UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

croissant.png

Msc. Thesis

Jacob Herbst (mwr148)

In-kernel Linux Proof Checker

For PCC'ifying the eBPF syscall

repository: `github.com`

Advisor: Ken Friis Larsen

2023-05-31

Contents

Abstract

This project investigates the possibility of using Proof Carrying Code (PCC) for the EBPF Linux subsystem. More specifically I focus on the validation of proofs in the context of PCC. I do so by considering the dependently typed language Logical Framework with Side Conditions (LFSC), how it can be used for proof-checking and an implementation in Rust. In doing so this project investigates both how suitable Rust is for implementing a PCC architecture that can run in the Linux kernel, aswell as considering if LFSC is at all useful when under the constraint of the Linux kernel.

Chapter 1

Introduction

Extended Berkeley Package Filters (eBPF) is a subsystem in the Linux kernel, which allows users of the system to dynamically load eBPF bytecode into the kernel. The program can then be executed when certain events happen. This technology enables many interesting features, such as high-speed package filtering (which was the initial intend with the system) and Express Data Path (XDP) by circumventing the network stack of the operating system. Furthermore it can be used as system monitoring by access to kernel probes etc.

eBPF allows untrusted users to run arbitrary code in the kernel, which in itself is no problem if the program is non-malicious, but can be detremental if not. Programs that run in the Linux kernel must therefore be both safe and secure. an usafe program in the linux kernel might break the system alltogether, whilst a malicious users could get access to secrets. To prevent this the eBPF subsystem will perform an abstract interpretation of a program and then reject unsafe programs, before they are loaded into the kernel. This process is called the verifier. The verifier has been subject to multiple bugs in the past, which can lead to priveledge escalation[**TODO: ebpf priveledge**]. Furthermore the verifier will outright reject programs with loops, since the domain of the abstract interpretation have no way to tell if the program will terminate (and this ofcourse is not possible in general).

An alternative to verifying the safety of programs in general is to do a formal proof of safety. An automatic way to do this is to reduce the problem of program verification to satisfiability of programs. This is done by generating a formula that describe the properties of a program, a so called verification condition, and then checking the satisfiability of the negated formula. The process of generating a verification condition is the fairly cheap, however checking the validity of the formula can be quite a heavy task. Proof Carrying Code is an architecture and security mechanism first introduced by Necula[**necula**] in 1998 which moves the responsibility of proving that a certain program follows a set of security parameters to the user and then have the kernel check that this proof along with some additional checking. The general concept is described more in detail in**??**. All in all these tasks are far cheaper task than producing a proof.

In this project I investigate the feasibility of a proofchecker that can run inside the Linux kernel written in Rust. The proof checker leverages the Curry-howard correspondence and amounts to typechecking a dependently typed language, Logical Framework with Side Conditions (LFSC).

?? explains the necessary understanding PCC **??** explains how we enables the feature of a proof checker that can run in hte Linux Kernel. **??** gives a formal defintion of the LFSC language **??** describes the concrete syntax and gives an example **??** presents the overall design of the proof checker, whilst the concrete implementation is covered in **??**. Lastly I evaluate the performance of the proof checker and analyze the feasibility comparative to the verifier.

Enjoy.

Chapter 2

TODO [0/1] Proof Carrying Code

Proof Carrying Code (PCC) is a mechanism which guarantees safety for a host system in execution of programs from untrusted sources. A diagram of the acrhitecture can be seen in figure**??**. We split the architecture over two spaces, namely that of the code-producer and the code consumer. The code producer has the intend of having the code producer run some code. To protect the code consumer of any malicious or unsafe programs the code producer will issue a collection of safety rules that a code producer should adhere to, denoted the safety policy. Such safety policy could include no out of bound memory operations and termination of loops. The code producer will then use the safety policy to make a certification of the program. This will happen in a compilation stage of the untrusted binary, and the actual certification will depend on the implementation. In any case the compilation and certification process will produce both the native code to be executed at the consumer and the certificate for the safety of the native code. This process will be the most computationally heavy, otherwise the code consumer could do the certification and compilation itself.

The next stage is the verification where the producer handover the result of stage one to the consumer and the consumer check the validity of the proof. The check amounts to two things:

1. The safety proof must be valid modulo the safety policies.
2. The safety proof must correspond to the native code.

This process should be quick and follow an algorithm trusted by the consumer. Given that the two criteria is met the consumer can then mark the native code as safe and can proceed to execute the program possibly many times.

Chapter 3

TODO [0/4] The Linux Kernel

3.1 TODO Linux

The linux kernel is a monolithic kernel which means not only does it contain the core necessities for a operating system but also a virtual interface to communicate with hardware, such as filesystems, networkstacks and device drivers. All of these run in kernel space and is made available to the user through a common interface called system calls (syscalls). So in general all communication between user space and kernel space will happen through the systemcall. In some sense it is also possible to communicate between the two using the filesystem however this is merely an additional layer of abstraction.

Despite being monolithic, the Linux kernel is modular, meaning it is possible to dynamically load kernel modules, Essentially we can extend kernel to undertake arbitrarily complex services. Kernel modules works very similar to filesystems in general.[1]

In the following sections **??**-**??** we will describe what and why of ebpf and discuss the different ways to realize the PCC architecture for the eBPF subsystem.

3.2 TODO eBPF

As mentioned the Linux kernel allow arbitrary extending the kernel thorugh Loadable Kernel Modules (LKMs), so what even is the point of eBPF? The two main reasons for this is that LKMs requires both root priveledge and are unsafe by nature. The unsafety comes from the fact that a kernel modules like any other kernel code has ring 0 permission meaning very little security is provided and thus a malicious LKM can destroy a kernel completely. This becomes especially apparent when considering that kernel modules might be proprietary. Once again we reach a dilemma of extensionality vs trust. LKM in general has no reason to provide any safety measures because of the priveledge requirements, but this blindness to trust in the user (superuser in this case) can be detremental to a system.[2] eBPF takes a different approach.

BPF (Berkeley Package Filter), as it was originally presented, is a system for effective filtering

[1]this might be slightly simpified
[2]is this to aggressive and dark?

of network packages by allowing dynamically loaded package filters from userspace into the network stack in the kernel space. eBPF extends this functionality to a wide variety of tasks in observability, security and networking. eBPF leverages the priveledges of the kernel to oversee the entire systems and thus allows for more powerful control. eBPF is a virtual machine that JIT-compiles a RISC instruction set (also called eBPF..), running inside the kernel. eBPF programs are event-driven and must be attached to a hook to be executed. The exact process is as follows:

1. A user optains a eBPF program, either using an abstraction tool such as BCC, libbpf etc. or if none of these tool provide the intend can write the concrete program by hand.
2. the program can be loaded using the bpf syscall. The verifier will perform static analysis in the form of abstract interpretation using tristate numbers `tnums` cycle detection etc. Later in this section we go through exactly what the verifier does, but it will check certain criteria such as division by 0, no backwards jumps(i.e. loops) etc.
3. If the program is allowed by the verifier it is JIT-compiled making eBPF programs as fast as native code.
4. Programs can then be attached to a certain hook and every time an event occurs the program is triggered. An instance of this could be a socket, and every time something is written to this socket the program is triggered.
5. Allthough an eBPF program lives in kernel space, it conceptually lives somewhere in between user and kernel space. But it has two way to interact with both kernel and user space.
   - Unlike LKMs eBPF programs cannot call kernel functions directly as eBPF is designed to be kernel version agnostic (in reality this is not really the case, because of the verifier), but the eBPF subsystem also provides a stable API of helper functions to provide functionality not immediately accessible in the limited instruction set.
   - The subsystem also provides a collection of key-values stores called maps, in a variety of different datastructures such as ring buffers, arrays, etc. These datastructures also live in kernel space and are constructed through the bpf syscall, eBPF programs can then read and write to a map. Likewise users can read and write to the maps using the `bpf` syscall.

Figure **??** shows the process of loading the ebpf program, whilst Figure **??** shows the interaction with maps.

3.2.1 The eBPF verifier

This section serves as a mapping of the eBPF verifier. The purpose of this is to serve as a basis for further discussion on why using a Proof Carrying Code approach to eBPF. The mapping is quite verbose and may not be of interest if one has a general understanding of how the verifier works.

1. The bpf syscall All interaction between user and kernel regarding eBPF related matter uses the bpf syscall[3] and has the following signature:

[3]bpf() has syscall number 321

```
asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Argument `cmd` is an integer that defines the intended interaction, for the purpose of this report we only care about the cmd `BPF_PROG_LOAD`. To be able to load a eBPF one of the following criteria must be met: either a program/user must be root or be bpf capable, or the kernel.unprivileged$_{\text{bpfdisabled}}$ kernel parameter must be set to 0, meaning regular users are capable of loading programs. This features has been disabled on many modern Linux distributions for security reasons, such as redhat, ubuntu and suse.[**<empty citation>**]

the `attr` argument is a union of structures that must be correspond to the argument type. For program loading this stuct notably contains the type of program to load, which could be socket programs, kernel probes, Express Data Path etc. The syscall will call the appropriate cmd after some sanity checks, such as wellformedness of the `bpf_attr` union.

2. `bfp_prog_load` The `bpf_prog_load` will do a lot of checks related to capabilities and kernel configurations. These configurations includes memory alignment of the system etc. The specifics is irrelevant but it suggests that when designing a new BPF structure such checks should be considered.[4] For instance, we might consider a static design where only users with network capabilities might load network related eBPF or we could consider a dynamic structure where capabilities are also included in the security policy.

   The program is then prepared for the verifier.[5]

3. Static analysis `bpf_check` The `bpf_check` is what we usually denotes as the verifier. It will perform the static analysis. Firstly the checking environment is setup. This is a large struct and thus is allocated and deallocated attached at each call to `bpf_check` and is thus also too large to see here but can be seen in appendix **??**.

   (a) Firstly subprograms and kernel functions[6] are added to the instructions of the ebpf program.

   (b) Afterwards the function `check_subprogs` is called, where some very basic testing is done, such as subprograms not being allowed to jump outside of its own address space. Control flow is here limited to subprogram and loops is in general not allowed. The last instruction must be either an exit or a jump to another subprogram.

   (c) Next `bpf_check` will check the control flow graph to detect loops in the code.

   (d) all the subprograms are then check according to their BPF TypeFormat (BTF), and the code is checked in a similar manner to the main program according to the abstract interpretation.

   The following is a simplification of[**<empty citation>**]. A program must follow these requirements:

   (a) Registers may not be read unless they have previously been written. This is to ensure no kernel memory can be leaked.

[4]An interesting sidenode is that eBPF programs must be between 1 and 1M instructions depending on user capabilities.

[5]should I specify what is going on here?

[6]why does ebpf.io say no kernel functions?

(b) Registers can either be scalars or pointers. after calls to kernel functions or when a subprogram ends, registers r1-r5 is forgotten and can then not be read before written. r6-r9 is callee saved and thus still available.

(c) Reading and writing may only be done by registers marked by `ptr_to_ctx`, `ptr_to_stack` or `ptr_to_map`. These are bound and alignment checked.

(d) stack space, for same reason as registers, may not be read before it has been written.

(e) external calls are checked at entry to make sure the registers are appropriate wrt. to the external function.

To keep track of this the verifier will do abstract interpretation. the verification process tracks minimum and maximum values in both the signed and unsigned domain. It furthermore use `tnums` which is a pair of a mask and a value. The mask tracks bits that are unknown. Set bits in the value are known to be 1. The program is then traversed and updated modulo the instructions. For instance if register `r2` is a scalar and known to be in the range between `(0,IMAX)` then after abstractly interpreting a conditional jump `r2 > 42` the current state is split in two and the state where the condition is taken now have an updated range of `42 <= r2<= IMAX` etc. Pointers are handled in a similar manner however since pointer arithmetic is inherently dangerous modifying a pointer is very limited in eBPF. Additionally pointers may be interpreted as different types of pointers and are check wrt. the program type they occur in. For instance... TBD...

If all these requirements are met, then an eBPF program is loaded. This mapping ofcause is simplified a lot, but it shows that the current process of checking a valid eBPF program is not a simple task and thus a potential overhaul could be welcomming. The entire verification process (except for a few structs) is placed in `kernel/bpf/verifier.c` which at the time of writing is roughly 19000 lines of code, and these have in the past shown to errorprone.

3.2.2 eBPF and PCC

From the description of PCC in **??** and the description of the eBPF subsystem above, it is straightforward to see responsibility differences. We can compare the two pipelines as follows:

1. **Compilation and Cetification**: For PCC the untrusted program is both compiled and a certificate for safety policy compliance is generated. eBPF does not really "do" anything at this stage as source code is passed directly to the kernel using the syscall.

2. **Verification of certificate**: In PCC the consumer will check the validity of the certification wrt. the safety policy and the source program (possibly in native format), while eBPF will have to do a similar check but directly on the eBPF program. As mentioned the current eBPF verifier uses a abstract interpretation model with a tristate number domain, which is roughly equivalent in complexity.[7]

3. In both structures, once a certificate is checked the program is free to use possibly many times.

So why would we want to swap out the current structure vs eBPF?

[7]is this even true? LOOK at verifier source code.

3.3 TODO How to extend the Linux kernel

3.4 TODO Rust in the kernel

Chapter 4

TODO [0/2] Certification of programs

As mentioned in section**??** the code producer must hand over code in the appropriate format, as well as the certificate, so we must consider an appropriate format for the certificate. The process of checking the certificate should also be both fast and simple. I consider a PCC architecture which uses Verification condition generation on the eBPF most likely by using a derivation of Hoare Logic, since this propose an automatic way of generating formulas, which describes programs.[1] The process of proving the validity of such a verification condition however is not a simple task and is, dependending on the logic, undecidable. We will therefore require the code consumer to rely on the process of making this proof and then present the proof to the kernel. Checking the satisfiability of a formula can be done by a Satisfiability Modulo Theories (SMT) solver. In many SMT solvers it is possible to extract a proof that a certain formula is satisfiable. I have in this work considered two output formats/languages, Alethe and Logical Framework with Side Conditions (LFSC), supported by the CVC5 SMT solver. I will briefly describe why I have chosen to use the LFSC language over Alethe. Both formats follow the LISP family of languages and is therefore simple to parse. Alethe is designed to be easily readable by humans and is strutured as a box style proof.[2] This is not a property necessary for a PCC architecture where we want to automate the entire process. By this construction the Alethe format provides a set of basic inference rules of which there are 91[**<empty citation>**] on which proofs are built, for instance rule 20 denotes reflexivity often denoted as `refl` which describes syntactical equality between two terms modulo renaming of bound variables. This entails that an implementation must implement all rules necessary for a security policy and will thne not be easily extended in the future. LFSC on the otherhand is a metaframework that exploits the Curry-Howard isomorphism, explained in **??**. This metaframework allows for the security policy to be established by signatures, which can easily be extended. This is very much a property of interest, as eBPF might evolve to support new datastructures etc. Furthermore this approach can move bugs out of the in kernel certifier and into the specification. This will enable system administrators to quickly deploy fixes for a bug, by not allowing specific faulty signatures. It is hard to consider both time and memory used of the two languages, without actually implementing both of them. But if we consider the amount of code require for the two formats then there exists a rust implementation for an Alethe proof checker, called

[1] should i go more indepth with this?
[2] example?

carcara[**<empty citation>**]. This implementation is ~13500 lines of code, and this specific implementation does not even support all theories present in CVC5, such as bitvectors. LFSC has a proof checker written in C++ and its code is merely 5000 lines, while the signatures may be arbitrarily large. The solution I present is ~3000 lines. CVC5 provides some signatures constituting <1500 lines. Hence LFSC seems like a better choice for an in kernel proof checker.

4.1 TODO Curry-Howard Isormorphism

4.2 TODO Logical Framework with Side Conditions

LFSC is a extention of the Edinbourgh Logical Framework (LF)[**<empty citation>**] and is a predicative typed lambda calculus with dependent types. This allow proof systems to be encoded as signatures, which amounts to a set of typing declarations. LFSC extends LF by including sideconditions. In this context sideconditions is a operational semantic which might be evaluated during typechecking. I will in the following section describe the calculus and its typing along with the operational semantic of sideconditions.

4.2.1 Syntax

The syntax has 5 categories. At its core LFSC is a typed lambda calculus, so it consists of terms/objects, types and kinds. Terms are denoted by M, N and O. Types are denoted by A and B and used for classification of Terms. Kinds are denoted by K and is used to classify types. I use `x` to be a metavariable ranging over the set of variables that might occur in terms, `c` to denote constants in terms (free variables). `a` will range over constants in types. Sideconditions is denoted by S and T. Patterns describes pattern in side condition match cases and is denoted by P. I write keywords in **bold**. I write [M_1/$x_{1,\ldots,M_n}$/x_n]N, for simultanious substitution of `M_1`, `M_2 , . .` with free ocurrences of `x_1`, ~x_2 ~,. . . in N. We assume renaming, to avoid name clashing. I use * to denote a hole.

Figure **??** shows the syntactical categories. TODO: make the figure.

Terms maybe be either a constant `c`, a variable `x`, a type annotation `M : A` stating that term M must have type `A`, an abstraction λ `x [:M] . N`, which binds the term M to `x` in `N`, and lastly a term might be an application of term M to N, notice that application in terms is left-associative, so M N O is (M N) O.

Types may be a type constant `a`, an application of a type to a term, `A M`. a lambda abstraction . . . , or a dependent product type. Pi types may contain sidecondtions in their binders.

Kinds may be either the **type** which classifies types or they may be a pi type.

Sideconditions may be:

1. an unbounded number, which may be either an integer or a rational.
2. a variable `x`
3. a **let** binding, setting `x` to `S` in `T`.

4. an application. Notice that we require that functions are fully applied and thus non-associative,

sideconditions may then also be one of the categories Compounds, Numerical and Sideeffects.

Compound sideconditions may be either a **fail** `A`, which raises an exception with type `A`, **match** expressions, which will match the scrutinee against patterns and evaluate corrsponding `S_i` in a similar manner to ML. **ifequal** compares `S` and `T` for syntactical equality.

Numerical sideconditions may be the binary operation addition, multiplication and division, these follow standard conventions. It may be a negation, a convertion from integer to rational or it may be one of the two branching constructs.

Sideeffects may be **ifmarked** a branch based on the marking of a varible. markvar which marks a variable, **do** S T, which is equivalent to **let** `x = S T` where `x` does not occur in `T`.

Patterns may be either a constructor applied to 0 or more arguments, but the constructor must always be fully applied.

4.2.2 Signatures and Contexts

TODO: add signatures. In LFSC there is two construct we use to keep track of variables and constants. We have signatures, and contexts. Signatures are used to assign kinds and types to constants. This is what the metaframework revolves around. Contexts are used to assign types to variables. we write them as such and use Σ, Σ' to denote the concatenation of the two signatures Σ and Σ' and similarly for contexts.

$$\begin{aligned} \Sigma \ &::= \langle\rangle \mid \Sigma, a : K \mid \Sigma, c : A \\ \Gamma \ &::= \langle\rangle \mid \Gamma, x : A \end{aligned}$$

The typesystem of LFSC is syntax directed meaning there we have only a single typing rule for each syntactical object. We achieve this by bidirectional typing. That means instead of stating that an expression must have a type, we can either construct a type from it (called synthesis) or we can check that an expression has a type.

$$\begin{aligned} \Sigma \checkmark \quad & (\Sigma \text{ is a valid signature}) \\ \vdash_\Sigma \ \Gamma \quad & (\Gamma \text{ is a valid context in } \Sigma) \\ \Gamma \ \vdash_\Sigma K \quad & (K \text{ is a kind in } \Gamma \text{ and } \Sigma) \\ \Gamma \ \vdash_\Sigma M \Leftarrow A \quad & (M \text{ can be checked to have type } A \text{ in } \Gamma \text{ and } \Sigma) \\ \Gamma \ \vdash_\Sigma M \Rightarrow A \quad & (M \text{ can be synthesized to have type } A \text{ in } \Gamma \text{ and } \Sigma) \end{aligned}$$

Valid signatures: empty contexts are valid. TODO: write something more.

Valid contexts:

$$\text{EMPTY-SIG}\ \frac{}{\langle\rangle\checkmark}\ \text{KIND-SIG}\ \frac{\Sigma\checkmark \quad \vdash_\Sigma K \quad a \notin dom(\Sigma)}{\Sigma, a : K \checkmark}\ \text{TYPE-SIG}\ \frac{\Sigma\checkmark \quad \vdash_\Sigma A : K \quad c \notin dom(\Sigma)}{\Sigma, c : K \checkmark}$$

$$\text{EMPTY-CTX}\ \frac{\Sigma\checkmark}{\vdash_\Sigma \langle\rangle}\ \text{KIND-SIG}\ \frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma A : Type \quad x \notin dom(\Sigma)}{\vdash_\Sigma \Gamma, x : A}$$

$$\text{TYPE}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{type} \Rightarrow \mathbf{kind}}\ \text{TYPEC}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{type^c} \Rightarrow \mathbf{kind}}\ \text{LOOKUP-CTX}\ \frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x, A \Rightarrow}\ \text{LOOKUP-KIND-SIG}\ \frac{\vdash \Gamma}{\Gamma}$$

$$\text{ANN}\ \frac{\Gamma \vdash \check{M} A}{\Gamma \vdash M : A \Rightarrow A}\ \text{PI}\ \frac{\Gamma \vdash \check{M} A \quad \Gamma, x : A \vdash B \Rightarrow \alpha \quad \alpha \in \{\mathbf{type}, \mathbf{type^c}, \mathbf{kind}\}}{\Gamma \vdash \Pi x : A.B \Rightarrow \alpha}\ \text{PI-SIDE}\ \frac{\Gamma \vdash S, A \Rightarrow \quad T}{\Gamma \vdash \Pi x : ST \Rightarrow}$$

4.2.3 Typing

For brevity we simply use $\vdash$instead of $\vdash_\Sigma$ when its meaning is obvious.[3]

[3]what is the right word here?

Chapter 5

TODO [0/5] The in-kernel proof checker

In this Section we provide a highlevel overview of the in-kernel proof checker. Followed by an indepth description of implementation for each subpart of the design.

5.1 TODO Overall design

We can split the actual design into multiple levels. Firstly we must consider the overall interaction between the code producer and the code consumer. In this interaction we will strive for doing as little work as possible inside the kernel. Specifically we want the following properties for an implementation:

1. The implementation should be correct and follow soundness of the LFSC typesystem.
2. The implementation must be both memory and runtime efficient (comparative to the verifier).
3. The implementation should be safe.
4. The implementation should be simple in nature, to minimize the risk of bugs (WELL, NICE NOT NEED?)

Moving as much computation to user-space as possible will give the best chance of an implementation that will be competitive with the verifier whilst being less code heavy and proovably correct. Unsurprisingly, most of the work still needs to reside in the kernel,

however if we require that the input must be using De Bruijn indices for bound variables we can eliminate a fraction of both memory from variable names when looking up variables. Furthermore we get equality for free, as it simply amounts to syntactical equality.

By using Rust as implementation language, we can get a lot of the requirements for free. Although it does not guarantee the implementation to be safe in terms of malicious inputs, it will greatly decrease the risk of any memory leak.

ALL OF THIS IS GARBAGE!!!

5.2 IDEA The Linux build system?

5.3 TODO Reading proofs

5.4 TODO Typechecking LFSC

5.5 TODO Plugging into the eBPF syscall

Chapter 6

TODO Experiments

- Take some examples, (for instance from Mads Thesis and check the proofs and see if the new format will let the updated ebpf syscall allow programs to load)
- obviously also check that it will reject programs that either does not match the program (if this can even happen by design), or if the proof is unsatisfiable.
- We really want the functionality to be non-blocking. Is this easy or difficult in Rust and in the Kernel?

Chapter 7

TODO Evaluation

- Videnskabsteori: Hypothesis etc.
  - Correctness
  - Expressiveness
  - Performance
  - Ease of use?
  - maybe more???

Chapter 8

Conclusion