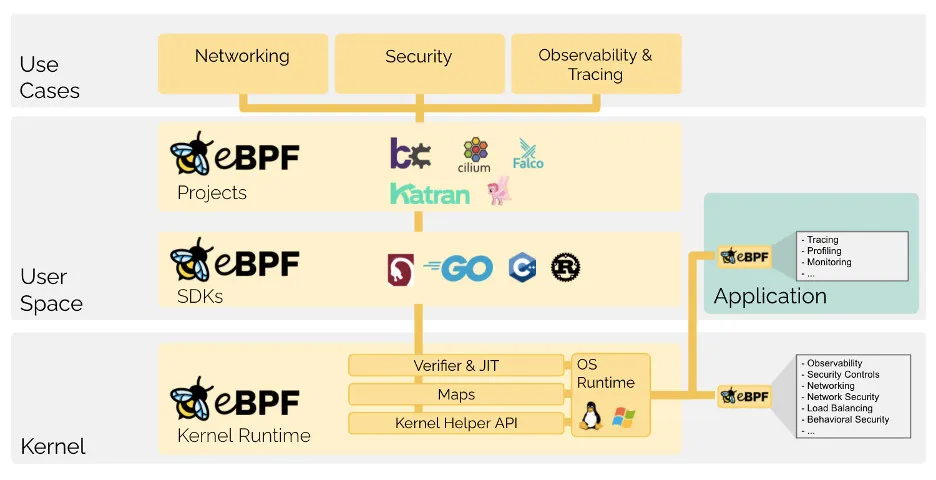
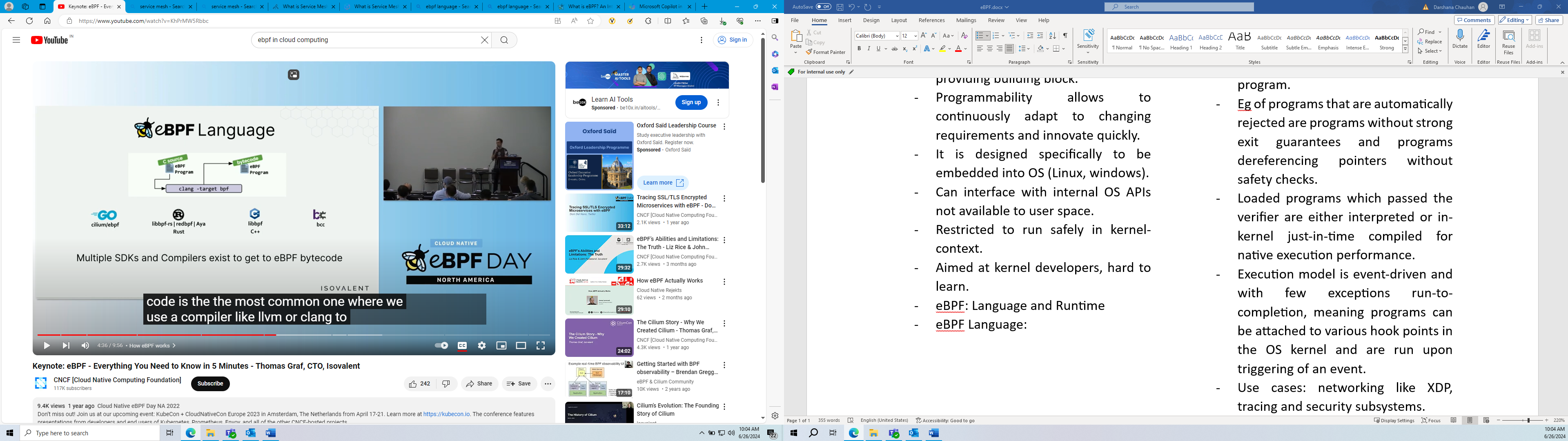
**eBPF: (extended Berkeley Packet Filter)**

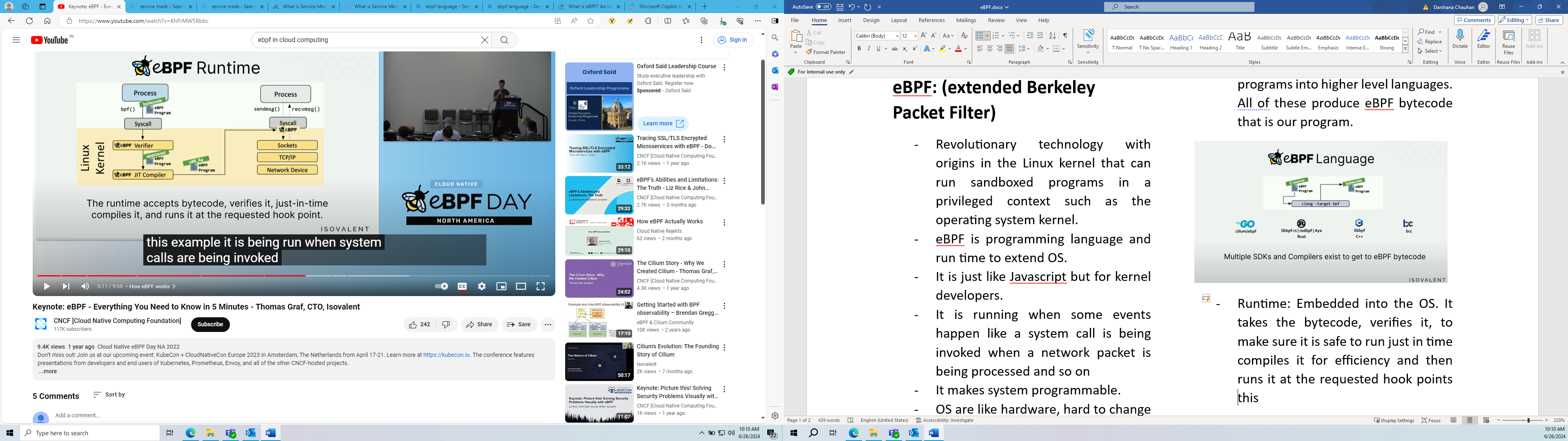
* Revolutionary technology with origins in the Linux kernel that can run sandboxed programs in a privileged context such as the operating system kernel.
* eBPF is programming language and run time to extend OS.
* It is just like Javascript but for kernel developers.
* It is running when some events happen like a system call is being invoked when a network packet is being processed and so on
* It makes system programmable.
* OS are like hardware, hard to change and with long innovation cycles.
* Long innovation cycle results in need to predict use cases or stick to providing building block.
* Programmability allows to continuously adapt to changing requirements and innovate quickly.
* It is designed specifically to be embedded into OS (Linux, windows).
* Can interface with internal OS APIs not available to user space.
* Restricted to run safely in kernel-context.
* Aimed at kernel developers, hard to learn.



* eBPF: Language and Runtime
* **eBPF Language**: pseudo c code is the most common one where we use a compiler like LLVM or clang to compile that into bytecode but there are several different language frameworks to express eBPF programs into higher level languages. All of these produce eBPF bytecode that is our program.



* **eBPF Runtime**: Embedded into the OS. It takes the bytecode, verifies it, to make sure it is safe to run just in time compiles it for efficiency and then runs it at the requested hook points this example, it is being run when system calls are being invoked.



* Properties:
  + Secure:
  + Efficient:
  + Portable:
* Use cases:
  + Cloud native eBPF Landscape
    - Application observability
    - Networking and service mesh
    - Security
  + High performance
  + Load-balancing
  + Low overhead
  + Performance
* eBPF is a technology that can run programs in a privileged context such as the operating system kernel.
* eBPF programs used to add additional capabilities to the OS at the runtime.
* Successor to BPF filtering mechanism in Linux and is also used in other parts of Linux kernel as well.
* For safely and efficiently extend the capabilities of the kernel at runtime without requiring changes to kernel source code or loading kernel modules.
* Safety – by an in-kernel verifier which performs static code analysis and rejects programs which crash, hang, or otherwise interfere with the kernel negatively.
* Validation model differs – sandboxed environments, execution environment is restricted, and the runtime has no insight about the program.
* Eg of programs that are automatically rejected are programs without strong exit guarantees and programs dereferencing pointers without safety checks.
* Loaded programs which passed the verifier are either interpreted or in-kernel just-in-time compiled for native execution performance.
* Execution model is event-driven and with few exceptions run-to-completion, meaning programs can be attached to various hook points in the OS kernel and are run upon triggering of an event.
* Uses:
  + networking like XDP, tracing and security subsystems.
  + Observability:
  + Tracing and Profiling:
  + Security:
* BPF operates with hooks into the kernel. Whenever one of these hooks triggers, the corresponding eBPF program will run.
* It allows programs to run into a protected environment, within kernel space, that verifies the safety of code before allowing it to execution.

[What is eBPF? An Introduction and Deep Dive into the eBPF Technology](https://ebpf.io/what-is-ebpf/)

<https://www.datadoghq.com/knowledge-center/ebpf/>

what is kernel?

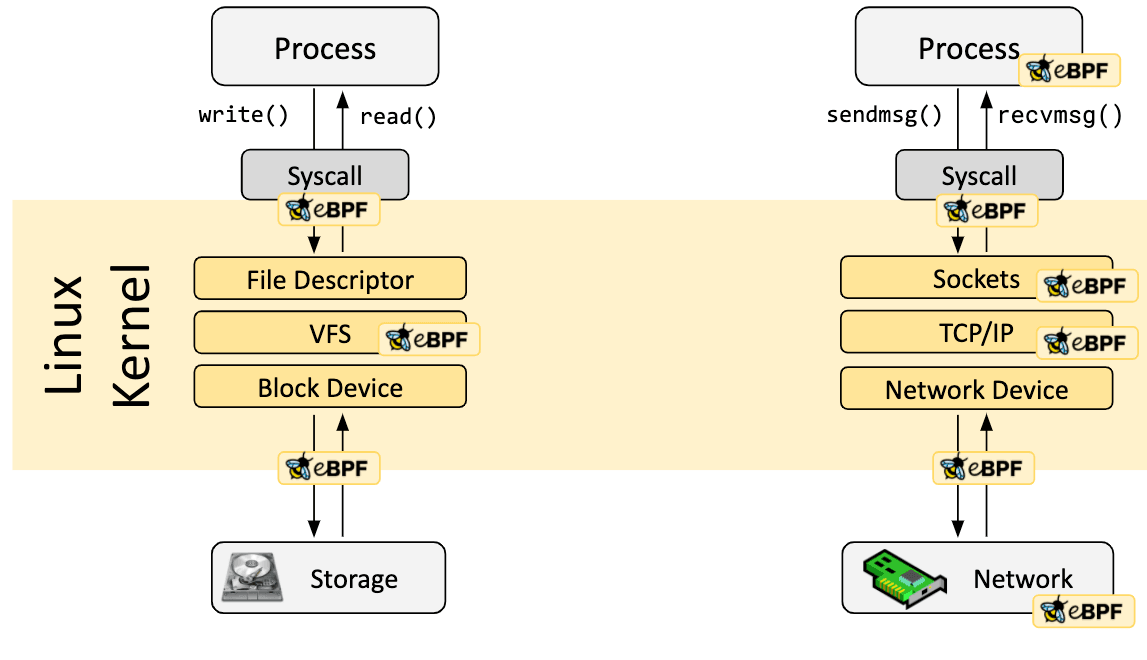
It is the portion of the operating system code that is always resident in memory and facilitates interactions between hardware and software components.

What is eBPF.io?

* a place for everybody to learn and collaborate on the topic of eBPF.

Introduction:

* eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point.
* hooks include system calls, function entry/exit, kernel tracepoints, network events, and several others.
* If a predefined hook does not exist for a particular need, it is possible to create a kernel probe (kprobe) or user probe (uprobe) to attach eBPF programs almost anywhere in kernel or user applications.

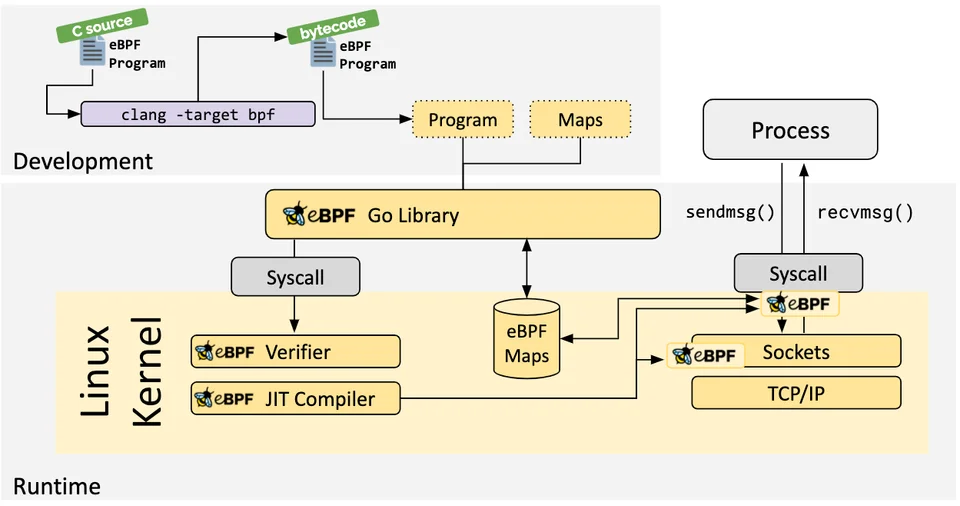


How are the programs written?

* Sometimes, eBPF is used via projects like Cilium bcc, bpftrace which provide an abstraction on the top of eBPF and can specify intent-based definitions which are then implemented with eBPF.

Loader and Verification Architecture:

* Verification:
  + Verify that the program is safe to run.
  + Validates that the program meets the conditions.
  + Process holding eBPF program holds the required capabilities (privileges). Unprivileged eBPF is enabled, only privileged processes can load eBPF programs.
  + The program doesn’t crash or otherwise harm the system.
  + The program runs to completion.



* JIT Compilation:
  + Translates the generic bytecode of the program into the machine specific instruction set to optimize execution speed of the program.
  + makes programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module.
* Maps:
  + Vital aspect is the ability to share collected information and to store state. For that it has the concept of eBPF maps
  + It can be accessed from eBPF programs as well as from application in user space via a system call.
* Helper Calls:
  + eBPF programs cannot call into arbitrary kernel functions.
  + Allowing this would bind eBPF programs to particular kernel versions and would complicate compatibility of programs.
  + eBPF programs can make function calls into helper functions, a well-known and stable API offered by the kernel.
* Tail and function calls:
  + Composable by this.
  + FC Allows defining and calling functions within an eBPF program,
  + TC can call and execute another eBPF program and replace the execution context.
* eBPF safety:
  + several layers:
    - required privileges.
    - verifier
    - Hardening
    - Abstracted runtime context

eBPF’s impact on the Linux Kernel

* Purpose of Linux Kernel: to abstract the hardware or virtual hardware and provide a consistent API (system calls) allowing for application to run and share the resources.

Why do I need eBPF if I have a space in hardware?

* eBPF is a kernel technology that allows programs to run without having to change the kernel source code or add additional modules. It's like a lightweight, sandbox virtual machine (VM) inside the Linux kernel.
* The user space is unprivileged; therefore, it can’t access the hardware directly. When an application requires something from the hardware, it will need to request the kernel, which is privileged, to do it on its behalf using the system call (syscall) interface. eBPF allows you to run sandboxed programs in the Linux Kernel, giving engineers powerful and customizable insights into system behavior. It's not just about packets anymore; eBPF can help you explore various system calls, trace points, and more.

Why do I need to implement an application at the kernel level?

* Implementing an application at the kernel level can provide several benefits. The kernel provides a layer of abstraction between the hardware and the application programs, enabling software developers to write programs without needing to understand the specifics of the underlying hardware.
* Kernel threads are created by the kernel and are visible to it. A user process with the help of a provided library asks the kernel to create an executable thread for that process and the kernel in turn creates the thread on behalf of the process. This can be extremely beneficial when used under the right circumstances.
* In summary, both eBPF and kernel-level applications provide powerful tools for interacting with and manipulating the system at a low level, offering greater control and flexibility than would be possible otherwise.