（数组）
  - 固定大小
  - 通过索引快速读取数据，适合常规的计数器、缓存等
- 使用场景
  - 可以用来存储一组 CPU 核心的负载信息，使用数组索引来访问和更新每个 CPU 数据

```
// 定义数组 BPF Map
struct bpf_map_def SEC("maps") my_array = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(int),    // 键为数组下标，类型是 int
    .value_size = sizeof(int),  // 数组值的类型也是 int
    .max_entries = 128,         // 数组的最大大小
};
```

Per-CPU Hash/Array Map

- Per-CPU Hash/Array Map（基于 CPU 的 Map）
  - 和 Hash 或 Array 相似，每个 CPU 拥有独立的数据，不会造成竞争冲突
  - 在多核系统中性能表现更好
- 使用场景
  - 高并发场景中，用于统计每个 CPU 的独立数据，比如每个 CPU 的网络数据包计数

```
// 定义 Per-CPU哈希表
struct bpf_map_def SEC("maps") my_per_cpu_hash = {
    .type = BPF_MAP_TYPE_PERCPU_HASH,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 128,
};
```

LPM Trie

- LPM Trie（最长前缀匹配前缀树）
  - 适用于需要根据前缀做快速匹配的场景
  - 常见于数据包处理和网络性能优化中
- 使用场景
  - 用于存储和查找基于 IP 地址前缀的路由记录，常用于网络流量引导

```
// 定义 LPM Trie 结构 BPF Map
struct bpf_map_def SEC("maps") my_lpm_trie = {
    .type = BPF_MAP_TYPE_LPM_TRIE,
    .key_size = sizeof(struct { u32 prefixlen; u32 ip; }), // IP前缀结构
    .value_size = sizeof(int),
    .max_entries = 128,
    .map_flags = BPF_F_NO_PREALLOC,  // 不预分配
};
```

Queue/Stack Map

- Queue/Stack Map（队列/栈）
  - 支持队列或栈操作，适合分段处理数据流
- 使用场景
  - 用于监控系统事件流，将事件按顺序存储，并处理/清除旧事件

```
// 定义一个 Queue BPF Map
struct bpf_map_def SEC("maps") my_queue = {
    .type = BPF_MAP_TYPE_QUEUE,
    .key_size = 0,
    .value_size = sizeof(int),
    .max_entries = 64,
};
```

Ring Buffer Map

- Ring Buffer Map（环形缓冲区）
  - 数据被循环存储在缓冲区中，用于收集和实时发送 eBPF 事件到用户空间
  - 高效的事件推送机制，特别适用于高频但只需要短暂存储的数据流
- 使用场景
  - 用于实时跟踪进程上下文切换、网络包流入、日志等高频事件

```
// 定义一个环形缓冲区
struct bpf_map_def SEC("maps") my_ringbuf = {
    .type = BPF_MAP_TYPE_RINGBUF,
    .key_size = 0,
    .value_size = sizeof(int),
    .max_entries = 4096,
};
```

LRU Hash Map

- LRU Hash Map（基于 LRU 的哈希表）
  - 带有自动淘汰功能的哈希表
  - 当 Map 达到最大容量时，它会自动删除最久未使用的数据以腾出新空间
- 使用场景
  - 适合需要有限存储资源的场景，例如 DNS 缓存、session 缓存
  - 适用于流量监控或网络跟踪设备，将最近频繁使用的 IP 或 MAC 地址保存在哈希表中

```
// 定义 LRU 哈希表
struct bpf_map_def SEC("maps") my_lru_hash = {
    .type = BPF_MAP_TYPE_LRU_HASH,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 128,          // 最大支持128个条目
};
```

LRU Per-CPU Hash Map

- LRU Per-CPU Hash Map（基于 CPU 的 LRU Map）
  - 每个 CPU 都有自己独立的副本
  - 多 CPU 更有优势，避免了竞争条件，性能更加优化
- 使用场景
  - 适合需要有限存储资源的场景，例如 DNS 缓存、session 缓存
  - 适用于流量监控或网络跟踪设备，将最近频繁使用的 IP 或 MAC 地址保存在哈希表中

```
// 定义 LRU Per-CPU哈希表
struct bpf_map_def SEC("maps") my_lru_percpu_hash = {
    .type = BPF_MAP_TYPE_LRU_PERCPU_HASH,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 128,     // 每个 CPU 都有128个条目的独立副本
};
```

ARRAY_OF_MAPS

- ARRAY_OF_MAPS（Map 数组）
  - 与普通的 ARRAY map 思路类似，但存储的数据类型是 map 的引用
- 使用场景
  - 动态管理或切换多个 Maps，在多个数据结构之间进行切换和管理
  - 网络流量分析中可以动态选择不同的 Maps 进行跟踪（基于时间段、网络分区等）

```
// 定义一个 Array of Maps
struct bpf_map_def SEC("maps") my_array_of_maps = {
    .type = BPF_MAP_TYPE_ARRAY_OF_MAPS,  // 定义这是一个 Map 数组
    .key_size = sizeof(int),             // 数组索引类型，这里是 int
    .max_entries = 4,                    // 数组具有 4 个元素
    .inner_map_fd = bpf_map_fd(my_map),  // 每个数组元素是 another_map 的引用
};
```

HASH_OF_MAPS

- HASH_OF_MAPS（Map 的哈希表）
  - 跟普通 Hash Map 类似，但不同的是存储的是其他 map 的引用
- 使用场景
  - 层次化数据存储
  - 分组或动态管理多个 Map

```
// 定义一个 Hash of Maps
struct bpf_map_def SEC("maps") my_hash_of_maps = {
    .type = BPF_MAP_TYPE_HASH_OF_MAPS,   // 定义这是一个 Hash of Maps
    .key_size = sizeof(int),             // 键类型，这里是 int
    .max_entries = 128,                  // 总共有 128 个元素
    .inner_map_fd = bpf_map_fd(my_map),  // 内部 map 的引用句柄
};
```

DEVICE_MAP

- DEVICE_MAP（设备映射）
  - 允许 eBPF 程序直接影响和控制特定网络设备的数据包转发行为
  - **主要用于 XDP（eXpress Data Path） 场景**
- 使用场景
  - 通过 eBPF + XDP，提供了极快速的数据包转发、过滤和丢弃方案
  - 比如将从一个网络接口接收的数据包直接重新发送到另一个接口

```
// 定义一个 DEVICE_MAP
struct bpf_map_def SEC("maps") my_dev_map = {
    .type = BPF_MAP_TYPE_DEVMAP,          // 设备映射类型
    .key_size = sizeof(int),              // 设备索引
    .value_size = sizeof(int),            // 设备 ID（网络设备接口）
    .max_entries = 128,                   // 最大支持 128 个不同设备映射
};
```

SOCKET_MAP

- SOCKET_MAP（套接字映射）
  - 用于存储 Socket
  - **允许 eBPF 程序与 Socket 直接交互，可以对 TCP/UDP 套接字进行操作，例如转发某些应用数据流或者修改其行为**
- 使用场景
  - 负载均衡和 TCP 流管理：将不同的数据流标记到不同的 Socket，进行流量分配
  - 封装和拆封网络流量

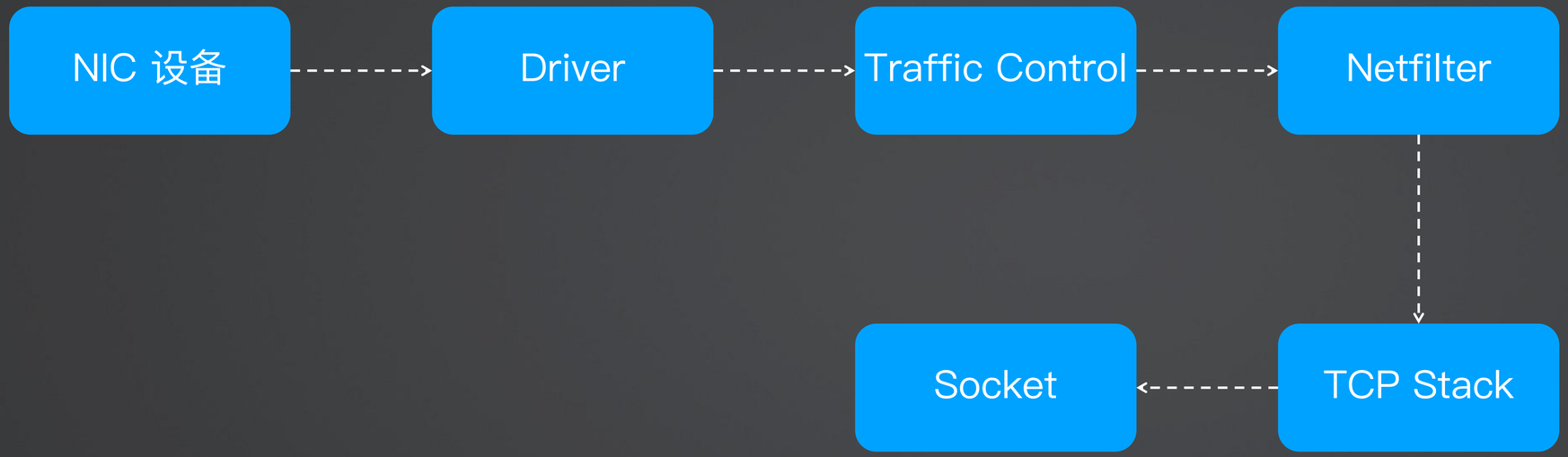
```
// 定义一个 SOCKET_MAP
struct bpf_map_def SEC("maps") my_sock_map = {
    .type = BPF_MAP_TYPE_SOCKMAP,        // 套接字映射类型
    .key_size = sizeof(int),             // 套接字的索引
    .value_size = sizeof(int),           // 套接字的文件描述符（FD）
    .max_entries = 64,                   // 最大支持 64 个套接字
};
```

3. XDP

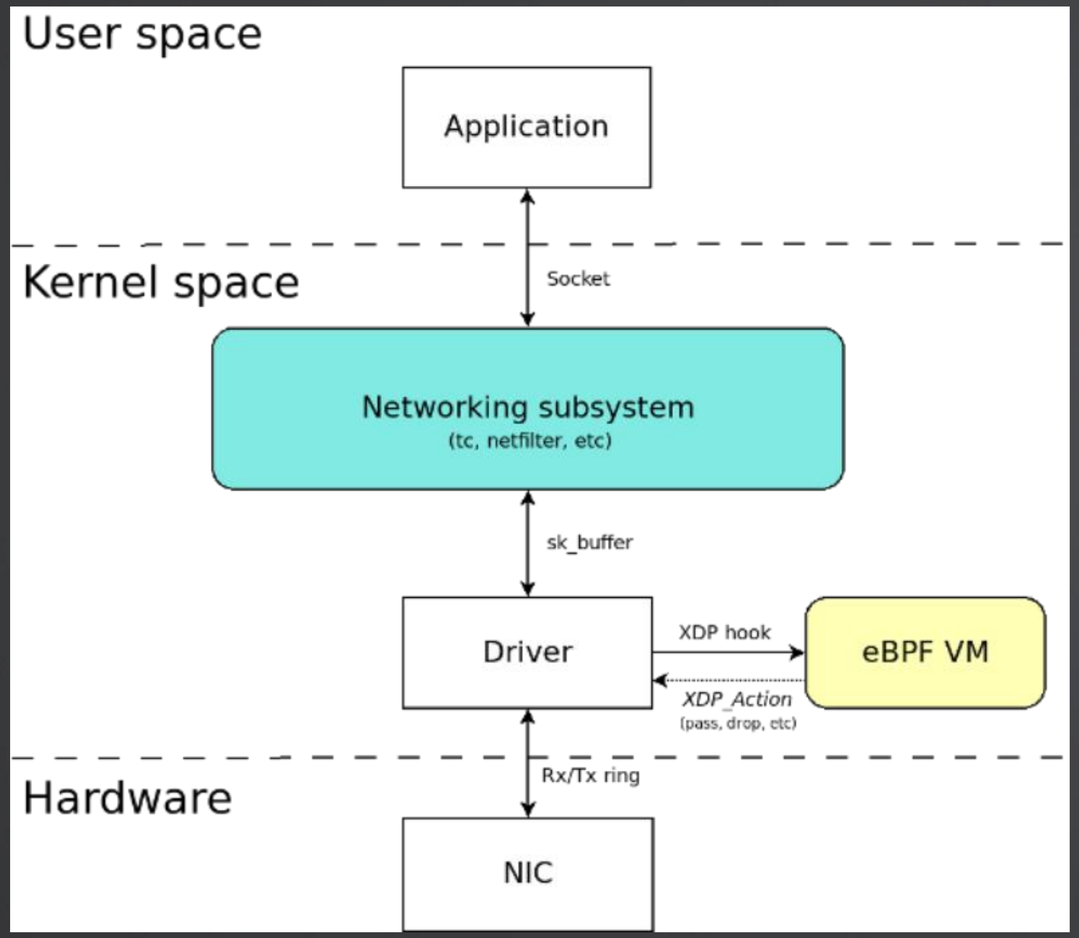
什么是 XDP

- XDP（eXpress Data Path）是基于 eBPF 的一个框架，专门用于网络数据包处理
- XDP 允许你在接收到网络数据包时立即触发一个 eBPF 程序，可执行以下动作
  - 丢弃（DROP）
  - 转发（TX）
  - 重定向（REDIRECT）
  - 通过协议栈处理（PASS）
  - 暂时持留（ABORTED）

引入 XDP 之前

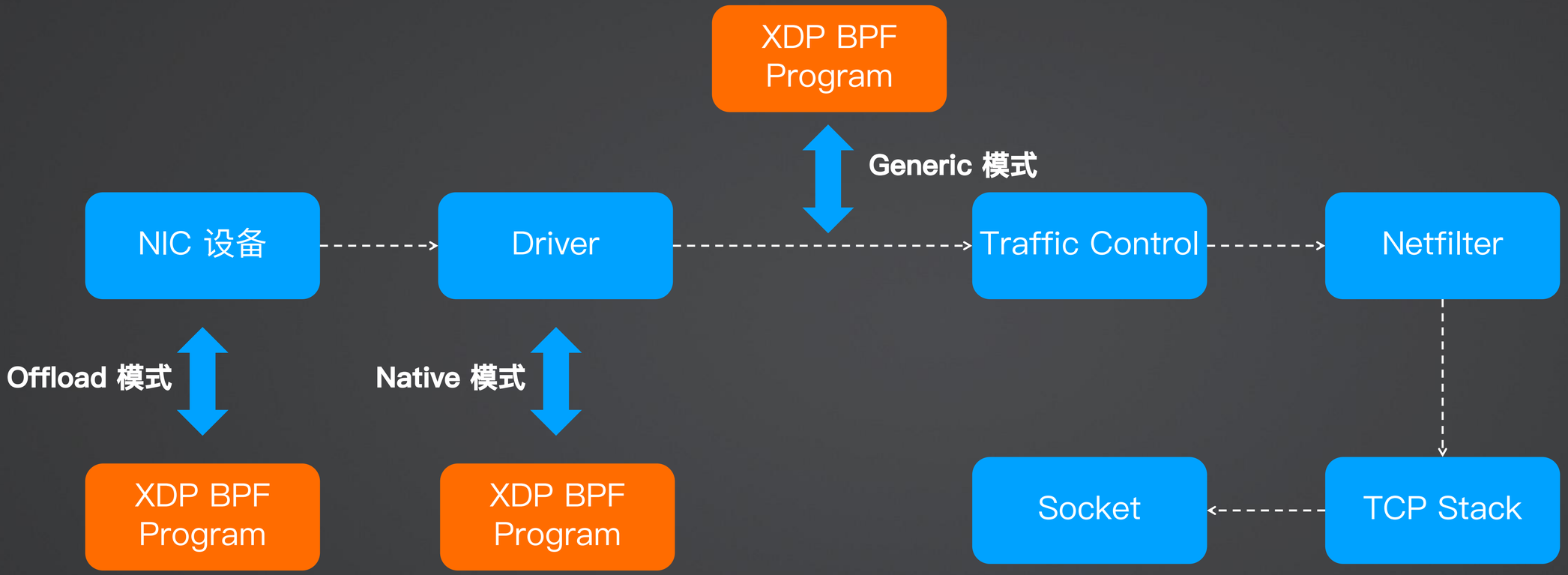


引入 XDP 之后



- 在内核网络堆栈中添加一个检查点（Checkpoint）
- 将数据包传给 eBPF 程序
- 程序决定如何处理数据包
  - 丢弃
  - 通过

XDP 挂载模式

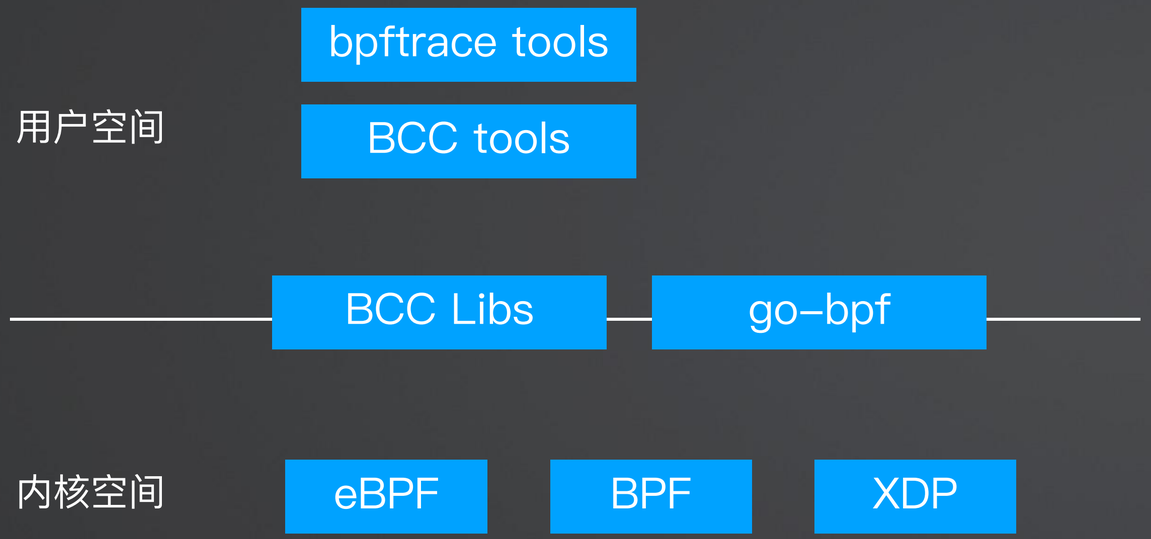

XDP BPF Program
Generic 模式
NIC 设备
Driver
Traffic Control
Netfilter
Offload 模式
Native 模式
XDP BPF Program
XDP BPF Program
Socket
TCP Stack

XDP 的使用场景

- DDoS 攻击和防火墙：使用 XDP_DROP 丢弃网络包
- 负载均衡：使用 XDP_TX 和 XDP_REDIRECT 对包进行负载均衡和转发
- 监控和流量采样：通过自定义元数据来识别包
- Cilium 项目的核心是 XDP 技术
- Cloudflare 基于 XDP 实现了高效的 DDoS 防护

4. eBPF 相关的项目

eBPF 相关项目



Bcc lib – 用于高级应用程序与 bpf 通信的库

go-bpf – 用于 bpf 的 golang lib Bcc 工具

Bcc-tools – 类似于 tetracer 的用户空间工具，用于跟踪状态

bpftrace – 基于高级用户空间 bpf 的跟踪工具。

THANKS