UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Msc. Thesis

Jacob Herbst (mwr148)

Verifying eBPF programs in the Linux Kernel

Ivestigation of feasibility of checking Verification conditions for as part of the eBPF syscall

repository: `https://github.com/Spatenheinz/LeSpeciale`

Advisor: Ken Friis Larsen

2023-05-31

Abstract

eBPF (extended Berkley Package Filter) is a Linux functionality that allow users to run code of a limited assembly like language inside the Linux kernel. This can potentially be dangerous in the hands of a malicious or even uncareful users. To combat this, the Linux kernel provides a static analysis of eBPF programs to reject harmful programs. This static analysis have in the past shown to be unsound, meaning harmful programs have been accepted. In this project we investigate the feasibility of making a logical proof checker that runs inside the kernel, implemented in Rust. We do so by considering the Logical Framework with Side Conditions (LFSC) language. The motivation behind such a proof checker is to be able to formally prove the correctness of eBPF programs. We consider the feasibility not only as a standalone tool but as a hypothetical part of a larger Proof Carrying Code (PCC) architecture. By this we provide an analysis of how eBPF programs are loaded and what the static analysis consists of and hereby describe a vision for a PCC architecture. The main focus of the report is to describe and evaluate the implementation of LFSC with respect to eBPF, PCC and Rust in the Linux kernel. Although the implementation is not as fast as other LFSC proof checkers it does show decent promise for a part of a PCC system for eBPF.

Contents

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 for 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 unsafe program in the Linux kernel might break the system alltogether, whilst a malicious users could get access to confidential information. 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[15][16]. On the other hand the verifier also limits the capabilities of eBPF programs, by outright rejecting programs with loops. This is a design choice, since non-halting programs should not live in the kernel, and there is no way to tell if a program will halt using static analysis, as per the halting problem. The result is a security mechanism that is both overly conservative in some cases and unsound.

An alternative to static analysis is to verify the safety of programs using formal logic. An automatic way to do this is to reduce the problem of program verification to satisfiability of programs. This is done by generating a logical 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 cheap, however checking the validity of the formula can be a computationally heavy task. Proof Carrying Code is an architecture and security mechanism first introduced by Necula in 1998[14]. Here the user is responsible for proving that a certain program follows a set of security parameters and then the kernel only have to check this proof. The general concept is described in detail in Section **??**. All in all these tasks are far cheaper than producing a proof.

In this project we investigate the feasibility of a proof checker that can run inside the Linux kernel. The implementation not only tests this subcomponent of proof carrying code, but also test if the Rust programming language, introduced in the Linux kernel 6.1 and forward, is a good tool for the job. The proof checker leverages the Curry-howard correspondence and thus checking a proof amounts to typechecking. We specifically consider the dependently typed meta framework, Logical Framework with Side Conditions (LFSC)[17]. We describe the semantics of the typecheking and present an implementation.

The report is structured as follows. In Section 2 we give a brief introduction to eBPF, Proof Carrying Code, Rust and what limitations. Then in Section 3 we go into detail about how the eBPF verifier works and discuss Proof Carrying Code in the context of eBPF. We then present the syntax and semantics of the LFSC in Section 4, to serve as a basis for the implementation of the proof-checker. We do not prove any properties of the type system. In Section 5 we present the design choices and implementation details of the typechecker called LFSCR. We then evaluate the performance and general usability of LFSCR in a PCC system Section 7, by considering proofs generated from a simple verification condition generator, presented in Section 6. Lastly we conclude the project and present some ideas for future work in 8 and 9.

2 Background

2.1 Linux and the eBPF subsystem

The Linux kernel is a monolithic kernel, meaning that it provides not only the fundamental services necessary for a fully functioning operating system, but also a virtual interface for communication with hardware, including filesystems, network stacks, and device drivers, among others, all in a single executable. This contrast with micro kernels, where only the core functionality of the operating system exists in kernel and all other functionality lives in userspace. In both cases the standard form of communication between processes in userspace and the kernel is through system calls. This for instance happens when accessing hardware, requesting memory etc.

One major advantage of monolithic kernels is the minimal overhead of system level tasks as there is no need for interprocess communication from the different parts of the system. In a micro kernels parts of the functionality provided by a monolithic kernel live in userspace, such as filesystems. Since processes does not have direct "knowledge" of other processes, there will be an overhead when doing some system level tasks. On the other hand monolithic kernels in general provides little to no extensionality, where microkernels on the other hand is design for exactly this.

Albeit being a monolithic architecture the Linux kernel is characterized by its modular nature. The modularity is achieved either with dynamically loadable kernel modules (LKMs) or through the eBPF subsystem. LKMs are more general and can extend functionality and support a diverse range of services, such as device drivers, virtual filessystems etc. They function similarly to traditional filesystems in their execution and can be loaded and unloaded as necessary. They provide great power but at the same time no safety guarantees. Thus LKMs have inherent security risks and require root privileges, limiting their use to trusted users. A malicious LKM can destroy a kernel completely, especially considering that kernel modules may be proprietary. This creates a tradeoff between extensionality and trust.

eBPF on the other hand is more limited in capabilities but provides a platform for doing smaller tasks which still leverages the power of the kernel without the same level of risk.

2.1.1 eBPF

BPF (Berkeley Package Filter) as it was originally presented is a system for effective filtering of network packages by allowing dynamical loading of package filters from userspace into the network stack in kernel space. The extended Berkeley Packet Filter (eBPF) was later introduced in the Linux kernel, offering a different approach to kernel extension. The eBPF virtual machine leverages the privileges of the kernel to oversee the entire system, enabling more powerful control. The eBPF name furthermore refers to a just-in-time (JIT) compiled reduced instruction set computer (RISC). The ecosystem seeks to ensure security by performing static analysis of the limited language. This language is designed not to be turing complete, to reduce the number of rejected programs due to limitations in the static analysis. In eBPF there are 11 registers, r0-r10, where r0 is used for return values of functions and r10 is a readonly framepointer to a 512 byte stack. Instructions can be moves, addition and jumps in a similar manner to other reduced instructions sets.

The eBPF program loading process is presented in Figure 1. First step in the process involves obtaining a program using an abstraction tool such as BCC[3] or libbpf[12] or writing the program by hand in C macros. The `bpf` syscall is then invoked and the eBPF bytecode is moved into the kernel. The verifier then performs a series of security measures that determines if a program is accepted for loading or rejected. The security measures involves static analysis in the form of abstract interpretation using tristate numbers, cycle detection, division by zero, and more. We describe this in detail in Section 3.1. If a program is determined safe by the verifier, the program is loaded into the kernel, either for interpretation or JIT-compiled depending on the kernel configurations. The program is then attached to a hook.

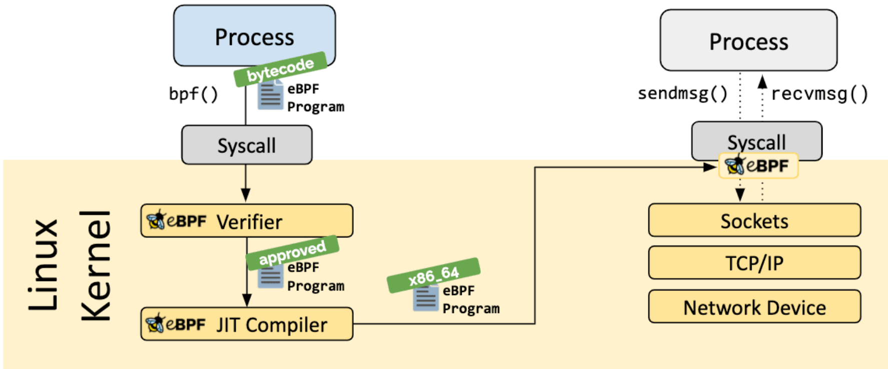

Figure 1: eBPF loading process [13]

eBPF programs are event-driven, meaning they can be attached to a certain hook, and every time the corresponding event occurs, the program is triggered. For example, a program can be attached to a socket, and every time something is written to this socket, the program is triggered. Although an eBPF program lives in kernel space, it conceptually resides somewhere between user and kernel space. It can interact with both the kernel and user space through a collection of key-value stores called maps, realized as a variety of different data structures such as ring buffers, arrays, etc. These data structures also reside in kernel space and are constructed through the `bpf` syscall, allowing eBPF programs to read and write to a map. Users can also read and write to the maps using the `bpf` syscall, thus serving as a communication layer between eBPF program and user space.

eBPF programs should likewise not been seen as a coherent part of the kernel as it cannot call kernel functions directly. The reason for this is that eBPF is designed to be kernel version agnostic[7], and Linux kernel functions are not stable. In reality it is not agnostic since different versions of the verifier will reject and accept different programs. Instead, the eBPF subsystem provides a stable API of helper functions to provide functionality not immediately accessible in the limited instruction set. Furthermore it is possible to have eBPF programs call other eBPF programs, and even chain them together in tail calls similar to standard function calls, greatly extending the functionality of eBPF programs.

This subsystem hence serves an opportunity for users to extend the kernel with functionality that can run in a sandboxed manner while still leveraging the powers of the kernel. Unfortunately due to bugs in the verifier, most major Linux distributions such as Ubuntu, Fedora, Redhat and many more disallow non-root users to load eBPF programs.

2.2 Proof Carrying Code

We are interested in investigating Proof Carrying Code as an alternative to the eBPF verifier. Proof Carrying Code (PCC) is a mechanism designed to ensure the safe execution of programs from untrusted sources.

The PCC architecture can be divided into two spaces, namely the code producer and the code consumer. The code producer aims to execute some code at the expense of the code consumer. In case there is not perfect trust between code consumer and code producer, the code consumer must protect itself from potentially malicious or unsafe programs. To achieve this, the code consumer issues a collection of safety rules, referred to as the safety policy, which the code producer must adhere to. Depending on the specific domain, the safety policy could include constraints such as loop termination, no out-of-bounds memory operations along with other constraints that ensures the code consumer is not getting corrupted. The code producer then uses the safety policy to certify the program in a compilation stage. The certification process produces both the native code to be executed at the consumer and the certificate for the safety of the native code. This process can potentially be computationally heavy, and it is preferable for the code producer to perform the certification and compilation.

The next stage is the verification phase, where the producer hands over the results of the certification process to the consumer. The consumer then checks the validity of the proof in two ways. First, the safety proof must be valid modulo the safety policies, and second, the safety proof must correspond to the native code. This process should be quick and follow an algorithm trusted by the consumer. If both criteria are met, the consumer can mark the native code as safe and proceed to execute the program, possibly multiple times.

2.3 Rust

In this section we give a short description of the Rust programming language to motivate why it is used as host language in this project. Rust is a modern systems programming language that was first introduced in 2010. It was designed to address common issues faced by developers in writing low-level, high-performance code, such as memory safety and thread safety.

One of the key features of Rust is its ownership model, which ensures that memory is managed efficiently and safely. Rust's ownership model is based on the concept of ownership and borrowing, which allows the compiler to track the lifetime of objects and manage memory without the need for a garbage collector. This ensures that common issues like null pointer dereferences, use-after-free errors and buffer overflows can occur, while at the same time being comparable to C in performance.

Rust's syntax is similar to that of C and C++, but it also includes modern language features like pattern matching, closures, and iterators. Rust also has a strong focus on performance and optimization, which makes it an ideal language for building high-performance applications and systems.

An intrinsic part of the Rust language is the borrow checker; a form of static analysis which ensures that a program complies with the ownership rules. The rules are 3 fold.

- Each value has an owner.

- There can only be one owner for each value.
- When the owner goes out of scope, its values are freed (dropped in Rust terminology).

If we for instance consider:

```
fn main() {
    let x : i32 = 42;
}
```

Then the value 42 has an owner `x`. When the `main()` function ends then the owner `x` goes out of scope and the value is dropped. This specific type of `x` is `i32` and will thus reside on the stack, however if we needed something that was heap allocated then we could create a value using a Box:

```
fn main() {
    let y : Box<i32> = Box::new(42);
}
```

Then `y` will be a `Box` type, which is the simplest form of heap allocation. Objects in rust can implement a trait, essentially an interface, called `Drop`, and define a `drop` function. When `y` goes out of scope a call to `drop` happens and the memory will be freed.

Ownership can be transfered by either `move`, `copy` or `clone`. For primitive types, those that usually reside on the stack, they can be `copied` from one variable to another. Heap allocated values can be either `cloned`, which essentially works as a `memcpy`, or by moving it. This means that the snippet below will get a compile time error at line 6, because the variable that owns the box containing 42 is no longer owned by `y` but rather by `a`.

```
fn main() {
    let y : Box<i32> = Box::new(42);
    let z = y.clone();
    assert_eq!(y, z);
    let a = y;
    assert_eq!(y, z);
}
```

Since cloning is a `memcpy` it can be pretty inefficient and should most often be avoided. Rust allow pass by reference in the form of borrowing. Borrowing literally describes the action of receiving something with the promise of return. When borrowing a value the memory address of the value is referenced by an & and is essentially just a pointer. It is worth noting that a reference differs from pointers in C in that they cannot be `NULL` and thus by mere construction eliminates `NULL` pointer problems. There are two types of references, exclusive and shared references. Exclusive references can mutate the borrowed value, while a shared reference may only read the reference. There are 3 rules that borrowing is subject to:

1. There can exist either a single exclusive reference or multiple shared references at a given time.
2. References must always be valid, which mean it is impossible the borrow a value after its owner has gone out of scope.
3. A value cannot be modified whilst referenced.

These rules invariantly ensure that a reference is always as perceived to the borrower. If multiple mutable borrows were allowed at the same time, or even simultaneously with a shared reference, then one of the mutable borrows may destroy the reference for the others. An example where this is extremely obvious is when we consider a reference to a datastructure that might need to be reallocated, such as dynamic arrays. In such a situation the address of the old array will no longer be valid. This is ensured to never happen according to rule 3.

Rust also promises zero cost abstractions and high-level features such as sum types, pattern matching and traits/interfaces, while still having performance similar to C.

These promises of memory safety and high level abstractions is what constitutes the basics of the Rust programming language. The safe memory management without the overhead of a garbage collector has gained Rust popularity in recent years, especially in the systems programming community, due to its combination of performance, safety, and ease-of-use. This popularity also includes the Linux Development community, where safety, security and reliability is mission critical. As of kernel 6.1 Rust is officially supported in the Linux kernel, albeit fairly limited as described throughout this report.

Hence Rust seems like an appropriate tool for a program that has to run in the kernel, and where robustness is critical. Bu we must also consider what restrictions the kernel imposes on Rust.

2.3.1 Rust in the kernel?

As of kernel version 6.2.8rc, the Rust kernel development framework not a lot of functionality is exposed. The crates/modules immediately exposed in the kernel is `alloc`, `core`, `kernel`, `compiler_builtins` and `macros`. The `macros` crate is tiny and exposes the ability to easily describe a LKMs meta-data. The `compiler_builtins` are compiler built in functionality which usually resides in the standard library `std`. The builtins supported in the kernel at the moment is nothing more than a panics (exceptions). The `kernel` crate exposes the kernel APIs, such as character devices, file descriptors etc. The functionality of this crate is mostly intended for use in LKMs, but does provide some features that could be used elsewhere such as random numbers etc. The `alloc` and `core` crates constitutes most of the `std` library in Rust and is respectively the implementation of a memory allocator and core functionality. The `alloc` and `core` crates are often used in embedded systems and others situations where no operating system to provide the functionality of the standard library. The `core` crate exposes basic functionality such as primitive types, references etc. The `alloc` crate exposes memory allocations and in userspace uses some exposure of malloc, while in kernel space may use either `kmalloc` or `kvmalloc` to allocate physical and virtual memory inside the kernel. In its current form the `alloc` crate does not provide much functionality. Only simple allocation types such as `Box` are exposed and their API is conservative. The reason behind is that the kernel has no way to handle Out-Of-Memory cases. Thus most datastructures are simply not allowed, because they dont expose a fallible way to allocate memory. Whenever a new allocation need to happen a `try_new()` function can be called, which will return a `Result` type with either a reference or an error. For infallible memory allocations with `new()` an out of memory will throw an exception, which there is no good way to handle. The only datastructure available is `Vec`, a dynamic array. For faster performance on lookup, we might need other datastructures. Furthermore the `alloc` crate is compiled with a `no_rc` feature meaning there is no way to use the reference counted pointers defined in Rust. The reason for this is that maintainers of the Rust functionality in Linux have decided that it is unnecessary, since the C part of the kernel already defines a reference counting functionality. To the best of my knowledge there is no clear exposure of this functionality in any of the crates available. We need reference counting for our implementation. It is easy to remove this restriction, but may make a potential PCC implementation harder to get merged into the upstream Linux.

It is possible to compile crates that support a `no_std` feature (it relies on `alloc` and `core`) and that also does no infallible memory allocations. One example of a library that does this is parser-combinator library `nom`, which we use for parsing.

2.4 TODO Curry Howard Isomorphism

3 Proof Carrying Code in the Kernel

Before we can get a grasp of what it takes to make PCC work in the kernel, one must first have a better understanding of what is actually happening in the verifier. In Section 3.1 we take a "deepdive" into the inner workings of the verifier. The purpose of this is to create a bridge between eBPF and PCC and to further emphasize why we should investigate other methods of verification than the static analysis that exist at the moment. In Section 3.2 we describe an overall design to how we can use PCC in the Linux kernel. In this section we further argue for some general design decisions taken in the process of creating the proof checking part of the PCC system.

3.1 Spelunking the eBPF verifier

In this section we describe in detail how an eBPF program is loaded. We first describe, how the `bpf` syscall is defined and we then proceed to give a general understanding of what steps is taken in the process of loading an eBPF program. Specifically it is important what requirements the programs must follow and how this is realized.

3.1.1 The bpf syscall

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

```
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 project we only care about the cmd `BPF_PROG_LOAD`, but intentions such as `BPF_MAP_-UPDATE_ELEM`, `BPF_MAP_CREATE` and `BPF_MAP_LOOKUP_ELEM` also exists.

To be able to load an eBPF program one of the following criteria must be met:

- Either a the caller of the syscall must be root or be bpf capable,

[1] bpf() has syscall number 321

- or the `kernel.unprivileged_bpf_disabled` kernel parameter must be set to 0, meaning regular users are capable of loading programs. As mentioned previously for security reasons this is disabled in most major Linux distributions.

The `attr` argument is a union of structures that must correspond to the argument type. For program loading `attr` notably contains the type of program to load, which could be socket programs, kernel probes, Express Data Path or one of the many other possibilities, as well as the program that needs to be loaded. The syscall will call the appropriate `cmd` after some sanity checks, such as wellformedness of the `bpf_attr` union with respect to the command and the `size` parameter. The first function called is `bpf_prog_load`.

3.1.2 Capabilities and Kernel configurations: `bfp_prog_load`

The main purpose of `bfp_prog_load` is to check for user capabilities and setting the parameters used in the verifier based on kernel configurations. It checks the following:

1. Normal users may only load socket programs, whilst network related programs such as XDP requires network capabilities or system administrator capabilities, and performance monitoring of hardware requires perfmon capabilities.
2. Only `bpf_capable` users may use unaligned memory access in eBPF maps.
3. eBPF programs must be between 1 and 4096 instructions. For capable users the limit is 1 million instructions.
4. The license of the program is checked and programs with GPL capabilities may call helper functions with GPL licence.
5. Programs may be either eBPF programs or BTF objects, which is typelevel information about eBPF programs. These are mainly used as debug information about eBPF programs and they are irrelevant to the verifier, thus also the work done in this report.
6. The program is checked for device boundness, as eBPF programs can be offloaded to devices.

These are all properties of an eBPF program, and must still be checked even if the verifier is not present. In the design of a new loading procedure `bpf_prog_load` should be mostly kept intact. We would however still need to modify the call to `bpf_check` as this is the entry point to the verifier.

3.1.3 Static analysis: `bpf_check`

The `bpf_check` is what we usually denotes as the verifier. Firstly the checking environment is setup. The environment is a big struct with all necessary information to complete the validation. The procedure starts by checking more capabilities, for instance we have the following lines:

```
env->allow_ptr_leaks = bpf_allow_ptr_leaks();
env->allow_uninit_stack = bpf_allow_uninit_stack();
env->bypass_spec_v1 = bpf_bypass_spec_v1();
env->bypass_spec_v4 = bpf_bypass_spec_v4();
env->bpf_capable = bpf_capable();
```

The first 4 are flags only set for perfmon capabilities, which allow perfmon capable users to do more with the eBPF stack. Environment variables such as these introduce a dilemma in terms of what can be removed from the verifier. This is discussed in 3.2.2.

After these initial flags, the function does the following checks:

1. Firstly eBPF subprograms and kernel helper functions are added to the environment. Notice here, that the main eBPF program is also considered a subprogram, so whenever we state subprograms are checked this correponds to the main as the eBPF subprograms.
2. Then function `check_subprogs` is called, where some simple checks are conduncted, such as subprograms not being allowed to jump outside of its own address space. The last instruction of a subprogram must either exit or jump. Interestingly enough a jump at the end should in general not be allowed since we may not jump backwards, and we may not jump out of the subprogram.
3. If the eBPF program is supposed to be device bound, it is prepared for that. We omit the details as we have not taken device offloaded into account for this project. But it is an additional factor to consider in the replacement of the verifier.
4. Next `bpf_check` will check the control flow graph for loops using a non-recursive depth first search approach. If a cycle is detected, the program is rejected.
5. All the subprograms are then checked according to their BPF TypeFormat (BTF).
6. The last step before loading a program is abstract interpretation using tri-state numbers (`tnum`).

The following is a simplification of the kernel documentation about the verifier[8]. A program must follow these requirements:

1. Registers may not be read unless they have previously been written. This is to ensure no kernel memory can be leaked.
2. Registers can either be scalars or pointers. After calls to kernel functions or when a subprogram ends, registers r1-r5 are forgotten and thus cannot be read before written. r6-r9 are callee saved and thus still available.
3. 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.
4. Stack space, for same reason as registers, may not be read before it has been written.
5. External calls are checked at entry to make sure the registers are appropriate w.r.t. the external functions.
6. All read and writes to the stack and maps should be within bounds.
7. Division by 0 is not allowed, unless the divisor is a register in which case the program is patched later in the verification process.

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`. 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 checked wrt. the program type they occur in. For instance `BPF_MAP_TYPE_SOCKMAP` may only be used with socket type programs.

After abstract interpretation, the stack depth is checked, meaning we simply check if the function calls can fit within the stack space allocated for the eBPF program.

Next dead code is eliminated. The argumentation in the comments for the implementation are questionable. Specifically they mention that malicious code can have dead code too, which clearly is correct, but also completely irrelevant. Especially since they are turned into `JA -1` instructions.

If all these requirements are met, then an eBPF program is loaded. This mapping is simplified a lot, but it shows that the current process of checking a valid eBPF program has many steps of which some are directly code specific and some are tied to intentions and capabilities. This makes a PCC system a little more difficult to realize.

3.2 eBPF and PCC

From the description of PCC in 2.2 and the description of the eBPF subsystem above, it should be clear that eBPF and the verifier has some clear similarities with PCC. eBPF present a clear set of security policies, which must be followed to load programs into the kernel. Likewise there is a clear distinction between the user space and kernel space or in PCC terminology between code producer and consumer. Where the current eBPF loading system differs from the PCC described by Necula is where the responsibility lies. Considering the pipeline we have:

1. **Compilation and Certification**: For PCC the untrusted program is both compiled and a certificate for safety policy compliance is generated by the code producer. eBPF does not really “do” anything at this stage as source code is passed directly to the kernel using the syscall, and then possibly JIT compiled later in the pipeline.
2. **Verification of certificate**: In PCC the consumer will check the validity of the certification wrt. the safety policy and compatability between the code and the certificate. eBPF will have to do a similar check but directly on the eBPF program. From a purely complexity wise standpoint verification by checking a proof should not be any harder than performing the static analysis done by the verifier.
3. **Running**: In both structures, once a certificate is checked the program is free to use possibly many times.

So if they are so similar in structure, why would we want to replace the verifier with an actual proof checker? As already mentioned the eBPF verifier has been prone to bugs in the past, and the code for the verifier is charaterized by patching of these bugs, instead of implementing a proved sound abstract interpretation. Having a proof checker that implements a sound approach to proof checking will both give a higher certainty in safety as well as more accepted programs, because a proof checker does not have to be as conservative as static analysis.

3.2.1 How to add PCC to eBPF

To realize PCC in the Linux kernel we must extend it in some way. There are mainly four ways we can extend the Linux kernel by the documentation[1].

1. If an operation can be achieved using one of the many filesystems already present in the kernel, then it should do so.
2. For operations that are device specific, we should consider a LKM.
3. If new functionality is to the kernel it should be in the form of a syscall.
4. If strictly necessary a system call should be modified, but its API should be the same.

The first two options are not really viable solutions for a PCC infrastructure, since it would require plugging into the subsystem in some way, and this interaction in general seem to break many responsibility patterns and would still require modifying the `bpf` syscall. Option 3 likewise imposes a responsibility mismatch. All interactions with the eBPF subsystem goes through the syscall, and having separate system calls to validate the code and load it is not optimal. Likewise the proof checker cannot substitute the entire loading process but only the verifier, and probably not all of the checks can be completely removed. For instance we might still require capabilities to be checked seperately, but include memory alignment in the verification condition for the program. On the other hand capabilities are just boolean flags, which can easily be represented in logic. Such design decisions would require more experimentation.

The most optimal solution would be to modify the `bpf` syscall directly. We can here also get partial transition by adding the feature of a proof checker and give the proof as part of the `attr` struct for the syscall. Then when confidence in the proof checker is high enough the verfier can be phased out.

3.2.2 Certifications

As previously mentioned we only implement part of PCC in this report, and this implementation should also just be seen as a prototype.

For the general architecture we consider a certification format based on formal logic of verification conditions based on predicate transformer semantics[6]. This is particularly useful for automatic generation of formulas to describe programs. The process amount to showing the satisfiability of the negated formula, since this gives validity. The process of proving the validity of such a verification condition however is not a simple task and is, dependending on the logic, undecidable. Checking the satisfiability of a formula can be done by a Satisfiability Modulo Theories (SMT) solver. In some SMT solvers it is possible to extract a proof that a certain formula is satisfiable. We have in this work considered two output formats/languages, Alethe[**alethe**] and Logical Framework with Side Conditions (LFSC)[17], 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 strutured as a box style proof, with subproofs, conclusions etc. which makes understanding the proofs easy. This is not a property necessary for a PCC architecture where we want to automate the entire process, and dont care about the plain text format, but in fact rather would want a bytecode format. By this construction the Alethe format provides a set of 91 inference rules[**alethe**], on which proofs are built, As an example rule 20 is reflection often denoted as `refl` which states equality after applying a context. This entails that an implementation must implement all rules necessary for a security policy and will not be as easily extended in the future as part of a Linux kernel.

LFSC on the otherhand is a metaframework that exploits the Curry-Howard isomorphism by dependent type theory. This metaframework allows for the security policy to be established by signatures, which encodes similar rules to the ones defined in Alethe. The metaframework allow us to implement a simple algorithm for typechecking signatures and verification conditions, which once defined does not have to be changed. New functionality can then easily be added by new signatures. 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.

Another important feature of a certification checker in the kernel is that it should be both memory and time efficient. 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 carcara[4]. This implementation is ~13500 lines of code, and while not all rules may be necesssary this specific implementation does not even support all theories present in CVC5, such as bitvectors. Bitvectors are an essential part of an eBPF verification condition since operations are bitwise. It is however worth noting that the Alethe format allows bitvectors and it is only carcara that does not handle them. LFSC has a proof checker written in C++ and its code is merely 5000 lines[11], and new signatures requires no modification to the code. In general the takeaway from this is that an Alethe proof checker is far more complex to implement and maintain than an LFSC proof checker.

Overall LFSC provides a better basis for the use case. When using LFSC in the proof checker we suggest that the process of checking that the eBPF program to be loaded correspond to the proof is done by generating a verification condition and then comparing it to the assertions in the proof. The specific formula that needs to be proved is directly imbedded in an LFSC proof. Hence we suggest that an in kernel verification condition generator generates formulas that can easily be compared with the formula from the proof. This also means that we can reuse some of the code used to implement the proof checker for the VC generation.

4 Logical Framework with Side Conditions

LFSC[17] is a extention of the Edinbourgh Logical Framework (LF)[9] 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 side conditions. In this context sideconditions refers to a functional programming language with an operational semantic which is evaluated during typechecking. Section 4.1 gives an introduction to the abstract syntax of LFSC. Section 4.2 describes the *signatures* and *contexts* under which typing is judged and then we describe the typing rules and the operational semantics of the side conditions in 4.3 4.4. Lastly we briefly introduce the concrete syntax in 4.5 and then we present a "small" example that shows that $P \wedge \neg P$ is unsatisfiable.

4.1 Abstract Syntax

Figure 2 shows the 5 categories of the LFSC language. At its core LFSC is a typed lambda calculus and it consists of terms/objects, types and kinds. The last two categories are patterns and side conditions.

Terms are denoted by *M*, *N* and *O* and are used as syntactical entities, proofs or inference rules in a logic. Types are denoted by *A* and *B* and used for classification of Terms, and to the describe judgements and assertions. Kinds are denoted by K and is used to classify types.

We use x to be a metavariable ranging over the set of variables that might occur in terms, c to denote constants in terms, i.e. free variables. a will range over constants in types. Sideconditions is denoted by *S* and *T* and *U*. Patterns describes pattern in side condition match cases and is denoted by *P*. Primitives and keywords of the side condition language are represented in **bold**. z and q are metasymbols representing integers and rationals. $*$ represents a hole, which might be any term. Both Π and λ are abstractions which bind the varaible in the body. type annotations are given by : and $\{S\ M\}$ is a pair.

$$
\begin{aligned}
K &::= \mathbf{type} \mid \mathbf{type}^c \mid \mathbf{kind} \mid \Pi x : A.K \mid \mathbf{mpz} \mid \mathbf{mpq} \\
A, B &::= a \mid A\ M \mid \Pi x : \{S\ M\}.A \mid \Pi x : A.B \\
M, N, O &::= x \mid c \mid z \mid q \mid\ * \mid M : A \mid \mathbf{let}\ x\ M\ N \mid \lambda x.M \mid \lambda x : A.M \mid M\ N \\
P &::= c \mid c\ x_1 \ldots x_n \\
S, T, U &::= x \mid c \mid -S \mid S \oplus S \mid c\ S_1 \ldots S_n \mid \mathbf{let}\ x\ S\ T \mid \mathbf{markvar}\ S \mid \\
&\qquad \mathbf{ifequal}\ S_1\ S_2\ T\ U \mid \mathbf{match}\ S\ (P_1\ T_1)\ \ldots\ (P_n\ T_n) \mid \mathbf{fail}\ S \mid \\
&\qquad \mathbf{ifneg}\ S\ T\ U \mid \mathbf{ifzero}\ S\ T\ U \mid \mathbf{ifmarked}\ S\ T\ U \mid \mathbf{ztoq}\ S \\
\oplus &\in \{+, /, *\cdot\}
\end{aligned}
$$

Figure 2: Syntactical categories of LFSC

4.2 Signatures and Contexts

In LFSC there are two constructs we use to keep track of variables and constants. We have signatures, and contexts. Signatures are used to assign kinds and types to constants, and thus defing the formal system on which terms are judged. Contexts are used to assign types to variables. we write them as in Figure 3 and use we use Σ, Σ' to denote the concatenation of the two signatures Σ and Σ' and similarly for contexts.

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

Figure 3: Syntactical categories of LFSC

The typesystem of LFSC is syntax directed meaning there exists 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 inference) or we can check that an expression has a type. All assertions has one of the following forms.

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

The validity of signatures is depicted in Figure 4. The empty signature is valid. The concatenation of singatures are valid if, Σ is valid and the constant is not present in the context. For *KIND-SIG* K must be valid in Σ, while for *TYPE-SIG* we require the A can be inferred to have type K.

$$
\text{EMPTY-SIG}\ \frac{}{\langle\rangle \checkmark}
$$

$$
\text{KIND-SIG}\ \frac{\Sigma \checkmark \qquad \vdash_\Sigma K \qquad a \notin dom(\Sigma)}{\Sigma, a : K \checkmark}
$$

$$
\text{TYPE-SIG}\ \frac{\Sigma \checkmark \qquad \vdash_\Sigma A \Rightarrow K \qquad c \notin dom(\Sigma)}{\Sigma, c : A \checkmark}
$$

Figure 4: Valid signatures

An empty context is valid under Σ given the validity of Σ. For concatenation x must not occur in Σ but can occur in Γ, and again we require that A can be inferred as any kind, as seen in Figure 5

$$
\text{EMPTY-CTX}\ \frac{\Sigma \checkmark}{\vdash_\Sigma \langle\rangle}
$$

$$
\text{TYPE-CTX}\ \frac{\vdash_\Sigma \Gamma \qquad \Gamma \vdash_\Sigma A \Rightarrow K \qquad x \notin dom(\Sigma)}{\vdash_\Sigma \Gamma, x : A}
$$

Figure 5: Valid contexts

4.3 Typing

With the signature and contexts we can define the typing rules for LFSC. For the entire section we brevitate $\vdash_\Sigma$ to $\vdash$ and it is assumed Σ is valid.

4.3.1 Lookup

Figure 6 describes type inference with respect to the signatures and contexts. That is, they contain all rules that where variables or constants are inferred wrt. the signature and context. Given that Γ and Σ is valid, we can infer the build in types **type** and **type$^{\mathbf{c}}$** to be **kind**. Notice here that **type** and **type$^{\mathbf{c}}$** are kinds which describes types, whereas **kind** describes kinds. Notice further that **kind** cannot be inferred and thus ensures that no type contains itself hence providing a consistent metalogic. constants can be inferred either as a kind or a type respcetively if they appear in the signature, whereas object level variables must occur in the context.

$$\text{TYPE}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{type} \Rightarrow \mathbf{kind}} \qquad \text{TYPEc}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{type}^c \Rightarrow \mathbf{kind}}$$

$$\text{MPZ}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{mpz} \Rightarrow \mathbf{type}} \qquad \text{MPQ}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{mpq} \Rightarrow \mathbf{type}}$$

$$\text{LOOKUP-CTX}\ \frac{\vdash \Gamma \quad x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

$$\text{LOOKUP-KIND-SIG}\ \frac{\vdash \Gamma \quad a : K \in \Sigma}{\Gamma \vdash a \Rightarrow K} \qquad \text{LOOKUP-TYPE-SIG}\ \frac{\vdash \Gamma \quad c : A \in \Sigma}{\Gamma \vdash c \Rightarrow A}$$

Figure 6: Typing rules for looking up types.

4.3.2 Terms and Types

Figure 7 describes the bidirectional typing of LFSC.

- *ANN* states that given object *M* can be checked to have type *A*, then the annotation can be inferred to type *A*.
- *LAM-ANN* states that a bound variable *x* has type *A* in *M* can be inferred as a $\Pi x : A.B$, given that *A* is **type** and that *M* can be inferred as *B* with $x : A$ added to the environment.
- Lambda abstractions, *LAM*, is the only type that we cannot directly infer, but instead must be checked to be a function type. A lambda a is valid function type if the body *M* can be inferred as *B* with $x : A$ added to the context.
- Both *TYPE-APP* and *APP* states that the function point (left construct) must be a function type and that the argument (right construct) can be checked to be the type of domain of the function type, then an application can be inferred as the substitution of the operand with the *x* in the construct describing the range of the function.

- *APP-SC* is where the type system of LFSC differs from that of LF. If the type of M is a function type, where the range itself is a function type with a side condition as domain, then N will be checked to have type A and the sidecondition S is evaluated by its operations semantics, and must result in O.

$$\text{Z}\ \frac{\vdash \Gamma}{\Gamma \vdash z \Rightarrow \mathbf{mpz}} \qquad \text{Q}\ \frac{\vdash \Gamma}{\Gamma q \Rightarrow \mathbf{mpq}}$$

$$\text{ANN}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash M : A \Rightarrow A}$$

$$\text{PI}\ \frac{\Gamma \vdash A \Leftarrow \mathbf{type} \quad \Gamma, x : A \vdash C \Rightarrow \alpha \quad \alpha \in \{\mathbf{type}, \mathbf{type^c}, \mathbf{kind}\}}{\Gamma \vdash \Pi x : A.C \Rightarrow \alpha}$$

$$\text{PI-SC}\ \frac{\Gamma \vdash S \Rightarrow \mathbf{type} \quad M \Rightarrow \mathbf{type} \quad \Gamma \vdash B \Rightarrow \mathbf{type}}{\Gamma \vdash \Pi x : \{S\ M\}.\ B \Rightarrow \mathbf{type^c}}$$

$$\text{TYPE-APP}\ \frac{\Gamma \vdash A \Rightarrow \Pi x : B.\ K \quad \Gamma \vdash M \Leftarrow B}{\Gamma \vdash A\ M \Rightarrow [M/x]K}$$

$$\text{APP}\ \frac{\Gamma \vdash M \Rightarrow \Pi x : A.\ B \quad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash M\ N \Rightarrow [N/x]M}$$

$$\text{APP-SC}\ \frac{\Gamma \vdash M \Rightarrow \Pi x_1 : A.(\Pi x_2 : \{S\ O\}.B) \quad \Gamma \vdash N \Leftarrow A \quad |\Sigma| \vdash \epsilon; [M/x]S \downarrow [M/x]O; \sigma}{\Gamma \vdash M\ N \Rightarrow [N/x_1]B}$$

$$\text{LAM-ANN}\ \frac{\Gamma \vdash A \Rightarrow \mathbf{type} \quad \Gamma, x : A \vdash M \Rightarrow B}{\Gamma \vdash \lambda x : A.\ M \Rightarrow \Pi x : A.\ B}$$

$$\text{LAM}\ \frac{\Gamma, x : A \vdash M \Rightarrow B}{\Gamma \vdash \lambda x.\ M \Leftarrow \Pi x : A.\ B}$$

Figure 7: Bidirectional typing rules for LFSC

Here $[M/x]K$, denotes the substitution of M with x in K. The letter C is either a type or a kind. $|\Sigma|$ denotes all sidecondition function definitions in Σ.

4.3.3 Side conditions

For easier readability we split the side condition language into three subcategories:

1. **numerical** side conditions focused around numerical values.
2. **side effects** side conditions can update the state in evaluation.
3. **compound** side conditions are construct that dont fall into the other categories.

For numerical sideconditions the rules are straight forward. They are represented for **integer** in 8 but work similarly for rational, except for Z-TO-Q, which as the name suggest requires S to be of type **integer**. *IFNEG* and *IFZERO* are branching constructs where the condition must be one of the two number types and the branches must have the same type.

For side effects rules we have that *LET-SC* works as its counter part, and *DO* is mere syntactical sugar for *LET* but with the binding of a variable. *MARKVAR* will simply return the type the

$$\text{INT}\ \frac{}{\Gamma \vdash n \Rightarrow \textbf{integer}} \qquad \text{NEG}\ \frac{\Gamma \vdash S \Rightarrow \textbf{integer}}{\Gamma \vdash -S \Rightarrow \textbf{integer}} \qquad \text{Z-TO-Q}\ \frac{\Gamma \vdash S \Rightarrow \textbf{integer}}{\Gamma \vdash \textbf{ztoq}\ S \Rightarrow \textbf{rational}}$$

$$\text{BINOP}\ \frac{\Gamma \vdash S \Rightarrow \textbf{integer} \qquad T \Rightarrow \textbf{integer}}{\Gamma \vdash S \oplus T \Rightarrow \textbf{integer}} \oplus \in \{+, *, /\}$$

$$\text{IFNEG}\ \frac{\Gamma \vdash S \Rightarrow \textbf{integer} \qquad \Gamma \vdash T \Rightarrow A \qquad \Gamma \vdash U \Rightarrow A}{\Gamma \vdash \textbf{ifneg}\ S\ T\ U \Rightarrow A}$$

$$\text{IFZERO}\ \frac{\Gamma \vdash S \Rightarrow \textbf{integer} \qquad \Gamma \vdash T \Rightarrow A \qquad \Gamma \vdash U \Rightarrow A}{\Gamma \vdash \textbf{ifzero}\ S\ T\ U \Rightarrow A}$$

Figure 8: Typing rules for numerical sideconditions

inner sidecondition and *IFMARKED* will check the "marked" sidecondition *S* and check that the two branches have the same type, resulting in the type of the branches.

$$\text{IFMARKED}\ \frac{\Gamma \vdash S \Rightarrow A \qquad T \Rightarrow B \qquad U \Rightarrow B}{\Gamma \vdash \textbf{ifmarked}\ n\ S\ T\ U \Rightarrow B}$$

$$\text{MARKVAR}\ \frac{\Gamma \vdash S \Rightarrow A}{\Gamma \vdash \textbf{markvar}\ n\ S \Rightarrow A}$$

$$\text{Let}\ \frac{\Gamma \vdash S \Rightarrow A \qquad \Gamma, x : A \vdash T \Rightarrow B}{\Gamma \vdash \textbf{let}\ x\ S\ T \Rightarrow B}$$

$$\text{DO}\ \frac{\Gamma \vdash S \Rightarrow A \qquad \Gamma \vdash T \Rightarrow B}{\Gamma \vdash \textbf{do}\ S\ T \Rightarrow B}$$

Figure 9: Typing rules for sideeffects

Compound sideconditions may be a **fail**, described by the *FAIL* rule, which simply typechecks the inner value. The reason **fail** must take an argument is that we have no polymorphic type $\forall \alpha . \alpha \rightarrow \alpha$ under dependent types. For *IFEQ* S_1 and S_2 must have the same type and the branches T_1 and T_2 must equally have the same type. Match statement work similar to other functional languages and given the scrutinee S can be inferred as A then for all match cases P_i must also be inferreable as *A*, whilst all branches T_i must be inferreable to the same type *B*. $\textbf{ctx}(P_i)$ describes the context created from P_i. concretely $\textbf{ctx}(P_i) = \langle\rangle, x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$, when $P_i = cx_1\ x_2\ \ldots\ x_n$ For applications the function point must not be a dependent function.

$$\text{SCAPP}\ \frac{\Gamma \vdash S \Rightarrow \Pi x : A.B \qquad \Gamma \vdash T \Rightarrow C \qquad x \notin FV(B)}{\Gamma \vdash S\ T \Rightarrow B}$$

$$\text{MATCH}\ \frac{\Gamma \vdash S \Rightarrow A \qquad \forall i \in \{1, \ldots, n\}.(\Gamma \vdash P_i \Rightarrow A \qquad \Gamma, \textbf{ctx}(P_i) \vdash T_i \Rightarrow B)}{\Gamma \vdash \textbf{match}\ S\ (P_1, T_1) \ldots (P_n, T_n) \Rightarrow B}$$

$$\text{IFEQ}\ \frac{\Gamma \vdash S_1 \Rightarrow A \qquad \Gamma \vdash S_2 \Rightarrow A \qquad \Gamma \vdash T_1 \Rightarrow B \qquad \Gamma \vdash T_2 \Rightarrow B}{\Gamma \vdash \textbf{ifequal}\ S_1\ S_2\ T_1\ T_2 \Rightarrow B}$$

$$\text{FAIL}\ \frac{\Gamma \vdash A \Rightarrow \textbf{type}}{\Gamma \vdash \textbf{fail}\ A \Rightarrow A}$$

Figure 10: Typing rules for compound sideconditions

4.4 Operational Semantics of Sideconditions

[2]The operational semantics is show in Figure 11. The operational semantics are under the judgement of $\Delta \vdash \sigma_1; S \downarrow T; \sigma_n$, where Δ describes all program definitions and σ_1 and σ_n, describes states mapping symbols to markings, where σ_1 is the state S is evaluated under, and σ_n is the state where S has been evaluated to T. Markings are simply a collection of 32 boolean flags, which can then be used in **ifmarked** conditions. The Δ is elided in all cases where it is unused.

Errors are not included in the operational semantics. Errors might occur when **fail** is evaluated, a scrutinee does not match any pattern, a **markvar** or **ifmarked** does not evaluate to a variable or if division by 0 occurs.

For a brief rundown of the rules we have:

- *CST-O*, *VAR-O* and *NUM-O* simply evaluate to itself and the store is unchanged.
- *CST-APP* applies a constant to n sideconditions, update the store with respect to all of them and the resulting value is the constant applied to each updated *S'*.
- *LET-O* and *DO-O* evaluates *S* and then evaluates *T* with the updated store and in the case of *LET-O* by substituting occurences of *x* in T.
- the two rules *IFEQUAL-T* and *IFEQUAL* describes a standard semantic for equality checks. Two terms S_1' and S_2' are considered equivalent with respect to \$$\beta\eta$\$-equivalence.
- Match constructs evaluate the scrutinee S and matches the result with one of the patterns. If a pattern matches then the given branch is evaluated.
- *FUN-APP* refers to the application of a side condition program. *f* must be a program in Δ and all arguments are evaluated and then results are substituted into the body of the program T and it is evaluated.
- *BINOP-O* and *NEG-O* works similar to any other language.
- *TOQ-O* evaluate S to an integer and then make the rational z/z.
- *IFNEG* and *IFZERO* rules are very similar, based on the evaluation of S, either branch *T* or branch *U* is evaluated.
- *IFMARKED* is again similar however the branching depends on the marking of variable *x* in the store.
- *MARKVAR-O* is the only rule that can update the store, by simply switching the flag for the specific mark.

[2]hvad skal jeg skrive som introducerende tekst

$$\text{CST-O}\ \frac{}{\sigma_1; c \downarrow c; \sigma_1} \qquad \text{VAR-O}\ \frac{}{\sigma_1; x \downarrow x; \sigma_1} \qquad \text{NUM-O}\ \frac{}{\sigma_1; r \downarrow r; \sigma_1}$$

$$\text{CST-APP}\ \frac{\forall i \in \{1, \ldots, n\}.(\sigma_i; S_i \downarrow S'_i; \sigma_{i+1})}{\sigma_1; (c\ S_1 \ldots S_n) \downarrow (c\ S'_1 \ldots S'_n); \sigma_{n+1}}$$

$$\text{LET-O}\ \frac{\sigma_1; S \downarrow S'; \sigma_2 \qquad \sigma_2; [S'/x]T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{let}\ x\ S\ T) \downarrow T'; \sigma_3} \qquad \text{DO-O}\ \frac{\sigma_1; S \downarrow S'; \sigma_2 \qquad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{do}\ S\ T) \downarrow T'; \sigma_3}$$

$$\text{IFEQUAL-T}\ \frac{\sigma_1; S_1 \downarrow S'_1; \sigma_2 \qquad \sigma_2; S_2 \downarrow S'_2; \sigma_3 \qquad S'_1 \equiv S'_2 \qquad \sigma_3; T_1 \downarrow T'_2; \sigma_4}{\sigma_1; (\textbf{ifequal}\ S_1\ S_2\ T_1\ T_2) \downarrow T'_2; \sigma_4}$$

$$\text{IFEQUAL-F}\ \frac{\sigma_1; S_1 \downarrow S'_1; \sigma_2 \qquad \sigma_2; S_2 \downarrow S'_2; \sigma_3 \qquad S'_1 \not\equiv S'_2 \qquad \sigma_3; T_1 \downarrow T'_2; \sigma_4}{\sigma_1; (\textbf{ifequal}\ S_1\ S_2\ T_1\ T_2) \downarrow T'_2; \sigma_4}$$

$$\text{MATCH-O}\ \frac{\sigma_1; S \downarrow (c\ S_1 \ldots S_m); \sigma_2 \qquad \exists i.P_i = (cx_1 \ldots x_m) \qquad \sigma_2; [S'_1/x_1, \ldots, S_m/x_1]T_i \downarrow T'; \sigma_3}{\sigma_1; (\textbf{match}\ S\ (P_1 T_1) \ldots (P_n T_n)) \downarrow T'; \sigma_3}$$

$$\text{FUN-APP}\ \frac{\begin{array}{c}\forall i \in \{1, \ldots, n\}.(\Delta \vdash \sigma_i; S_i \downarrow S'_i; \sigma_{i+1}) \\ (f(x_1 : A_1 \ldots x_n : A_n) : B = T) \in \Delta \qquad \Delta \vdash \sigma_{n+1}; [S'_1/x_1, \ldots S'_n/x_n]T \downarrow T'; \sigma_{n+2}\end{array}}{\sigma_1; (f\ S_1 \ldots S_n) \downarrow T'; \sigma_{n+2}}$$

$$\text{BINOP-O}\ \frac{\sigma_1; S \downarrow r_1; \sigma_2 \qquad \sigma_2; T \downarrow r_2; \sigma_3 \qquad r = r_1 \oplus r_2}{\sigma_1; S \oplus T \downarrow r; \sigma_3} \oplus \in \{+, *, /\}$$

$$\text{NEG-O}\ \frac{\sigma_1; S \downarrow r; \sigma_2}{\sigma_1; -S \downarrow r; \sigma_2} \qquad \text{ZTOQ-O}\ \frac{\sigma_1; S \downarrow z; \sigma_2 \qquad r = z/z}{\sigma_1; \textbf{ztoq}\ S \downarrow r; \sigma_2}$$

$$\text{IFNEG–T}\ \frac{\sigma_1; S \downarrow r; \sigma_2 \qquad r < 0 \qquad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{ifneg}\ S\ T\ U) \downarrow T'; \sigma_3} \qquad \text{IFNEG–F}\ \frac{\sigma_1; S \downarrow r; \sigma_2 \qquad r \geq 0 \qquad \sigma_2; U \downarrow U'; \sigma_3}{\sigma_1; (\textbf{ifneg}\ S\ T\ U) \downarrow U'; \sigma_3}$$

$$\text{IFZERO–T}\ \frac{\sigma_1; S \downarrow r; \sigma_2 \qquad r = 0 \qquad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{ifzero}\ S\ T\ U) \downarrow T'; \sigma_3} \qquad \text{IFZERO–F}\ \frac{\sigma_1; S \downarrow r; \sigma_2 \qquad r \neq 0 \qquad \sigma_2; U \downarrow U'; \sigma_3}{\sigma_1; (\textbf{ifzero}\ S\ T\ U) \downarrow U'; \sigma_3}$$

$$\text{IFMARKED–T}\ \frac{\sigma_1; S \downarrow x; \sigma_2 \qquad \sigma_2 x \qquad \sigma_2; T \downarrow T'; \sigma_3}{\sigma_1; (\textbf{ifmarked}\ S\ T\ U) \downarrow T'; \sigma_3} \qquad \text{IFMARKED–F}\ \frac{\sigma_1; S \downarrow x; \sigma_2 \qquad \neg\sigma_2 x \qquad \sigma_2; U \downarrow U'; \sigma_3}{\sigma_1; (\textbf{ifmarked}\ S\ T\ U) \downarrow U'; \sigma_3}$$

$$\text{MARKVAR-O}\ \frac{\sigma_1; S \downarrow x; \sigma_2}{\sigma_1; (\textbf{markvar}\ S) \downarrow x; \sigma_2[x \mapsto \neg\sigma_2 x]}$$

Figure 11: Operational semantics for side conditions

4.5 Concrete Syntax

Although the concrete syntax of LFSC is not terribly important, there is no clear presentation of how signatures are supposed to represent a formal system purely from the abstract syntax and the typing rules, as there is no way to extend it from the rules. we give a brief introduction to make the understanding of the examples presented later easier to understand.

LFSC implements the core language and sideconditions as S-expressions. At the toplevel

LFSC allows the following commands.

- **define** takes two arguments the constant to be bound `c` and a term `M`. it will then bind x to the term `M` with type `A`.
- **declare** takes a constant `a` and a type `A` check `A` to be a kind, then bind `a` to `A`.
- **function** is used to define sideconditions. It takes a constant name, a pairwise list of (x A), which is the arguments to the function, then a return type and a body. It will then check the body with the parameters added to a context, and then match the type of the program body with the return type.
- **check** will take a single argument, a term or a type and then typecheck it.

The intention here is that **declare** and **function** is allowed only in the signature definitions, which defiens the formal system, and then a user proof can use **define** and **check** to construct a proof under the formal system defined in the signature.

For the term language the following symbols are used:

Abstract Syntax	Concrete Syntax
Π x : A . B	! x A B
A : M	: A M
λ x : M N	# x M N
λ x N	\ x N
let x M N	@ x M N
{ S T }	ˆ S T
*	_

The sidecondition language directly uses the keywords marked with bold in the previous sections.

4.6 Using LFSC to show $P \wedge \neg P$ is unsatisfiable

In this Section we give an example of how LFSC can be used to construct proofs. We do so, by explaining a proof that $P \wedge \neg P$ is unsatisfiable. The proof can be encoded by the following commands:

```
(define cvc.p (var 0 Bool))
(check
(# a0 (holds (and cvc.p (and (not cvc.p) true)))
(: (holds false)
(resolution _  _  _
(and_elim _  _  1 a0)
(and_elim _  _  0 a0) ff cvc.p))))
```

Listing 1: Unsatisfiability proof of $P \wedge \neg P$ in LFSC

The program first defines a variable `cvc.p` which is an application of the function `var` which is used to define free constants under SMT-Lib, it is specifically charaterized by a unique number and a sort. This makes `cvc.p` unique. `cvc.p` describes the *P* in the formula above.

Then the proof is constructed by a check command. We first define `a0` stating that $P \wedge (\neg P \wedge \top)$ holds. The reason `true` is also included, is because SMT-lib considers applications as n-ary functions and these will be represented as a null-terminated curried form of higher order application. Specifically `and` is defined by:

```
(declare apply (! t1 term (! t2 term term)))
(declare f_and term)
(define and (# t1 term (# t2 term (apply (apply f_and t1) t2))))
```

the body of the check is then an annotation stating that line 5-7 should have the type `holds false`. We notice that there are quite a number of holes. these will get filled out as we go.

`resolution` and `and_elim` are declared as follows:

```
(declare resolution (! c1 term
                    (! c2 term
                    (! c term
                    (! p1 (holds c1)
                    (! p2 (holds c2)
                    (! pol flag
                    (! l term
                    (! r (^ (sc_resolution c1 c2 pol l) c) (holds c)))))))))
(declare and_elim (! f1 term
                  (! f2 term
                  (! n mpz
                  (! p (holds f1)
                  (! r (^ (nary_extract f_and f1 n) f2)  (holds f2))))))
```

We can from these declarations see that the 3 holes on line 5 in Figure **??**, should match parameters `c1, c2, c`, these occur later in the type and can therefore be "derived" the latter arguments. similarly for `and_elim` the hole used for argument `f1` will be filled by argument `p` because the types must match.

Since the fourth argument of the `and_elim` applications are `a0`, we can syntactically compare `a0` and `(holds f1)`, letting `f1` = `(and cvc.p (and (not cvc.p) true))`. Notice here that both applications of `and_elim` is provided with 4 arguments. But by the typing rules the inner sidecondition will be evaluated. Hence when all types are checked `nary_extract` is run. `nary_extract` is defined as:

```
(function nary_extract ((f term) (t term) (n mpz)) term
  (match t
    ((apply t1 t2)
      (mp_ifzero n
        (getarg f t1)
        (nary_extract f t2 (mp_add n (mp_neg 1))))))
)
```

It will extract the `n`-th element of an `f` application in `t`. so when $\text{nary}_{\text{extract}}$ is called with `f_-and`, `f1` = `(and cvc.p (and (not cvc.p) true))` and `1`, then by beta reduction we have: `f1` = `(apply (apply f_and cvc.p) (and (not cvc.p) true))`. The only branch of `nary_extract` matches this scrutinee and we check if n is 0. Since it is not we recursively call extract. In the next iteration we get: `t` = `(apply (apply f_and (not cvc.p)) true))`, now `n` is also 0 and we call `getarg` defined as follows:

```
(function getarg ((f term) (t term)) term
  (match t ((apply t1 t2) (ifequal t1 f t2 (fail term)))))
```

`t` now is `(apply f_and (not cvc.p))`. Since `t1` and `f` is both `f_and` we therefore get `(not cvc.p)` back. This is then checked agains `f2` (in the run case of `and_elim`) and the body `(holds f2)` is then checked. By similar approch we get `cvc.p` from the other application of `and_elim`.

By now we have established all the arguments to `resolution`, `c1` = `(not cvc.p)`, `c2` = `cvc.p`, `c` is still a hole and `p1` and `p2` is `(holds (not cvc.p))` and `(holds cvc.p)` respectively. and `sc_resolution` is evaluated.

```
(function sc_resolution
    ((c1 term) (c2 term) (pol flag) (l term)) term
  (nary_elim f_or
    (nary_concat f_or
      (nary_rm_first_or_self f_or
         (nary_intro f_or c1 false)
         (ifequal pol tt l (apply f_not l))
          false)
      (nary_rm_first_or_self f_or
          (nary_intro f_or c2 false)
          (ifequal pol tt (apply f_not l) l)
           false)
    false)
  false))
```

at the inner most applications we have calls to `nary_intro` which lifts a value into `n`-ary form. The two calls will turn `c1` into `(or c1 false)` and vice versa. `nary_rm_first_-or_self` will then check if the result of `nary_intro` is equivalent to the `ifequal` call and return the fourth argument `false` if that is the case otherwise it will remove the first occurence of `f_or`. For the application in line 4, the arguments are `(or (not cvc.p) false) (apply f_not cvc.p) false`. The first two arguments are not equivalent and thus we remove the first occurence of `f_or` in the first argument. β-reduced the term is `(apply (apply f_or (not cvc.p)) false)` hence the result will be `false`. For line 5 we equally get `false`.

`nary_concat` will then also return `false`, since the arguments are not `n`-ary applications. We can then clearly not eliminate any `f_or` from the expression, thus the result becomes `false`.

We match hole `f_2` in `resolution` with the result and get `holds false` which is equivalent to the annotated type, hereby concluding the proof.

From this small example, the takeaway should be that sideconditions can be useful because they allow recursive structure, meaning we for instance can extract the n^{th} clause of a conjuction, instead of applying conjunction elimination n times.

5 The in-kernel proof checker - LFSCR

In this section we present the implementation, LFSCR. We first some high level design decisions taken to reduce the amount of memory needed and to make the typechecking algorithm efficient. We then discuss the actual representation of the language and how we process it in terms of parsing and typechecking. In this process we discuss why we have chosen the specific Rust features we have with respect to the Linux kernel.

We uses normalization by evaluation with De Bruijn indices and explicit substitutions.

5.0.1 De Bruijn Indicies

From the description of the type semantics in **??**, we notice that for a type to be correctly checked it must have definitional equality. Using De Bruijn indices makes this process a lot easier since it allows for α-equivalence syntactical equivalence. Specifically with De Bruijn indices, when $\beta\eta$ normal form has been achieved, we get α-equivalence for free. De Bruijn indices even further makes the process of beta-reduction easier, as variables dont have to be renamed, i.e. α-convertion. Consider application $(\lambda x.\lambda y.xy)y$ then if we were to do direct substitution in $[y/x](\lambda y.xy)$ we would get $\lambda y.yy$ This changes the meaning of the term. De Brujin indices instead swap each bound variable with a positive integer. The meaning of the integer n is then constituted by the n^{th} enclosing abstraction, Π, λ or let. Specifically if we have the following $\lambda x.\lambda y.\lambda y.xy$ and $\lambda y.\lambda x.\lambda x.yx$ which are on $\beta\eta$-long form and alpha-equivalent but not syntactically identical. Using De Bruijn notation we get: $\lambda\lambda\lambda 20$ for both, since the function point in the inner application is described by the second inner-most binder, starting from zero, whilst the argument for the application is 0, since it is captured by the innermost abstraction. If we again consider $(\lambda x.\lambda y.xy)y$ the De bruijn representation is $(\lambda\lambda 10)y$ and by beta reduction becomes $\lambda y0$. We only consider De Bruijn notation for bound variables, this way we get around any capture avoiding complications. We could potentially also consider using De Bruijn indices for free variables, however this would complicate the code as this would require lifting binders. Although it could be interesting to consider De Bruijn levels since these are not relative to the scope.

We also consider De Bruijn indices for other binders such as program definitions, like:

```
(function sc_arith_add_nary ((t1 term) (t2 term)) term
  (a.+ t1 t2))
```

will get converted into

```
(function sc_arith_add_nary ((term) (term)) term
  (a.+ _1 _0))
```

Where _ denotes a De Bruijn index, to distinguish them from integers.

Similarly in pattern matching, when a constructor is applied to multiple arguments:

```
(match t
  ((apply t1 t2)
    (let t12 (getarg f t1)
      (ifequal t12 l tt (nary_ctn f t2 l))))
  (default ff))
```

gets converted into:

```
(match t
  ((apply 2)
    (let (getarg f _1)
      (ifequal _0 l tt (nary_ctn f _1 l))))
  (default ff))
```

Notice here that the arguments `t1` and `t2` is substituted by a 2, we need to save the number of arguments to not lose the meaning of the constructor, as they must be fully applied. In the example we also converted the binder of the "let".

5.0.2 Explicit substitutions

We have already touched upon substitution, but another matter at which we shall consider is the sheer cost of direct substitution. Performing direct substitution on terms we cause an explosion in the size of the term and unnecessarily waste both memory and execution time because we have to copy the struct at each occurence and also traverse terms multiple times. Considering explicit substitution on the other hand allow us to not generate anything unnecessarily large and keep the computation at a minimum. We consider a substitution which is lazy, meaning we use the result of a substititution, by lookup, when necessary and then proceed, but does not generate any explicit substitutions. Specifically we use Rust closures to capture Γ.

5.0.3 Normalization by Evaluation

As mentioned, we consider checking of types w.r.t. definitional equality. To do this we must have terms on normal forms, and we use the Normalization by Evaluation (NbE) for this. NbE is a process first introduced by Berger and Schwichtenberg[2] for efficient normalization of simply typed calculus, but it has since been refined for other systems in Barendregt's lambda cube. This implementation is inspired by Christensens writeup "Checking Dependent Types with Normalization by Evaluation"[5]. The technique ties a connection between syntax and semantics.

The process of evaluating a programming language amounts to either compilation to machine code and then execution or by an interpreter. In both evaluation gives meaning to said program. For instance, if the result of an interpretation is a number then the number constitutes the meaning of that program. The meaning may not be concrete but can for instance also be functions, and since we consider typechecking that can envoke evaluation values and types live in a similar domain. evaluation of LFSC can result in values for the built in types **type**, **kind** and function types such as Π types. Evaluation in general is only sensible for closed terms, but we must also consider how to handle open terms.

The process of Normalization on the other hand is to transform a program into its normal form. Letting $nf(M)$ denote the normal form of term M with $\Gamma \vdash M : A$, then the following properties must hold:

- $\Gamma \vdash nf(M) : A$
- $\Gamma \vdash nf(nf(M)) = nf(M)$
- $[\![nf(M)]\!] = [\![M]\!]$

That is the normal form has the same type as the original term. The normal form is idempotent and cannot be further normalized and the meaning does not change when normalizing a term. Many functions have these properties, so we further consider the normal form to be expressions which contains no redexes. A redex is function type applied directly to an argument. Specifically the normalization is considered with respect to β-conversion. As already mentioned the process of β-reduction is slow, since it requires multiple traversals of terms. Instead we interpret the understanding of finding a normal form can as evaluation on open terms. The result of such an evaluation will not have any meaning; hence not be a value but rather a modified term with possibly unknown values. We denote these as neutral expressions. A neutral expression in general may be free variables or application where the function point is a neutral, or in the case of LFSC, a hole. By including neutrals, we can in fact perform evaluation on open terms. We define an evaluation reflection function $T \longrightarrow [\![T]\!]$ giving mening to terms. Then to convert the meaning back into a normal form, we define a reification function $[\![T]\!] \longrightarrow T$. A normal form is then obtained by evaluation followed by reification. We describe the concrete implementation of these in 5.2.8 and 5.2.7.

5.1 Reading and formatting proofs

Because of the design decisions taken above we immediately, get a complication with the concrete syntax. Proofs generated by CVC5 will follow the concrete syntax, but we need a representation using De Bruijn indices. Therefore We propose an approach to loading proofs as seen in Figure 12. The proof is constructed by an external SMT solver, such as cvc5. We then have 3 programs in userspace:

1. A parser, that reads the concrete syntax.
2. A converter for translating the concrete syntax into a De Bruijn representation.
3. A transformer which will transform the De Bruijn representation into a format the kernel can read.

Then we have another parser in kernel space, which read the appropriate format for the type-checker.

For the prototyping we have done in this report we have not completely followed this pipeline, as the implementation still is userspace specific. But the different parts are available, except for the in-kernel parser. Ideally the data transfered from the userspace into the kernel should

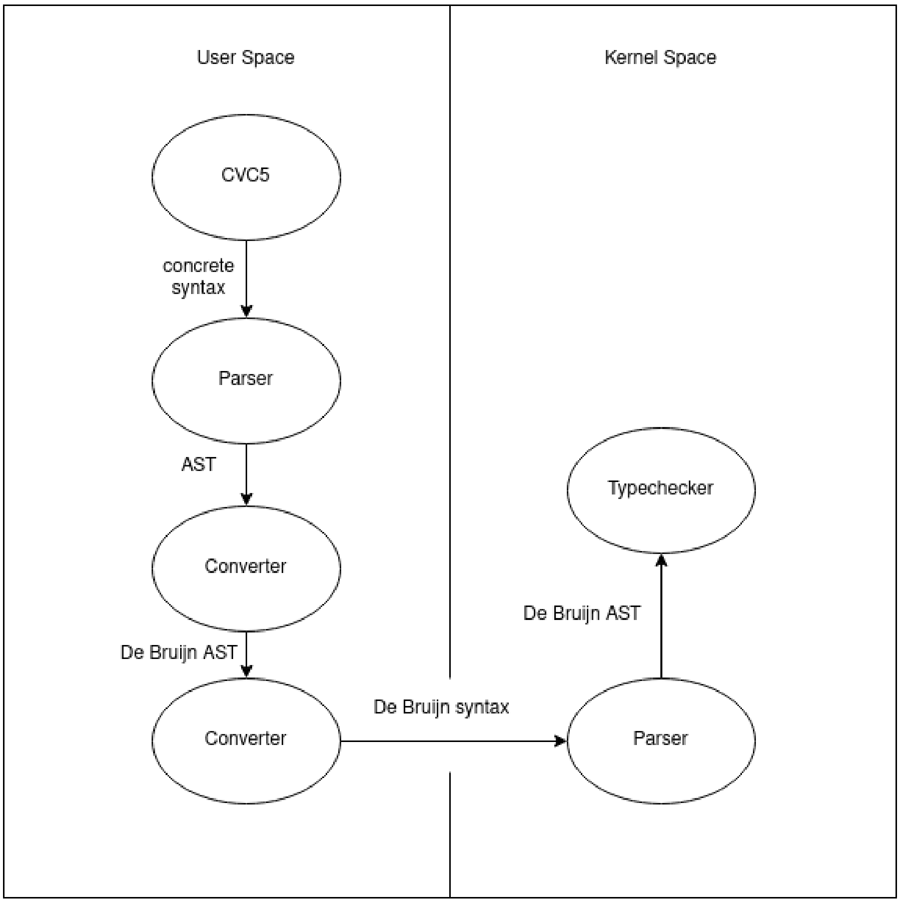

Figure 12: Data flow diagram of sending and processing a proof in the kernel.

be in a zero copy serializable format. From the general investigation into what crate can be compiled in the kernel, we have not found any of the major crates for zero copy to work out of the box, such as `Cap_N_Proto` and `Rkyv`.

5.1.1 Abstract Syntax in Rust

Despite being similar to C and CPP in syntax, Rust provides a much richer typesystem that allow us to create enumerations with fields, a.k.a Sum types. We might for instance define a construction for Identifiers as such:

```
pub enum Ident<Id> {
  Symbol(Id),
  DBI(u32)
}
```

An identifier can either be a Symbol if it is free or a De Bruijn index if it is bound. Terms are then defined almost identical to the abstract syntax. The major difference comes from the way we represent binders.

```
pub enum BinderKind {
  Pi,
  Lam,
  Let,
}
pub enum Term<Id> {
  Binder{ kind: BinderKind, var: Id,
          ty: Option<Box<Type<Id>>>,
          body: Box<Term<Id>> },
  // rest of terms
}
```

A binder is either a Π type, a λ abstraction or a let binding. We use an option type as λ abstractions might contain an annotation but can have an anonymous type aswell. Π, let and annotated λ all have a Some value for ty. We can reuse the same structure as terms and types are both defined by `Term`. This structure is convenient in the frontend representation of the language as this allow for simpler α-normalization. In the backend language we split this structure into seperate constructors of the `AlphaTerm` enum. A similar structure is used for the side condition language.

```
pub enum AlphaTerm<Id> {
    Number(Num),
    Hole,
    Ident(Ident<Id>),
    Pi(Box<AlphaTerm<Id>>, Box<AlphaTerm<Id>>),
    Lam(Box<AlphaTerm<Id>>),
    AnnLam(Box<AlphaTerm<Id>>, Box<AlphaTerm<Id>>),
    Asc(Box<AlphaTerm<Id>>, Box<AlphaTerm<Id>>),
    SC(AlphaTermSC<Id>, Box<AlphaTerm<Id>>),
    App(Box<AlphaTerm<Id>>, Box<AlphaTerm<Id>>),
}
```

We parameterize `AlphaTerm` by `Id` which is the data representation of symbols. In the specific implementation we consider a `&str`, which is a reference to a fixed sized string. We use this type over a `String` type because it is more efficient and there is no need for a term to own the string. Having terms parameterized by the `Id` type allow for easily convertion to De Bruijn levels instead of string identifiers.

5.1.2 Parsing LFSC

We use nom for parsing. nom is a parser combinator library that has evolved over the years from being mainly driven by macros to in version 7 using composable closures. It is mainly focused around parsing bytes and hereby also str.

Taking the topmost function parse_file we structure it by compostion as such:

```
pub fn parse_file(it: &str) -> IResult<&str, Vec<StrCommand>> {
    delimited(ws, many0(parse_command), eof)(it)
}
```

delimited takes 3 parsers and constructs a closure from it. When supplied with argument it, parse it with the first parser, the second and then the third and return the result of the second parser.

We can parse term binders as such:

```
fn parse_binder(it: &str) -> IResult<&str, Term<&str>> {
    alt((
        map(
            preceded(alt((reserved("let"),reserved("@"))),
                          tuple((parse_ident, parse_term, parse_term))),
            |(var, val, body)|  binder!(let var, val, body)
        ),
        map(
            preceded(alt((reserved("pi"),reserved("!"))),
                     tuple((parse_ident, parse_term, parse_term)),
            |(var, ty, body)| binder!(pi, var : ty,  body),
        ),
        ...
    ))(it)
}
```

We parse the different aspects of a binder, indentifier, binding term and the bound term and the construct the appropriate binder. For terms in general, we must ensure the parser the parser to be robust and able to handle arbitrary nesting of parenthesis. So when parsing nonterminal terms, we first parse an open parenthesis followed by parse_term_, we then try to parse a binder followed by a closed parenthesis[3]. If we fail it must be a terminal or an application if we can parse multiple terms.

```
pub fn parse_term(it: &str) -> IResult<&str, Term<&str>> {
    alt((
       parse_hole,
       map(parse_ident, |x| Term::Ident(Ident::Symbol(x))),
       map(parse_num, Term::Number),
       open_followed(parse_term_),
    ))(it)
}

fn parse_term_(it: &str) -> IResult<&str, Term<&str>> {
    if let res @ Ok(..) = terminated(parse_binder, closed)(it) {
        return res;
    }
    let (rest, head) = parse_term(it)?;
    let (rest, tail) = many0(parse_term)(rest)?;
    let (rest, _) = closed(rest)?;
```

[3]here binders also include : and

```
    if tail.is_empty() {
        Ok((rest, head))
    } else {
        Ok((rest, Term::App(Box::new(head), tail)))
    }
}
```

At the moment no parser for De Bruijn syntax exists, however modifying this parser would not require much.

5.1.3 Converting terms

To convert from `Term` to `AlphaTerm`, we traverse the AST and use an lookup table to update symbols appropriately. The lookup is simply a `Vec` of names that need be substituted. When a new binder is found we push the identifier to the end of a the vector. When we meet a symbol is met we look it up in the vector and convert it into a De Bruijn index based on its location in the vector.

```
fn lookup_(vars: &[&str], var: &str) -> Option<u32> {
    vars.iter().rev()
               .position(|&x| x == var)
               .map(|x| (x as u32))
}
```

and specifically map the option as follow:

```
pub(crate) trait Lookup<'a> {
    fn lookup(vars: &[&'a str], var: &'a str) -> Self;
}

impl<'a> Lookup<'a> for StrAlphaTerm<'a> {
    fn lookup(vars: &[&'a str], var: &'a str) -> Self {
        lookup_(vars, var).map(|x| Ident(DBI(x)))
                          .unwrap_or(Ident(Symbol(var)))
    }
}
```

One thing to note is that this approach is errorprone without careful consideration. Consider the expression: $\lambda x.((\lambda y.xy) : (\lambda z.z))$ We start by pushing x to the `vars` environment. In the body of the abstraction we have two branches to the ascription. When transforming the term, we push `y` to `vars`, then we replace `x` with the index 1 and `y` with index 0. We then get to transforming the type of the ascription, and because vectors are a mutable structure when pushing z it will lie at `vars[2]`. After taking a branch we must thus ensure to truncate the vector to its original length. For a simple solution, We define a function `local` inspired by the effectful function `local` of the Reader monad.

```
fn local<'a, 'b, Input, Output>
    (fun: impl Fn(Input, &mut Vec<&'a str>) -> Output + 'b,
     vars: &'b mut Vec<&'a str>
    ) -> Box<dyn FnMut(Input) -> Output + 'b>
{
    Box::new(move |term| {
      let len = vars.len();
      let aterm = fun(term, vars);
      vars.truncate(len);
      aterm
```

```
    })
}
```

We create a closure which takes in a term, the closure will call `fun` with the term and `vars` as arguments and then it will truncate the environment to its size before `fun` was called.

We can then use the function as such:

```
  Term::Ascription { ty, val } => {
      let mut alpha_local = local(alpha_normalize, vars);
      let ty = alpha_local(*ty);
      let val = alpha_local(*val);
      Asc(Box::new(ty), Box::new(val))
  },
```

and hereby convert the AST to include De Bruin indices.

5.2 Typechecking LFSC

In this section we describe the implementation of the type-checkign semantics. We start by introducing the value representation obtained by inference and evaluation. Then we describe the Signature and Contexts, followed by inference, type-equality and then reification and evaluation at last.

5.2.1 IDK WHERE TO PUT THIS

We use compile time references when we can to not unnecessarily create new objects. When compile time references are not possible because we dont know the owner of a value and thus also not the lifetime of it we instead use reference counted pointers. Most of the functions we describe returns values. The ownership will then lie at the caller of the function, but in some cases the owner of a result value may be Σ. This for instance happen when inferring the type of a constant. Further because of the lifetime guarantee there is no way to create a value and return a reference to it.

The reference counted smartpointer looks as follows:

```
pub struct Rc<T: ?Sized> {
    ptr: NonNull<RcBox<T>>,
    phantom: PhantomData<RcBox<T>>,
}
```

An Rc is nothing more than a struct that contains a pointer to the inner value that is referenced and a phantom field. The phantom field is merely there to keep strong static typing in a similar way to a phantom type in Haskell. The `ptr` in this struct points to the following struct:

```
#[repr(C)]
struct RcBox<T: ?Sized> {
    strong: Cell<usize>,
    weak: Cell<usize>,
    value: T,
}
```

Which contains the values and the counts for strong and week reference counts. Whenever the Rc is then cloned we simply take the RcBox inside of Rc and increment the pointer, and construct a new Rc struct. The ease of use then comes from the Drop trait which will either decrement the count in the RcBox and drop the Rc or it will drop both if the strong count is 0. Hence we can easily create new references, but we dont need to know the owner, but it does not matter since they are deallocated automatically. This gives some overhead compared to regular references but is highly likely more efficient that cloning.

5.2.2 Values

Since we consider typechecking using normalization by evaluation we need a new type to describe the result of evaluation. We define them as such:[4]

```
pub enum Value<'term, Id: BuiltIn> {
    Pi(bool, RT<'term, Id>, Closure<'term, Id>),
    Lam(Closure<'term, Id>),
    Box, // Universe
    Star,
    ZT,
    Z(i32), // TODO: should in fact be unbounded
    QT,
    Q(i32, i32), // TODO: should in fact be unbounded
    Neutral(RT<'term, Id>, Rc<Neutral<'term, Id>>),
    Run(&'term AlphaTermSC<Id>, RT<'term, Id>, Rc<LocalContext<'term, Id>>),
    Prog(Vec<RT<'term, Id>>, &'term AlphaTermSC<Id>),
}
```

Just as AlphaTerm, Value is parameterized by an Id type. Now we require Id to implement the trait BuiltIn. The trait is bound by other traits: `pub trait BuiltIn: Eq + Ord + Hash + Copy`. It must be PartialEq to be able to look up the T in the environment. It must also be Copy. We uses this stricter trait than Clone, as it allows for quick "copying" and is a satisfied criteria for both &str and u32's that could be used for De Bruijn levels. Hash and Ord is not strictly necessary but is required to use Hashmaps or Btrees for Signatures. The BuiltIn trait itself defines how the builtin types **type**, **mpz** and **mpq** is defined. For &str this is simply a stringification of the literals, for u32 prepresented De bruijn indices these may be 0,1,2.

The 'term type parameter is the lifetime of the reference to an AlphaTerm, we need this since many of the values have references to terms.

A value might be one of the abstractions in the term language, as these cannot be reduced further. Abstractions Π and λ contain a closure with type $RT \longrightarrow \Sigma \rightarrow RT$, which when constructed closes over a local context and the some term, where RT is a reference counted pointer to Value. The reason Σ must be passed as argument to the closure is purely a matter of the borrowing rules. If a closure is constructed with a reference to Σ then we cannot extend Σ anymore. The Pi value further contain its domain and a boolean value. The boolean describes the freeness of the variable in the term captured by the closure. This is extremely important for performance reasons which we describe more in detail in Section **??**.

Values can then also be one of the built in types, where

[4]Notice here that Z and Q should actually have unbounded integers as fields

- `Box` correspond to keyword **kind**
- `Star` correspond **type**.
- and `ZT` and `QT` is **integer** and **rational**.

It can then be a value of Z or Q or a `Run`, which is simply a sidecondition $\{S\ M\}$.

Neutral expressions, consists of an RT which is the type describing it, and a the neutral expression it describe. The `Neutral` type be either a neutral variable of global or local scope, It can then be a hole, or an application of a neutral term to a normal form.

```
#[derive(Debug, Clone)]
pub enum Neutral<'a, T: Copy>
{
    Var(T),
    DBI(u32),
    Hole(RefCell<Option<RT<'a, T>>>),
    App(Rc<Neutral<'a, T>>, Normal<'a, T>),
}

#[derive(Debug, Clone)]
pub struct Normal<'a, T: Copy>(pub Rc<Type<'a, T>>, pub Rc<Value<'a, T>>);
```

Lastly `Value` can be programs and run commands. Programs cannot directly be constructed by inference or evaluation, instead `Program` is used to describe the type of a side condition program, since the `Pi` constructor is insufficient. `Run` is defined for pure convenience.

5.2.3 Signature and Contexts

Signatures Σ is denoted as `GlobalContext` while Γ is used for the `LocalContext`. They have a similar interface but internally works quite differently. A global context is defined as such:

```
pub struct GlobalContext<'term, K: BuiltIn> {
    kvs: HashMap<K, TypeEntry<'term, K>>
}
```

The only field of the struct is a `Hashmap` with key-value pairs. We only define it as a hashmap because the implementation does not live in the kernel. For a version compatible with the kernel, we would use a `Vec` or implement a hashmap that works in the kernel. In any case, the interface is the same.

A type entry is defined as:

```
pub enum TypeEntry<'term, Key: BuiltIn>
{
    Def { ty: RT<'term, Key>, val: RT<'term, Key> },
    IsA { ty: RT<'term, Key>, },
    Val { val: RT<'term, Key> },
}
```

Notice here that this does not directly correspond to our definition in 4.2.

- The `IsA` construct corresponds directly to $x : A$, while the other two are defined purely for ease of use.

- The `Def` constructor is used for definitions stating that a constant *c* is a term *M* with type *A*. We mainly use this for top-level definition aswell as type inference of let bindings.
- The `Val` is used in extending the environment in evaluation. This constructor can never occur in Σ but may occur in Γ. Having this constructor allow us to reuse Γ for evaluation.

The global context exposes the following functions:

```
pub fn insert(&self, key: K, ty: RT<'term, K>)
pub fn define(&self, name: K, ty: RT<'term, K>, val: RT<'term, K>)
pub fn get_value(&self, key: &K) -> ResRT<'term, K>
pub fn get_type(&self, key: &K) -> ResRT<'term, K>
```

If one tries to get the value of a `IsA` type, they get a neutral expression consisting of the stored type `ty` and a neutral symbol of the key. On the other hand if one tries to get the type of `Val` then an error occurs.

The local context exposes a much similar interface but with a different underlying datastructure. The local context is implemented as a linked list using reference counted pointers and much more closely represent the concatenation of Γ presented in 4.2.

```
pub enum LocalContext<'a, K: BuiltIn> {
    Nil,
    Cons(TypeEntry<'a, K>, Rlctx<'a, K>),
}
```

Most functions we use such as `eval`, `infer` etc, needs to have access to both Σ and Γ and thus for simplicity we define the following wrapper.

```
struct EnvWrapper<'global, 'term, T: Copy> {
    pub lctx: Rlctx<'term, T>,
    pub gctx: Rgctx<'global, 'term, T>,
    pub allow_dbi: u32,
}
```

Here `Rlctx` is a type synonym for a reference counted Γ. Whereas `Rgctx` is a standard reference to Σ, with lifetime `'global`.

5.2.4 Commands

Before explaining the inference algorithm we should quickly describe how commands are handled. We define a single function to handle a specific command and then apply this on an iterator of all commands presented to the proof checker. The function starts by constructing the environment wrapper, with the current Σ and an empty Γ.

- For declarations we first check that the constant we want to bind $a \notin dom(\Sigma)$ and then infers the type to make sure $a : K$ or $a : A$. We then evaluate the expression and insert it, as an `IsA`.
- Definitions is similarly first typechecked, but must not be of a kind level. They are then stored in the global environment as `Def` where the value is the evaluation of the term.
- Checks is nothing more than infering the type to check for well-typedness.

- Programs are complicated for multiple reasons. As for other modifications to Σ we check that the identifier is not in $dom(\Sigma)$. We check that the return type of a program is a **type**. We then check each argument against the empty Γ and add them to another Γ' which will be used for checking the body. By this we ensure that parameters does not depend on each other. Then before we can typecheck the body, we must first add it Σ, since side condition programs may be recursive. Then we have to drop the `EnvWrapper` because it borrows Σ immutably and we want to borrow it mutably to insert an entry. We insert symbol id as a `Def` with the type as the return type of the function and a `Prog` type as value. We can then check the body to have the return type.

```
let env = EnvWrapper::new(Rc::new(LocalContext::new()), gctx, 0);
    ... other cases ...
    Command::Prog { cache: _chache, id, args, ty, body } => {
      ... typesignature check ...
      let lctx = tmp_env.lctx.clone();
      drop(tmp_env);
      let typ = Rc::new(Value::Prog(args_ty.clone(), body));
      gctx.define(id, res_ty.clone(), typ);
      EnvWrapper::new(lctx, gctx).check_sc(body, res_ty)?;
      Ok(())
```

5.2.5 Inferring types.

To infer types of LFSC we define an `infer` function for each of the constructs in the language, The functions are implemented as inherent implementations (concrete associated functions) and has the types:

```
impl<'global, 'term, T> EnvWrapper<'global, 'term, T>
where T: BuiltIn
{
    pub fn infer(&self, term: &'term AlphaTerm<T>) -> ResRT<'term, T>
    pub fn infer_sc(&self, sc: &'term AlphaTermSC<T>) -> ResRT<'term, T> {
    fn infer_sideeffect(&self, sc: &'term AlphaSideEffectSC<T>) -> ResRT<'
    term, T>
    fn infer_compound(&self, sc: &'term AlphaCompoundSC<T>) -> ResRT<'term, T
    >
    fn infer_num(&self, sc: &'term AlphaNumericSC<T>) -> ResRT<'term, T>
}
```

The functions follow closely the rules in Figure 7. Taking `infer` as example it pattern-matches on `term`, and for λ, side conditions and holes the inference fails. And variables and symbols are simply looked up in either Σ or Γ respectively. For some of the more complicated rules:

inferring a `Pi` will check if the domain is a sidecondition. If that is the case, we infer the type of the side-condition and check it to have the same type as type of the second part of the pair. We create a Run type for the domain. In case it is not a sidecondition, we simply infer the domain to have **type** and then evaluate it, to get its value. We can then update the local environment stating that De Bruijn index 0 in the localcontext `IsA val` type.

```
AlphaTerm::Pi(a, b) => {
    let val =
        if let SC(t1, t2) = &**a {
            let t1_ty = self.infer_sc(t1)?;
            self.check(t2, t1_ty.clone())?;
            Rc::new(Type::Run(t1, t1_ty, self.lctx.clone()))
```

```
        } else {
            self.infer_as_type(a)?;
            self.eval(a)?
        };
    self.update_local(val).infer_sort(b)
},
```

For application instead of interpreting multiple continious application as curried form, we use a flat approach in which we evaluate each argument in a loop. First we infer the function point, this is saved in a mutable variable updated each iteration of the following loop. Each argument is checked against the domain of the Π type. If the variable bound by the Π is free in the body, then we evaluate it. Otherwise we return an arbitrary value. Lastly the body is evaluated, with either the new value or a default added to the local environment. The result is bound to `f_ty`. This happens for each iteration and lastly we return the bound value. In case the argument is a Hole, we do not check it, as it is trivially the correct type at this point. We add it to the environment and evaluate the body.

```
App(f, args) => {
    let mut f_ty = self.infer(f)?;
    for n in args {
        f_ty = if let Type::Pi(free,a,b) = f_ty.borrow() {
         if Hole == *n {
             let hole = Rc::new(Neutral::Hole(RefCell::new(None)));
             b(Rc::new(Type::Neutral(a.clone(), hole)), self.gctx)?
         } else {
            self.check(n, a.clone())?;
             let x = if *free { self.eval(n)? } else { a.clone() };
            b(x, self.gctx)?
         }
        } else {
            return Err(TypecheckingErrors::NotPi)
        }
    };
    Ok(f_ty)
```

similar inference is done for the sidecondition language.

5.2.6 Checking types

We define two functions for typechecking. One takes a term `term` and a `Value t2` and check against it. The other takes a sideconditions as first argument, but essentially does the same.

We match on `term` and if it an anonymous lambda then we check *LAM*-rule of Figure 7, otherwise we infer the type of the `term` to `t1` and check `t2` and `t1` for definitional equality.

This process is two-fold. Firstly we do a value comparison between `t1` and `t2`. If not succesful we convert them to their canonical form using reification and check for equality. Generally we cannot compare values, as functions such as `Pi` has a closures inside it that cannot easily be compared. The main reason for the `ref_compare` call, is to fill holes. The function return a boolean of the equality between the two values, and as long as the values are one of the simple types or syntactically same neutrals it returns true. If one of the arguments is a hole it is filled with the other value. This is done using the interior mutability of Refcells. We must do it this way, as holes cannot be filled after reading back values as a hole might occur in multiple places and we must ensure that it is filled with the same value. Empirically, the `ref_compare` will return `true` most of the time. In case it returns `false`, we reify the values into their normal form and compare them.

```
pub fn convert(&self,
            t1: RT<'term, T>,
            t2: RT<'term, T>,
            tau: RT<'term, T>) -> TResult<(), T>
{
    if ref_compare(t1.clone(), t2.clone()) { return Ok(()) }
    let e1 = self.readback(tau.clone(), t1)?;
    let e2 = self.readback(tau, t2)?;
    if e1 == e2 {
        Ok(())
    } else {
        Err(TypecheckingErrors::Mismatch(e1, e2))
    }
}
```

5.2.7 Readback (Reification)

Reification or reading back semantical objects into the term language is type directed. `read_back` takes a type and a value as arguments. first the term to be read back is check to be neutral, if that is the case then we call `read_back_neutral`, since `Neutral`'s encode their own type.

Variables can be read back directly, holes either get read back as its inner value or as a term language hole. Application will read back the function point and then argument point. And construct an application from it. Be aware here that the argument point is a `Normal` type. It will thus call `read_back` with its type and val.

```
fn readback_neutral(&self, neu: Rc<Neutral<'term, T>>)
                    -> TResult<AlphaTerm<T>, T>
{
    match neu.borrow() {
        Neutral::DBI(i) => Ok(Ident(DBI(*i))),
        Neutral::Var(name) => Ok(Ident(Symbol(*name))),
        Neutral::Hole(hol) => {
            if let Some(ty) = &*hol.borrow() {
                self.readback_neutral(ty.clone())
            } else { Ok(Hole) }
        },
        Neutral::App(f, a) => {
            let f = self.readback_neutral(f.clone())?;
            let a = self.readback_normal(a.clone())?;
            Ok(App(Box::new(f), vec![a]))
        },
    }
}
```

If the value of `read_back` is not neutral, we patternmatch on the type. If the type is Z or Q then we can read back integers and rationals respectively. All built in types can be readback to the built in terms. Pi types can be read back by reading back the domain, then evaluate the body and read the body back.[5]

[5]should i add more here?

5.2.8 evaluation

We define evaluation on two levels, both on terms and on the functional side condition language. Evaluation of the term language is straightforward. Sideconditions and holes cannot be evaluated. Application have a similar structure to `infer`. We consider applications in applicative order evaluation with a loop over the arguments. The function `do_app` is then called with the function and the evaluated argument:

```
pub fn do_app(&self, f: RT<'ctx, T>, arg: RT<'term, T>) -> ResRT<'term, T>
{
match f.borrow() {
    Value::Lam(closure) => closure(arg, self.gctx),
    Value::Neutral(f, neu) => {
        if let Value::Pi(_,dom, ran) = f.borrow() {
            Ok(Rc::new(Value::Neutral(
                ran(arg.clone(), self.gctx)?,
                Rc::new(Neutral::App(neu.clone(), Normal(dom.clone(), arg))))
    ))
        } else {
            Err(super::errors::TypecheckingErrors::NotPi)
        }
    }
    _ => Err(super::errors::TypecheckingErrors::NotPi)
}
}
```

Functions can be either a concrete lambda abstraction in which we simply evaluate the closure or a function can also be unknown. If this the case, we check that the type of the neutral value is a function type. We construct a new neutral value, where the type of the neutral value is the range of the Π and the value is an application of the unknown value onto the normal expression (`dom`, `arg`). stating that `arg` has type `dom`. For instance if we consider the application (f x y), where f :: a -> b -> c, then we construct:

$$Neu(b \to c, f(x : a))$$

by the first application and

$$Neu(c, (f(x : a))(y : b))$$

after the second application. To come full circle, if we want to read back, we will get (f x y) since it is already in normal form.

Evaluation of Π terms are also interesting. For standard Π constructs, we simply evaluate the domain, construct a closure around the body, check if the bound variable is free in the range and construct a Pi value. More interesting, if the domain of a Π is a sidecondition, we evaluate the sidecondition and the target and check for equivalence. Lastly the result is inserted in the local context and the body is evaluated. This is what enables execution of sideconditions in the typechecking. [6]

6 Experiments

Since the purpose of this project is to check the feasibility of an in-kernel proof-checker that can replace the eBPF verifier, we want to evaluate the implementation with proofs that resemble

[6] should i mention anything about evaluation of sideconditions?

what would occur in a PCC context. To do so, we consider a simple verification condition generator based on the weakest precondition predicate transformers. Specifically we consider a limited subset of eBPF consisting only of the following instructions:

$$\begin{aligned}(\text{Mov } r_d &:= src \\ (\text{Update} r_d &:= r_d \oplus src \\ (\text{Neg and assign} r_d &:= -src \\ &\oplus \in \{+, -, **, /, mod, xor, \&, |, \ll, \gg\}\end{aligned}$$

r_d denotes an arbitrary register, and src may be either, a register or a constant value of either 32 or 64 bits. Instructions can be:

- A move from src into r_d.
- A negation of a src value into r_d
- An update with one of the binary operators $\oplus$ where & and | is binary con- and disjunction, and « » is logical left and right shifts.

With this we want to do some positive testing by constructing valid programs and then check the proofs. We consider only valid programs as CVC5 does not generate proofs for satisfiable terms but rather satisfying models.

We use QuickCheck to generate arbitrary instructions with the property that register r_0 should be greater than 0 and smaller than some arbitrary value, 8192. This simulates a sequence of instructions followed by a memory access. This is a situation the eBPF verifier has been shown to be faulty at previously[15]. We ensure this property by evaluating the program by interpretation written in Haskell. If the program satisfy the property we add a prolog and epilog to the program to make a program that can be validated by the verifier. Specifically, we initialize the registers we use and iclude an exit command. We create the verification condition, with the post condition that $0 \leq r_0 < n$. We convert the representation of the verification condition into SMT2 and discharg it to CVC5. If the negation of the verification condition is unsatisfiable by CVC5, then we can discharge an LFSC proof. We compare the implementation represented in this report, from now on called *lfscr* with the proof checker in C++ provided by CVC5 called *lfscc*. We use this both for finding any bugs but also to benchmark the performance of *lfscr* with a high performance tool.

We can also check the runtime speed of loading an eBPF program into the kernel for comparison. Although this comparison is not very precise, because loading of an eBPF program does more than just verifying it, it still gives a good indication if proof checking is far more intensive than static analysis.

In creating this experiment some credit should be attributed to Ken Friis Larsen, Mads Obitsøe and Matilde Broløs. To discharge eBPF, we use Larsens `https://github.com/kfl/ebpf-tools` and to interact with eBPF we use a collection of FFI-bindings in Haskell by Obitsøe. The generation of code and evaluation is part of ongoing research by all three and I. Lastly convertion of the verification condition to SMT2 takes heavy inspiration from Obitsøes Thesis. The VC generator is made specifically for this project.

For the benchmarking, we create programs of size 1, 10, 100 and 1000. One small caviat that needs mention is that sometimes CVC5 will give a satisfiable result instead an unsatisfiable result. This is a problem somewhere in the pipeline but I have not had time to investigate this matter. For now we settle on generating a new program of same size and try again. In the process it showed to be unfeasible to prove the validity of programs of size 1000 in CVC5 on my i7-1165G7 CPU with 16 GB of ram, thus in the evaluation we only consider programs up to 100.

6.0.1 LFSC - without side conditions?

One interesting finding about the experiments, is that they do not include any side conditions. Therefore it may be interesting to see if a VC generator can be encoded in such a way that it produces terms with side conditions instead of purely logical connectives. If this can efficiently be done, then it may reduce both the size of the proof as well as making the typechecking faster.

7 Evaluation

In the following section we evaluate the implementation, *lfscr* by runtime speed and memory usage. We further argue for the completeness and correctness aswell as general comparison compared to *lfscc*.

7.1 Speed

In the first iteration of the code, the performance of *lfscr* presented in this report were atrocious. From Table 1 it should be clear that, *lfscr* were extremely slow. For a single instruction, the performance are similar to *lfscc*. In this case we essentially just check all the signatures distributed by cvc5[10] along with a small proof. Already for 10 instructions my implementation was using 4 times as long to check a proof, and for a 100 instruction straight line program this difference was close to 140 times slower.

instructions	1	10	100
lfscc	9.0 ± 3.3 ms	9.0 ± 3.2 ms	59.6 ± 0.7 ms
lfscr	12.7 ± 0.7 ms	36.6 ± 1.3 ms	8.3 ± 0.9 s

Table 1: First implementation lfscr Vs lfscc

Inspecting the proofs, which can be found at the repository, in the `vcgen` folder as, `benchmark_-n.plf`, one can notice that there are more than 1000 local bindings for a 100 line program. This lead be to belive that using cons lists would maybe not be optimal, switching to an approach that uses `Vec` as the underlying datastructure and truncating similar to the approach described in Section 5.1.3 gave a decent improvement in speed, being a couple of seconds faster than the first implementation.

instructions	1	10	100
Cons list	12.7 ± 0.7 ms	36.6 ± 1.3 ms	8.3 ± 0.9 s
Vec	18.2 ± 1.7 ms	51.8 ± 1.9 ms	6.4 ± 0.1 s

This was still early in the development and the current implementation still uses cons lists, as they provided easier implementation of the algorithm. For smaller proofs cons lists are still faster but it might be interesting to reinvestigate if using `Vec` is more efficient, now that the implementation is complete.

7.1.1 Massive speedup

Analysing the code with perf, it got clear that most of the time was used in evaluating applications, namely about 60 percent of the time spend was in `eval` and `do_app`. There is nothing inherently strange about this since proofs are mainly just applications and application chains get big for larger proofs. From analyzing the *lfscc* implementation it got clear that my implementation did unecessary computations. Considering the example from 4.6, `and_elim` is a 4 argument symbol, of which `p` is used to destruct the `holds` of the fourth argument and fill `f1`. In the example `a0 = (holds (and cvc.p (and (not cvc.p) true)))` and while the typechecking that a0 $\Longleftarrow$ holds f1 is necessary, the following call to `eval` to bind `p` in the range of the function is uneccesary since `p` does not occur free in the range. Already for this very small formula the application consists of 6 applications at the top level. This pattern appear often in LFSC proofs. Often Π types will include a parameter that does not occur free in the body, but merely exist to destruct a pattern onto an unfilled hole. So including a calculation of whether a bound variable occurs in the body and then checking the condition before evaluation can save massive amount of computation.

This line from the application case in `infer` (along with the actual function for calculating `free`) is enough to make *lfscr* 43 times faster and relatively compareable to *lfscc*.

```
let x = if *free { self.eval(n)? } else { a.clone() };
```

Specifically we get:

instructions	1	10	100
lfscc	8.4 ± 3.2 ms	10.7 ± 1.7 ms	59.2 ± 2.9 ms
lfscr	5.4 ± 1.9 ms	11.7 ± 0.6 ms	193.0 ± 4.6 ms

Hence *lfscc* is now merely 3 times faster than *lfscr. lfscc* takes a different approach than *lfscr. /lfscc* does everything all at once, meaning lexing/parsing and inference and evaluation all occurs in the same function in an online approach. This approach seems to reduce a lot of overhead, but function which does all of this also implements tail calls by using goto statements to the top of the function. If tail calls are eliminated, performance are almost identical for the two approaches.

ADDENDUM These benchmarks were done before, i realized that *lfscc* can be build in both a debug and release version. In the release version it is consistently 2-3 times faster than the results presented here. This suggest that a proof checker can indeed be effeciently implemented, but the approach done in this project is not ideal.

7.1.2 formal checking vs static analysis.

TODO

7.2 Memory

We should consider the memory usage of the implementation in two manners.

First, the size of proofs plays a key role in the feasibility of using proof carrying code. A proof for a single instruction program (actually 4 with pre initialization and the epilog), is 2.7KB in size, while 10 instructions are 8.6KB and 100 instructions gives 109KB. So the proofs, atleast for straight-line programs, scales linearly (or close) with roughly 1KB per instruction. Encoding the proofs in a more compact binary format could make these sizes even smaller. The sizes in themselves are not alarming and could still see use in devices with limited memory.

Secondly we should also look at how much memory the typechecker uses. Running both *lfscr* with the 1,10 and 100 line proofs, we get the following memory usage:

Program size	1	10	100
peak memory	1.3MB	1.8MB	5.7MB
peak RSS	9MB	15.7MB	25.3MB
temporary allocations:	50.13 %	46 %	40 %

From these results we see that that *lfscr* does not use a massive amount of memory. At a single point in time we allocate 5.7MB for a 100 line program and for the entire of a program use 25.3MB.[7] What is most interesting is that 40 % of allocations are temporary and for smaller programs even higher. This suggests that we do some uneccesary computations and that we maybe should use antoher approach than reference counted pointers. This especially become noticable, when similar diagnostics is done for *lfscc* For the 100 line program only 2,9MB memory is used at its peak, while it uses 10MB overall and only 6% of allocations are temporary. One thing to keep in mind is that about 1/4 of allocations are leaked. This is not ideal, but for very shortlived programs such as *lfscc* it is not a big deal. On the other hand for a program that runs in the kernel memory leaks is problematic.

In any case, we can again see that we can check large proofs without many resources needed. But that a "all in one" solution presented by *lfscc* could be worth prototyping in either pure C or in Rust.

7.3 LFSCR - strong suits and weaknesses.

Although *lfscr* is reasonable in both runtime and memory usage, the performance of *lfscc* suggest that a more efficient approach exists. This implementation does have a couple of features that are worth taking into consideration aswell. It is implemented completely in safe Rust, meaning we cannot have any illegal memory that potentially crashes the program. This might be the most desireable property for a program that is designed to run inside the kernel, as "proofs" could exploit such a vulnerability.

Equally an implementation should be robust in the amount of time it takes to check the proof. We showed before the performance difference in checking if the occurence of a variable was free could improve the performance from by 43 times. This immediately shows that we should also consider some sort of time limit for how long a proof must be, since a malicious "proof" could slow down a system massively.

[7]Note that this memory also include some heaptrack overhead.

lfscr has an additional advantage over *lfscc* when considering the positioon in a PCC architecture. Checking the proof has not been tampered with is straight forward and already implemented unintentionally. In its current state, the LFSC proofs discharged from CVC5 always contains the following pattern:

```
... POTENTIAL BINDINGS ...
(# a0 (holds x)
(: (holds false)
... ACTUAL PROOF...
```

here `x` is the formula unsatisfied by CVC5. Given that an in kernel VC generator construct its verification condition as a `AlphaTerm`, then the check is nothing more than normalizing the verification condition and the `a0` of the proof and check for equality.

The experiment has not only provided useful insight into the performance of the implementation; it also establishes confidence that the proof checker works as expected and follow the semantics from 4.3. Checking the signatures along with the generated proofs suggests that mostly all parts of the typechecker is correct. All matters of the term language is covered, and most of the side condition language is also checked. At the moment `markvar` and `if_marked` is left incomplete. The main reason for this is that there is currently no signatures distributed by CVC5 that include them. The side condition language could be tested more thoroughly as only a single larger tests has been conducted by the $P \land \neg P$ unsatisfiability proof from 4.6. Despite the example being rather small, it test a large part of the side condition language, both constant and program application, match constructs, branching and numerical functions. One point where the implementation is inherently wrong is the usage of i32's for the representation of integers and rationals, in fact these should be unbounded integers. This is not a problem for bitvector proofs, but only for arithmetic logics. I have however left the representation as is for now, as I have not been able to find a library that efficiently implements ubounded integers and rationals and are compatible with the kernel requirements.

Albeit the implementation does not run in the kernel, the implementation only uses the `core` and `alloc` crate along with `nom`, which I have been succesful in compiling and simple example of in a kernel module. Hence there is nothing theoretical stopping us from compiling *lfscr* into the kernel. The major work that should be done here is to make every allocation fallible by using the `try_new` counter parts to `new` allocations, and implementation a simple `From` trait to easily converting allocation errors into typechecking errors. Hereby the `?` shortcut can be used, and no types should changes as they already implement `Result` types.

8 Is PCC a good idea?

Even with a Rust implementation that promises memory safety and has no unexpected errors that can crash the program, the answer is not definite at this time. It might still not be feasible to use LFSC for an in kernel proof checker as part of a larger proof carrying code architecture, since a lot of questions are still unanswered. The eBPF verifier does a lot more than just validating instructions of a bytecode format. It check validity in memory alignment, user-rights, does program rewrites and much more. Some of these can definitely be encoded into a proof, but others may be harder to realize. Especially user-rights can prove as a challenge, since it either requires the code producer to make the proof themselves, meaning eBPF programs are not that easily distributed over machines, or they have to be patched in some way. Another possibility is only checking capabilities as a seperate stage before checking the proof, but this may reduce some functionality some "features" of eBPF in its current form for some users. Thus there is still a lot more work to be done in the architectural construction of a PCC system.

Another pressing matter is that of the execution time. We have seen that proof-checking of validity of eBPF programs (atleast straight line programs) can be efficiently done, but with this implementation we are not quite there yet. The benchmarking showed promises in a few different places. Using `Vec` instead of cons lists may be useful for larger proofs, and it would further be interesting to investigate if a modified version of the data layout would prove useful. For instance, we might be able to used tagged pointers or atleast a more compact data format for `Value`'s and `Neutral`'s to make the program both more memory and runtime efficient. Furthermore, the benchmarking we have done may not be entirely appropriate for determining the feasibility as we have only included straight line programs and no control flow constructs. In the end this can make proofs more complicated.

Despite all this the implementation we present is rather small and consist of only 2400 lines of code compared to 19000 in the verifier. Bugs are hence less likely to appear. In any case, we do not completely discard the idea of PCC in the kernel as it does show promises and with time could be a generally decent replacement for the eBPF verifier.

9 Conclusion

We have in this report given a brief overview of how the eBPF verifier works, and we have discussed some concerns with the current eBPF verifier. This motivates the intend of making a proof carrying code architecture as part of the Linux kernel. We have described how we can use the reasonably new feature of Rust inside the kernel, to make such an architecture. We have described the dependently typed language LFSC, which uses the curry howard isomorphism to construct logics, and although not complete, an implementation of a typechecker for the language. The typechecker in its current form, is not compileable inside of the kernel but uses only features of the Rust language, with some slight modifications, such as having to enable the Reference counted smart pointer. The implementation of the typechecker is reasonably simple in structure, with only 2400 lines of code and uses purely safe features in rust and thus provide the necessary safety to run inside the kernel. We have further done some experiments that generates arbitrary straight line eBPF programs and performs a naive verification condition on them to generate LFSC proofs with CVC5. We used these experiments to ensure the correctness of the implementation aswell as providing a framework for benchmarking the performance. In this process we have found the implementation described in this report to be inferior in performance to the *lfscc* proof checker from CVC5 by up to 10 times slower execution time, which suggest a modular approach using normalization by evaluation is not the ideal approach. Although the execution time is higher for this implementation it is still within reason and may, with some optimizations, be decent enough for an in kernel verifier. We have further by accident also facilitated the foundation for ensuring proofs correspond to the program it is supplied with in a PCC setting. The argumentation here is that the proof encodes the verification condition as an LFSC term, and if the in-kernel proof checker generates verification conditions in LFSC then checking the requirement is merely normalization followed by a equality check.

Thus, while we cannot say anything definite at this point, we have provided a basis for future work to make a PCC architecture in the Linux kernel.

9.1 Future work

We have presented only a small part of a proof carrying code architecture, so naturally for a final conclusion on the matter, discussed in this work, an in-kernel verification condition generator should be defined. We suggest such a proof checker represents the logic in the form of LFSC. This is however dependent on getting a faster proof checker and to solve the problem of potential clogging.

The implementation discussed, should be modified to run inside the kernel. Because of the preliminary work, this should be a minor task.

More work should be done in making the implementation fast.

Lastly, a more comprehensive analysis and design between PCC and the eBPF verifier should be conducted. Specifically, what parts of the verifier should co-exist with PCC. One such is example is checking of user-rights.

References

[1] *Adding syscalls*. URL: `https://www.kernel.org/doc/html/v4.10/process/adding-syscalls.html` (visited on 05/29/2023).

[2] U. Berger and H. Schwichtenberg. "An inverse of the evaluation functional for typed lambda -calculus". In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. 1991, pp. 203–211. DOI: `10.1109/LICS.1991.151645`.

[3] *BPF Compiler Collection (BCC)*. URL: `https://github.com/iovisor/bcc` (visited on 05/26/2023).

[4] *Carcara*. URL: `https://github.com/ufmg-smite/carcara` (visited on 05/29/2023).

[5] David Thrane Christiansen. *Checking Dependent Types with Normalization by Evaluation: A Tutorial (Haskell Version)*. 2019.

[6] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (August 1975), pp. 453–457. ISSN: 0001-0782. DOI: `10.1145/360933.360975`. URL: `https://doi.org/10.1145/360933.360975`.

[7] *eBPF Helper calls*. URL: `https://ebpf.io/what-is-ebpf/#helper-calls` (visited on 05/26/2023).

[8] *eBPF verifier*. URL: `https://docs.kernel.org/bpf/verifier.html` (visited on 05/29/2023).

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. "A Framework for Defining Logics". In: *J. ACM* 40.1 (January 1993), pp. 143–184. ISSN: 0004-5411. DOI: `10.1145/138027.138060`. URL: `https://doi.org/10.1145/138027.138060`.

[10] *LFSC signatures*. URL: `https://github.com/cvc5/cvc5/tree/main/proofs/lfsc/signatures` (visited on 05/28/2023).

[11] *LFSC: A High-Performance LFSC Proof Checker*. URL: https://github.com/cvc5/LFSC (visited on 05/29/2023).

[12] *libbpf*. URL: https://github.com/libbpf/libbpf (visited on 05/26/2023).

[13] *Loading process of ePBF programs*. URL: https://ebpf.io/static/7eec5ccd8f6fbaf055256da491b5f15/loader.png (visited on 05/26/2023).

[14] George C. Necula. "Proof-Carrying Code". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 106–119. ISBN: 0897918533. DOI: 10.1145/263699.263712. URL: https://doi.org/10.1145/263699.263712.

[15] Manfred Paul. *CVE-2020-8835: Linux Kernel Privilege Escalation via Improper eBPF Program Verification*. 2020. URL: https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification (visited on 05/25/2023).

[16] Simon Scanell. *Fuzzing for eBPF JIT bugs in the Linux kernel*. 2021. URL: https://scannell.io/posts/ebpf-fuzzing/ (visited on 05/23/2023).

[17] Aaron Stump et al. "SMT Proof Checking Using a Logical Framework". In: *Formal Methods in System Design* 42 (February 2013). DOI: 10.1007/s10703-012-0163-3.