Checkpoint/Restore In eBPF (CRIB)

Juntong Deng

juntong.deng@outlook.com

LINUX
PLUMBERS
CONFERENCE Vienna, Austria / Sept. 18-20, 2024

Overview

- CRIB (Checkpoint/Restore In eBPF) is an innovative checkpoint/restore approach.
- Dump/restore process information in the kernel via eBPF.
- Better performance, more flexibility, more extensibility (easier to support dumping/restoring more information), and more elegant implementation.

Agenda

- What is checkpoint/restore (C/R)?
- Previous C/R solutions
- Current issues
- CRIB introduction
- Why CRIB is better?
- Future plan
- Questions & Discussion

What is checkpoint/restore (C/R) ?

- Checkpoint: Freezing an application (or container) and exporting all process status information to image files.
- Restore: Restoring and running applications from previously exported image files.
- From the perspective of application, it does not know what just happened.

Checkpoint/restore usage scenarios

- Live migration
- Speed up application startup
- Seamless kernel upgrade
- Application snapshot
- ...

Checkpoint/restore difficulties

- Complete process status information is not only in user space, but also in the kernel, e.g. opened files, and sockets.
- Support checkpoint/restore for all features that the kernel provides to applications
- Need to keep up with the Linux kernel developments
- Never-ending story

As the Linux kernel has more and more features, implementing checkpoint/restore will become more and more complex and difficult.

We need a checkpoint/restore approach that is flexible, extensible, and has controllable maintenance costs.

Previous C/R solution: kernel-based

2008-2010: Oren Laadan and other developers tried to implement checkpoint/restore by adding checkpoint() and restart() system calls.

```
int checkpoint(pid_t pid, int fd, unsigned long flags, int logfd);
int restart(pid_t pid, int fd, unsigned long flags, int logfd);
```

But this solution was not merged, because:

- Too complex (~100 patches, ~23,000 lines of code)
- Too intrusive implementation (code spread across various subsystems)
- High maintenance costs
- Fragile serialization and deserialization
- ...

Previous C/R solution: mostly userspace-based

2011-now: Pavel Emelyanov sent patch series and proposed to implement most of checkpoint/restore in userspace with only minor help from the kernel.

With the efforts of Pavel Emelyanov and other developers, the CRIU (Checkpoint/Restore In Userspace) project was born.

CRIU is currently the most mature, feature-rich, and widely used checkpoint/restore implementation, and is integrated into other software such as OpenVZ, Docker, LXC, etc.

Technologies CRIU is based on

- procfs
- extended system call interface
- netlink
- ptrace()
- parasite code injection
- ...

Current issues: procfs

The checkpoint procedure of CRIU relies heavily on procfs to get process information, including file descriptors, memory maps, sockets, statistics, etc.

But procfs is not really suitable for checkpoint, because:

- Lots of system calls, lots of context switches (open, read, close)
- Fixed return information
- Lots of extra information, slow to read
- Variety of formats
- Non-extensible formats
- ...

Current issues: netlink

CRIU relies on sock_diag (netlink) to obtain some socket information. In 2015, Andrey Vagin tried to add similar task_diag to replace procfs to obtain process information.

But task_diag was not merged, because:

- Netlink is a kind of socket and cannot elegantly obtain the pidns and userns of the process.
- Since netlink cannot handle namespaces well, obtaining process information via netlink can lead to credential security issues (e.g. carefully constructed code can obtain process information under another pidns that should not be visible).

Current issues: extended system call interface

Most system call interfaces are not designed for checkpoint/restore, some process information is difficult to dump/restore via normal interfaces, so we need to extend it.

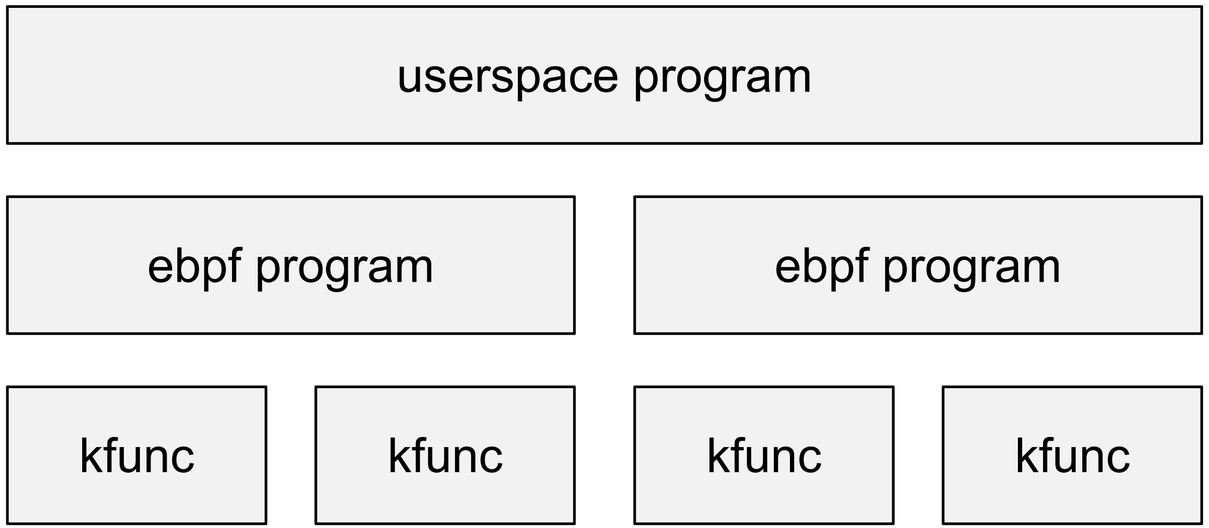
- Change system call behaviour (e.g. in TCP repair mode, recvmsg() will read packets back from the send queue, and sendmsg() will send packets to the receive queue.)
- Add more flags, options, and commands (e.g. TCP_QUEUE_SEQ option for getsockopt() is used to obtain the TCP sequence number).

That is feasible, but not good practice.

Current issues: system call interface

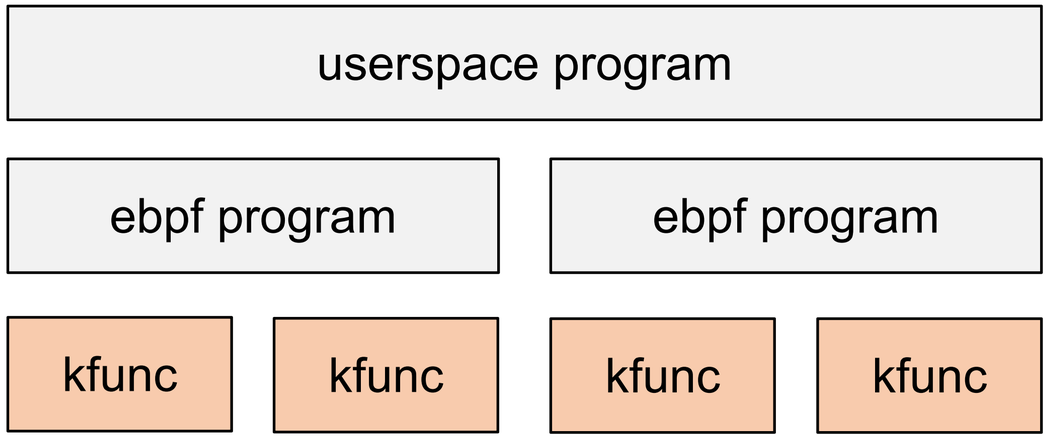
- Changing the standard behaviour of a system call is not a good design, adding a new interface might be better.
- As more and more features are added to the kernel, implementing checkpoint/restore via only normal interfaces will become more and more difficult (or even impossible).
- We need to continue to extend and need more and more xxx repair modes, ioctl commands, getxxxopt/setxxxopt options.
- Extensions are cumbersome, and even if we just want to get a new structure member, we need to patch the kernel to extend the interface.

CRIB introduction



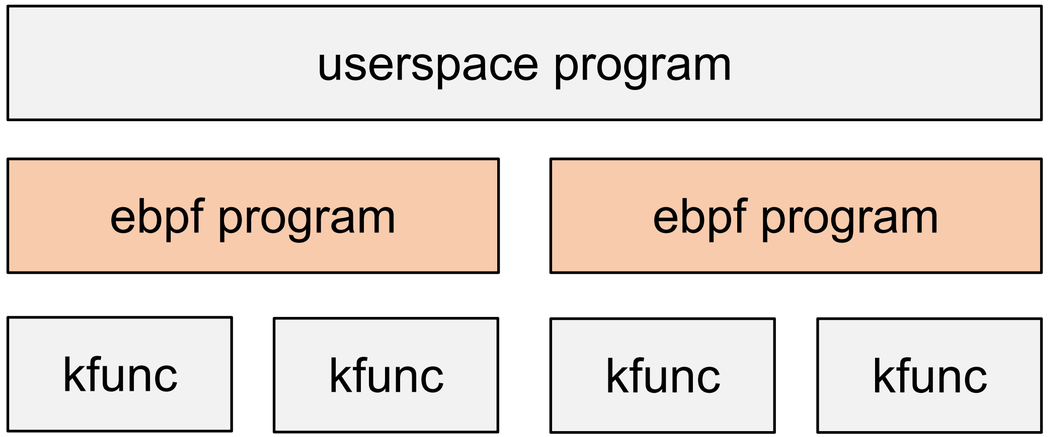
CRIB (Checkpoint/Restore In eBPF) consists of three parts: CRIB userspace program, CRIB ebpf programs, and CRIB kfuncs.

CRIB introduction: CRIB kfuncs



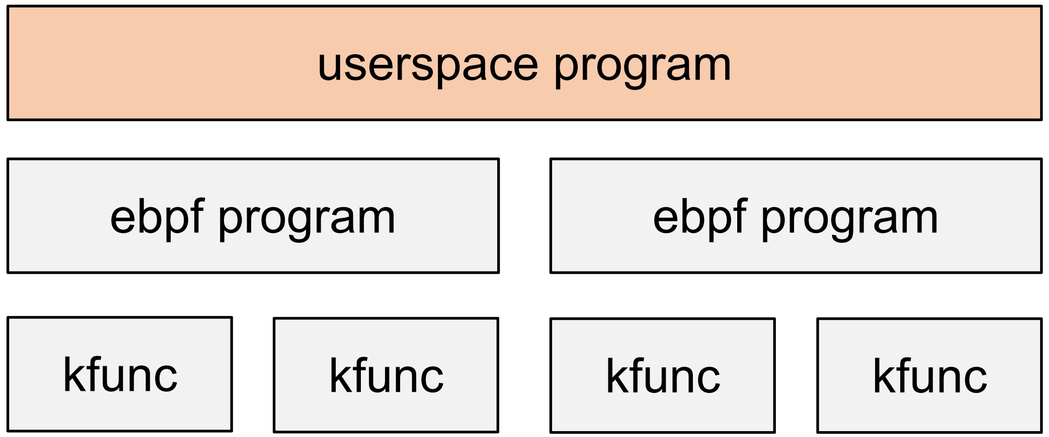
- Provide low-level APIs. Each low-level API is only responsible for a small task.
- Only used for cases where information cannot be obtained by reading structure via BPF_CORE_READ(), such as obtaining struct file object based on file descriptors.
- Each CRIB kfunc needs to be kept simple enough without too complex code and not require too much modification even in the future.

CRIB introduction: CRIB ebpf program



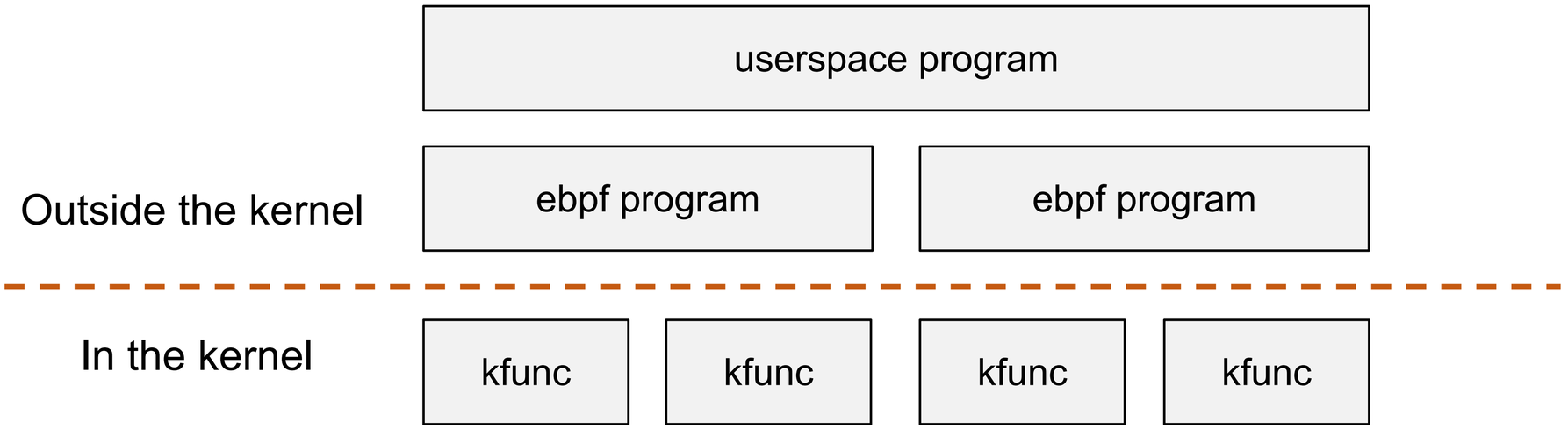
- Provide high-level APIs. Each high-level API is responsible for more complex tasks, such as obtaining all socket information of the process.
- Run through BPF_PROG_RUN and not attend to any hooks.
- Each CRIB ebpf program obtains process information in the kernel via CRIB kfuncs and BPF_CORE_READ(), and returns the data to CRIB userspace program via ringbuf.

CRIB introduction: CRIB userspace program



- Decide what needs to be dumped and what needs to be restored.
- Responsible for loading CRIB ebpf programs and calling CRIB ebpf high-level APIs.
- Responsible for saving the exported information to image files at checkpoint and parsing image files at restore.
- We use CRIU as the CRIB userspace program, CRIB as the new engine for CRIU.

Why CRIB is better: Complexity outside the kernel



- In the CRIB design, complex ebpf programs and userspace program are outside the kernel, and we do not need to add much complexity to the kernel.
- We only need to add some simple CRIB kfuncs to the kernel to provide some necessary help.
- CRIB ebpf programs will be maintained with the CRIB userspace program (CRIU).

Why CRIB is better: More flexible and extensible

```
struct tcp_sock *tp = bpf_tcp_sock_from_sock(sk);
e_tcp->hdr.type = EVENT_TYPE_TCP;
e_tcp->write_seq = BPF_CORE_READ(tp, write_seq);
e_tcp->rcv_nxt = BPF_CORE_READ(tp, rcv_nxt);
e_tcp->snd_wl1 = BPF_CORE_READ(tp, snd_wl1);
e_tcp->snd_wnd = BPF_CORE_READ(tp, snd_wnd);
e_tcp->rcv_wnd = BPF_CORE_READ(tp, rcv_wnd);
e_tcp->rcv_wup = BPF_CORE_READ(tp, rcv_wup);
e_tcp->max_window = BPF_CORE_READ(tp, max_window);
```

- Since ebpf programs run in kernel space, we can naturally read most of the information in kernel structures via BPF_CORE_READ.
- We do not need to add trivial CRIB kuncs to obtain structure members and to obtain a new structure member we do not need to modify the kernel.
- Since ebpf programs are maintained with userspace program, ebpf programs do not need to provide stable APIs and can be changed with the needs of userspace program.

Why CRIB is better: Higher performance

- Since CRIB is very flexible (ebpf programs are changeable), we can dump/restore only the information we need, without any unnecessary information.

- ebpf programs can return binary data directly (not text) via the BPF ringbuf, without any additional conversion or parsing.

- With BPF ringbuf, we can avoid lots of system calls, lots of context switches, and lots of memory copying (between kernel space and user space).

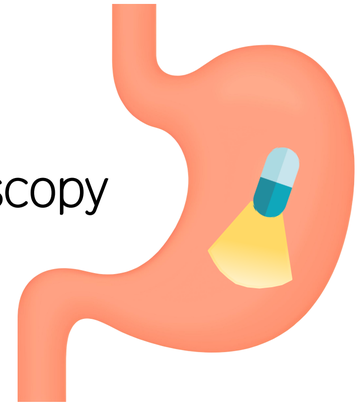
Why CRIB is better: Better support for namespaces

```
struct task_struct *find_task_by_vpid(pid_t vnr)
{
        return find_task_by_pid_ns(vnr, task_active_pid_ns(current));
}

__bpf_kfunc struct task_struct *bpf_task_from_vpid(pid_t vpid)
{
        struct task_struct *task;

        rcu_read_lock();
        task = find_task_by_vpid(vpid);
        if (task)
                task = bpf_task_acquire(task);
        rcu_read_unlock();

        return task;
}
```

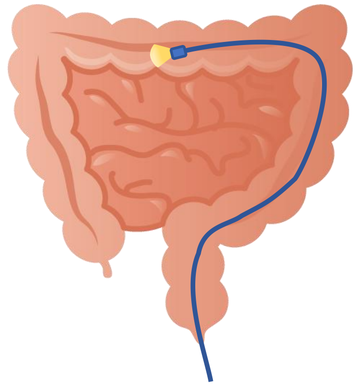
- With reasonable help from kfuncs, it is not difficult for ebpf programs to obtain the current namespaces (e.g. pidns, userns) of the current process (or other processes).
- The security issues in netlink mentioned earlier do not exist in CRIB. There is no situation where a process has dropped privileges, but CRIB cannot know that.

Why CRIB is better: More elegant way

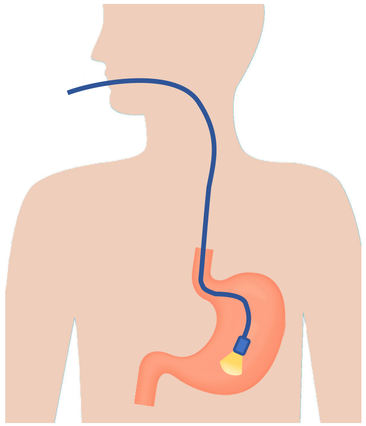
Capsule Endoscopy



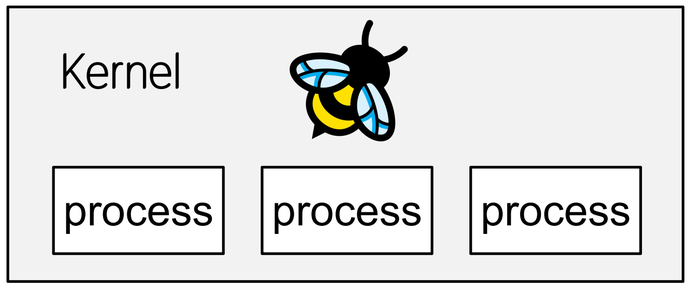
Colonoscopy



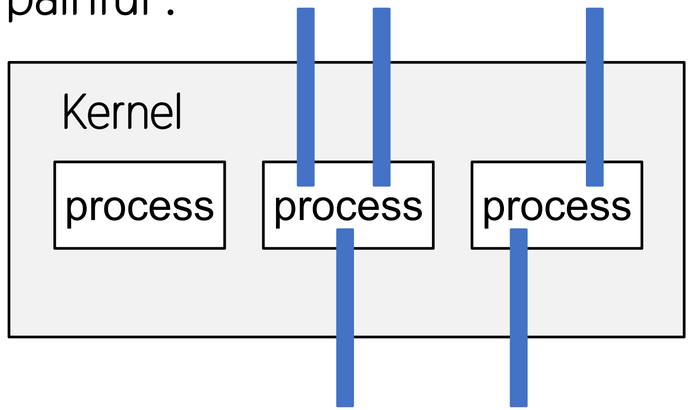
Gastroscopy



We no longer need to look for (open) more inelegant "holes" (interfaces) on the kernel to get information and make the kernel less "painful".



CRIB



system call interface

Shortcoming?

Yes, loading the ebpf programs takes time.

However, in most scenarios, CRIU runs as a service and is integrated into other software (via RPC or C API), rather than as a standalone tool.

This means that CRIU will handle multiple checkpoints/restores, but CRIB ebpf programs only need to be loaded once, and can be then used like normal APIs.

Overall, it is worth it.

Future plan

- Progressively replace inefficient procfs-based checkpoints in CRIU with CRIB (e.g., smap, mountinfo).
- Progressively support checkpoint/restore features that are currently difficult to support in CRIU through CRIB (e.g., UDP CORK-ed sockets, POSIX message queues).
- Add necessary CRIB kfuncs to upstream kernel for reasonable help.
- Add multi-engine support to CRIU. Add checkpoint/restore operation sets, and choose whether each operation uses CRIB method or traditional method when CRIU starts.
- Further future, progressively replace checkpoint/restore based on complex system call interfaces (e.g. TCP repair mode).

Future plan

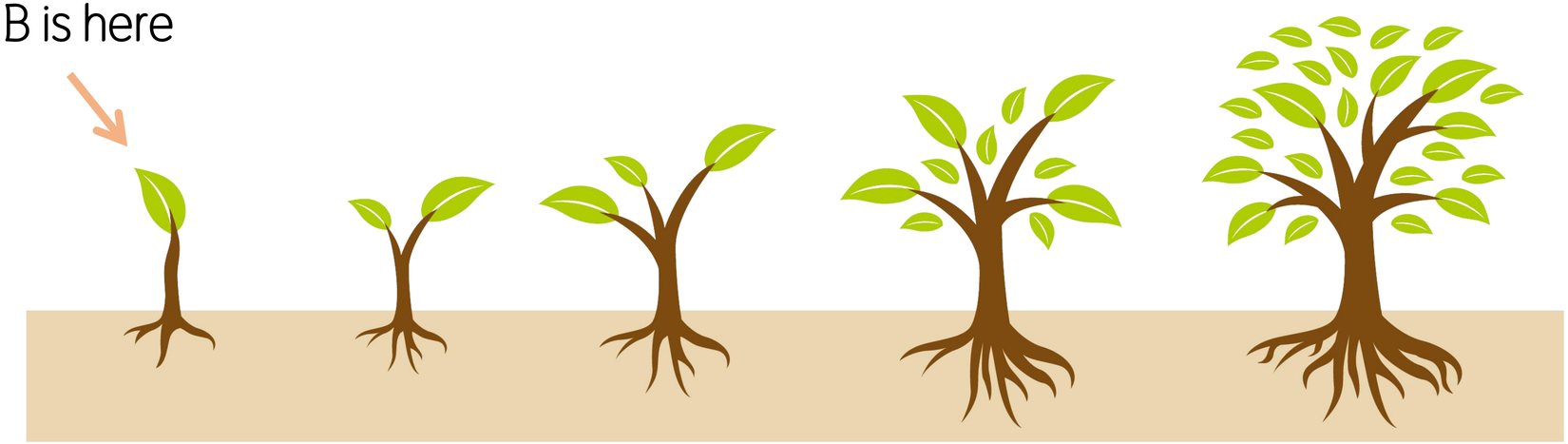
Currently CRIB is still in the very early stage

Some proof-of-concept programs in the initial patch series

CRIB needs time to grow up

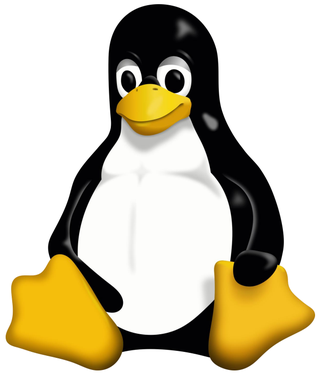
Maybe we can review the progress at LPC 2025...

CRIB is here



Thanks

Thanks to Pavel Tikhomirov, Alexei Starovoitov, Kumar Kartikeya Dwivedi, Linux Kernel Community, CRIU Community.



Thank you for listening

Questions from LWN.net

- Why not an out-of-tree module?

  This question is similar to ebpf vs kernel modules.

  - Portability: Based on BPF CO-RE, we can implement ebpf programs that support multiple kernel versions without needing to recompile or maintain different code for different kernel versions.
  - Security: Based on BPF verifier, CRIB ebpf programs will not damage kernel and cause kernel crash.
  - With the help from kfuncs, the functionality of CRIB ebpf programs does not differ from kernel modules.
  - After JIT compilation, ebpf programs should be as efficient as kernel code or kernel modules.

- Example applications?

  - This is answered in the "Checkpoint/restore usage scenarios" section.

Discussion: Portable restore

- On restore, we need a portable way to write data to different versions of the kernel.
- BPF_CORE_WRITE cannot be introduced because it will compromise BPF safety (it is unsafe for ebpf programs to write directly to kernel structures). We need to use kfuncs, but how to make kfuncs portable?
- Current method is to use variable-size structures based on __sz annotations (add version number field if necessary) for backward compatibility of kfuncs, and bpf_core_field_exists and LINUX_KERNEL_VERSION for forward compatibility of kfuncs.
- Support for different versions of structures, if old versions of structures lack fields then use default values.
- There are some maintenance costs, but we can benefit from performance improvements (compared to traditional system call interfaces).

Welcome to discuss any related ideas.

Questions & Discussion