Static Worst-Case Resource Analysis for Substrate Pallets

Dependable Systems Laboratory

ChainSecurity

Master Thesis, Spring 2022

Student :
Simon Perriard

Supervised by :
Dr. Dimitar Dimitrov (ChainSecurity)

Directed by :
Prof. George Candea (DSLAB, EPFL)

August, 2022

Acknowledgment

First, I would like to express my most sincere thanks to Dr. Dimitar Dimitrov for supervising this thesis. His support and expertise greatly helped me during these 6 months. It was a pleasure learning and working with him.

I had the chance to do this thesis at ChainSecurity, surrounded by experienced analysis tools developers and the very responsive Bjorn3[1], from the Rust compiler team, who answered my questions quickly and precisely. Thanks a lot to all of you and to ChainSecurity for your warm welcome and support.

Lastly, I would like to thank my friends and family for their support during the thesis and along my studies.

Simon Perriard, 12 August 2022

[1] `https://users.rust-lang.org/u/bjorn3/summary`

Abstract

We present `saft`, the first attempt of a static analyzer that extracts the asymptotic function complexity for the Polkadot/Substrate ecosystem, where the burden of accounting for computation resource consumption is put on the developer. `saft` is a tool meant to be used in a complementary way with pre-existing development tools to improve the Substrate-based blockchain security.

`saft` combines two techniques to analyze the MIR generated from the compilation of the blockchain subsystems, the *pallets*: (i) abstract interpretation for the over-approximation of the concrete semantics and extraction of complexity asymptotics and (ii) symbolic execution to track the length of dynamically sized vectors, increasing the precision of (i). The developers can then compare `saft` output with their resource consumption computation.

Our experimental evaluation showed that the tool scales well with arbitrary pallets, given some manual specifications.

Contents

1 Introduction

Major problems for permissionless blockchains are Denial-of-Service (DoS) attacks that consume the computational resources available to the distributed system. Because a permissionless blockchain allows anyone to submit transactions, a small group of users could execute transactions whose computational cost is an overwhelming fraction of the resources available. To prevent this, permissionless blockchains usually charge a fee proportional to the resources that a transaction consumes. This leads to the technical problem of how to determine the resource usage of transactions, so that appropriate fees can be charged. Failing to do so poses a significant security threat to the blockchain.

A typical design is based on runtime metering. The execution engine monitors some metrics, e.g. the execution time so far. If there is no sufficient fee to pay for further time, then the execution stops. Otherwise, the execution eventually terminates, in which case the remaining fee is refunded. While the approach is relatively straightforward it has two drawbacks. First, the extra instrumentation needed for metering could be costly on conventional hardware. Second, neither users nor system nodes know the precise fee in advance.

In an alternative approach, taken by the Substrate framework[2], fees are pre-computed. Every transaction in a Substrate-based blockchain must come with a function that, given the transaction's arguments, *weighs* the execution time needed. The transaction fee then includes a component based on those computed weights. For this to work out, three conditions must be met. First, computing the weights must be *cheaper* than the runtime monitoring it replaces. Second, the weights must be *sound*, that is, they must overestimate the actual cost. Underestimation would enable malicious users to trigger unaccounted worst-case executions leading to denial-of-service attacks. Third, the weights must be relatively *precise*, that is, they should not overestimate too much. Overestimation can make the fees too expensive, discouraging users from using the blockchain.

In this project, we provide a semi-formal approach to the worst-case resource analysis problem, that is, a combination of automatic and manual analysis. We tried to automate the process as much as possible but we found limits on what is possible to be done considering the implementation choices we made.

1.1 Goal and Challenges

Goal of the thesis. Substrate-based blockchain developers bear the burden of providing a suitable weight function. Coming up with one leads to at least two considerations. First, one needs to determine a symbolic expression for the execution time of the transaction in terms of possibly unknown constants and functions of the input. Second, one must fit the expression to actual running times. The first point requires careful analysis of the source code and the algorithms that are employed. The second point requires crafting benchmarks that test the source code in different scenarios, worst-case execution times in particular.

We aim to design and implement a static analysis that derives symbolic expressions that estimate the wort-case execution time of transactions in Substrate pallets (the components of a blockchain based on the Substrate framework). Since the actual execution time depends on a particular hardware and software configuration, there is a limitation of how far a purely static analysis approach can go. But what the approach can ensure is soundness with respect to a reasonable abstract cost model (e.g., source-level instruction count). Fitting the inferred symbolic expression to actual execution times still needs to be based on concrete benchmarks. Therefore, the main use of such an analysis is to assure developers that a transaction has a certain asymptotic behavior.

The main objective of this project is to increase the security of Substrate-based blockchains by developing a static analysis tool that enables developers to check whether their system allows heavy execution time for a cheap cost, or in other words whether the system is vulnerable to a cheap DoS.

Challenges. As well known, static analysis is generally unfeasible. Inferring execution time bounds implies termination, i.e., the problem is at least as difficult as proving termination. The best we can do is design algorithms that target specific classes of programs, and have a significant precision penalty outside of that class. There have been many works that approach the problem that may serve as a baseline for the analysis. Our starting point is an *abstract interpretation*-based approach (cf., [3, 6]) that we selected after a careful consideration of existing Substrate pallets. The selected baseline was implemented and tested on a representative set of pallets. The next stage was to identify the shortcomings and improve the baseline.

[2] `https://substrate.io/`

A major challenge was to decide on which of the program representations the analysis will operate on: whether that would be the source code (Rust), in which pallets are written, or the binary code (WebAssembly) that a pallet compiles down to. Source code has the advantage of being at a higher level of abstraction, allowing the source program to be simpler and easier to reason about. The drawback is that the source code is further away from the actual code executed. Program transformations down the line might lead to unpredictable changes in the execution time, compared to an abstract model on the source code. That is why the produced binary contains more information about execution time. However, WebAssembly is typically an intermediate representation itself, that is further just-in-time compiled, and therefore the unpredictability due to program transformations still remains. Moreover, binary code is more difficult to analyze, which leads to imprecision. Ideally, a source-level analysis (Rust) should suffice, with any mismatch with the binary code incorporated into the cost model. But this is not guaranteed, and that is why the Rust and WebAssembly code needed to be compared.

Other major challenges have been what subset of the input language to support, and what to do with the inevitable imprecision loss. For this project we do not plan to support some language features such as dynamic dispatch to unknown components (e.g., other pallets unknown at compile time). There will be cases where the analysis will be too over-approximating. In these cases, user-provided invariants could help derive precise execution time bounds. To know how useful such a feature is would depend on the abstract domains used in the analysis and could possibly require further research going outside of this thesis' scope.

Due to the complex nature of the Rust programming language and its multistage compilation process, it is important to understand the details of the language and the global compiler structure in order to be able to efficiently work with the provided API. The Rust compiler is a huge project that would take years to fully master so it is critical to be aware of what kind of relevant data structures are used internally and how to work with them, as well as be able to quickly target the relevant API functions and know what kind of information is available from the compilation process.

The Substrate framework is also a big project of its own, initiated by Gavin Wood, on of the co-founders of Ethereum. It has an intricate architecture that tries to ease the developing process for blockchain developers by leveraging the powerful Rust's `macro` system[3], which allows the parsing of a token stream and generating code from it at compile time, obfuscating a lot of code that could be useful to someone trying to understand the general picture of the framework. As often, simplicity for the user comes at the cost of a more complex underlying system and Substrate is not an exception.

Writing a useful static analysis is hard and a deep understanding of both the compiler and Substrate are a mandatory step to later be able to come up with a meaningful analysis.

After discussing the project with a team from Web3 Foundation[4], we decided that the proof-of-concept tool should aim to support the following five pallets: `balances`, `identity`, `multisig`, `utility`, and `vesting`. Those pallets are interesting in their diversity and are likely to be ones of the most used modules in the development of Substrate-based blockchains.

1.2 Related Work

Early notable work around static performance analysis has been in the wild since 1975 with the work of Ben Wegbreit [1]. He describes a static analyzer for Lisp, called *Metric*, that is able to output closed-form expressions describing the program's execution behavior. Following the work of Patrick and Radhia Cousot [2] on *abstract interpretation* in 1977, Mads Rosendahl used this kind of analysis in the 80's to develop a method that derives time upper bounds for a subset of Lisp programs as a function of the input size [3].

Though abstract interpretation is a powerful tool for static analysis of bounds, other paths have been explored. One of them is type-based analyses for inferring upper bound evaluation (worst-case) costs. [5] did it by extending the type system to sized types that account for the sizes of the function's arguments, as well as the computational overhead of calling the function. In addition to that, they also extend the type inference algorithm to gather recurrence relations in the form of constraints that they can solve later. The combination of these two methods gives an inference algorithm for the costs and sizes, that ultimately yields an upper bound on the execution cost.

[3] `https://doc.rust-lang.org/book/ch19-06-macros.html`
[4] `https://web3.foundation/`

Another very interesting static analysis method in the one proposed by Zachary Kincaid et al. (see [7] and [8]). The idea is to use an algebraic abstract interpretation framework to compute regular expression that describe the paths going from the input to a particular node in the program and then use a suitable *semantic algebra* to extract and solve recurrence relations from the program's control-flow graph.

Apart from worst-case complexity analysis, extensive research is also done on other kinds of resource analysis, like amortized analysis (see [14]). This is a more precise type of analysis since it considers the average cost over a sequence of operations instead of accounting for each statement separately.

Static analysis has been used for several decades to ensure the properties of critical systems as programs gain in complexity and the task of full execution paths testing becomes impossible. One of the most successful industry examples is AbsInt[5], who developed a product called aiT WCET [4] that relies on abstract interpretation and formal hardware modeling to infer precise execution time upper bounds for real-time embedded systems.

As permissionless blockchain technologies evolved from Bitcoin [21] to enable arbitrary program execution, like Ethereum [29], some systems that are built in this environment are highly security-critical as they could hold tokens (cryptocurrencies) valued at several billions of dollars. Such programs, called smart contracts in the Ethereum ecosystem, may have some tendencies to get hacked when they hold big amounts. An (in)famous and early hack is The DAO[6], where the attacker used an attack vector called re-entrancy, a bug that could have been uncovered by an effective static analysis. Since then, a lot of effort has been put into making smart contracts safer. The most known tools are `slither` [10] and `manticore` [11], the first one being a static analyzer for Solidity[7], the main language used for smart contracts development on Ethereum, that runs a series of vulnerability detectors running on a custom Solidity internal intermediate representation and the second one is a symbolic execution tool running on the Ethereum virtual machine's (EVM) bytecode, Linux ELF binaries and Wasm modules. Other interesting contributions to the Ethereum on-chain security are VerX [13], a verifier that can infer and check functional properties of a bunch of smart contracts, and Securify/Securify2 [9], one of the first Ethereum smart contracts static analyzer, developed by the SRI[8] laboratory at ETHZ and ChainSecurity[9].

While Ethereum-based blockchains are currently the most widely used, some effort has been put on other systems, like the Tezos platform and its Michelson[10] language. In this context, V. Pérez et al. [12] propose a *parametric resource analysis* for Michelson smart contracts, capable of deriving bounds on the execution cost, based on some user-defined parameters like the function's arguments and the storage it has access to.

Some work has already been done around static analysis for Rust and the language's Mid-level Intermediate Representation, the currently most used tool being `clippy` [22], a linter with more than 500 lints. It is also worth mentioning two interpreters for the MIR, `miri` [24] and `mirai` [19]. The first one is maintained by the Rust team itself and is designed to detect certain undefined behaviors and raise warnings. The latter is developed by Meta Experimental, the research branch of Meta, and is born from the will to increase the support for static analysis in Rust as they were developing their own blockchain Diem (previously Libra). The tool supports annotations that enable the users to verify some correctness properties of their code.

During the tool's development, we stumbled upon a recent master's thesis [16] that proposes a taint analysis for Rust based on the MIR. Their code base[11] was really helpful in understanding how to use the compiler's dataflow framework, described in Section 6.4.3.

At the time of writing, `saft` is the first known attempt to develop a static analysis tool for the Polkadot/Substrate ecosystem.

[5] `https://www.absint.com`
[6] `https://ogucluturk.medium.com/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562`
[7] `https://docs.soliditylang.org/en/latest/`
[8] `https://www.sri.inf.ethz.ch/`
[9] `https://chainsecurity.com/`
[10] `https://tezos.gitlab.io/active/michelson.html`
[11] `https://github.com/LiHRaM/taint`

2 Problem statement

The goal of the tool is to provide readable and reasonably precise upper bounds on the worst-case consumption of various resources during execution. We shall provide an example with the `identity::add_registrar` function of Substrate's Identity pallet, displayed in Listing 1. The function is rather simple: it reads from storage a vector (`Registrars`) with statically fixed maximum size, then it tries to push one element, and finally it emits an event if the push is successful.

Listing 1: The `identity::add_registrar` dispatchable function

```
/// # <weight>
/// - `O(R)` where `R` registrar-count (governance-bounded and code-bounded).
/// - One storage mutation (codec `O(R)`).
/// - One event.
/// # </weight>
#[pallet::weight(T::WeightInfo::add_registrar(T::MaxRegistrars::get()))]
pub fn add_registrar(
  origin: OriginFor<T>,
  account: T::AccountId,
) -> DispatchResultWithPostInfo {
  T::RegistrarOrigin::ensure_origin(origin)?;

  let (i, registrar_count) = <Registrars<T>>::try_mutate(
    |registrars| -> Result<(RegistrarIndex, usize), DispatchError> {
      registrars
        .try_push(Some(RegistrarInfo {
          account,
          fee: Zero::zero(),
          fields: Default::default(),
        }))
        .map_err(|_| Error::<T>::TooManyRegistrars)?;
      Ok(((registrars.len() - 1) as RegistrarIndex, registrars.len()))
    },
  )?;

  Self::deposit_event(Event::RegistrarAdded { registrar_index: i });

  Ok(Some(T::WeightInfo::add_registrar(registrar_count as u32)).into())
}
```

Our analysis infers the following bounds on the worst-case behavior of `add_registrar`:

- bytes read from storage due to `try_mutate`:

  (size of one `Registrars` element) × (maximum length of `Registrars`)

- bytes written to storage due to `try_mutate`:

  (size of one `Registrars` element) × (maximum length of `Registrars`)

- bytes deposited by events:

  (size of the `RegistrarAdded` event)

- computational steps (in big $\mathcal{O}$ notation):

  $\mathcal{O}$((maximum length of `Registrars`))

For comparison, the weight function derived by using the `benchmark` module is

$$16'649'000 + 241'000 \times r + \texttt{T::DbWeight::get().reads(1)} + \texttt{T::DbWeight::get().writes(1)}$$

where $r = \texttt{T::MaxRegistrars::get()}$

We find costs for two database accesses (a read and a write), a multiple of the the maximum vector length (r), and some constant cost.

As a reminder, the *weight function* specified by the pallet developers determines how much one needs to pay up-front for the execution of the dispatchable. The dispatchable itself can then refund some part depending on the resources actually consumed. In this example, the actual execution cost depends on the current number of registrars in storage. Weight functions should not depend on storage, hence they need to over-approximate. We only target expressions that help with weight functions, and we do not consider refunds.

2.1 The technical problem

The technical problem that the tool solves is to soundly approximate the worst-case resource semantics of the given program. We now describe what this means in more detail.

The starting point is the *operational semantics*, which describes the sequence of states that a program goes through from activation until termination. This sequence of states, called a *trace*, explains what the program *does* but it does not explain what it *costs*, since it does not come with a fixed notion of what a resource is. This is done by a *resource semantics* (a.k.a. a *cost model*) (see, e.g., [3]). A resource semantics depends on the resource of interest. It can be as simple as the number of steps executed in the trace, or something slightly more complicated, such as the peak size of a state, as determined by an appropriate size metric, e.g., the length of a bit encoding of the state. Let us illustrate both semantics on the implementation of a binary search in Listing 2.

Listing 2: Binary search over an array. The elements of the array are assumed to be ordered.

```
fn binary_search(key: i32, items: &[i32]) -> Option<usize> {
    let mut l = 0;
    let mut h = items.len();
    while l < h {
        let m = l + (h - l) / 2;
        match key.cmp(items[m]) {
            Equal => return Some(m),
            Less => h = m - 1,
            Greater => l = m + 1,
        }
    }
    None
}
```

In this case, a state consists of a value for each of the variables occurring in the procedure: the key, the array of items, and the low, high, and middle indices. In order to define the sequence of states the program transitions through, the operational semantics specifies what the primitive execution instructions are (triggering state transitions), and how control flows from one instruction to another, that is, how instructions are scheduled. For example, a primitive instruction in the binary search are all the arithmetical operations, and procedure call activations and returns, e.g., to and from `cmp` and `key`. The schedule depends on how the instructions are composed together: the semicolon operator schedules one fragment to execute after another; the `while` operator first schedules the test for execution, and then, if the test succeeds, it schedules the body, and then it repeats; compound arithmetical expressions are broken down into primitives and scheduled according to data dependencies. A simplified trace of the binary search is shown in Figure 1.

The figure also shows a simple resource semantics: the accumulated number of steps until the completion of the current primitive, counted from the beginning of the call to `binary_search`.[12] Here we assume that primitives can cost more than a single step to execute. For example, to call the binary search procedure costs $1 + 1 + 1 = 3$ steps: one for the call itself, and one per each of its constant size arguments (the cost of initialization). Similarly, a return takes additional steps if a return value is present, in order to make it accessible to the caller (e.g., by copying).

The form of resource semantics shown in the figure serves as the ground truth for resource consumption, but it is not very useful in practice. The problem is that it captures how consumption

[12] Of course, such a resource is only a proxy for the real execution time on a specific processor. The reason is that real execution time depends on many external factors beyond the operational semantics: the actual machine code executed, the time sharing with other active processes, the actual real time guarantees of the processor.

Primitive instruction	Result	State	Step
`call binary_search(0, [0])`		$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: \ , \texttt{h}: \ , \texttt{m}: \ \}$	3
`l = 0`		$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: \ , \texttt{m}: \ \}$	4
`call items.len()`		$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: \ , \texttt{m}: \ \}$	5
`return`	1	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: \ , \texttt{m}: \ \}$	8
`h = 1`		$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: \ \}$	9
`l < h`	`true`	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: \ \}$	10
`h - l`	1	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: \ \}$	11
1/2	0	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: \ \}$	12
`l + 0`	0	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: \ \}$	13
`m = 0`	0	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: 0\}$	14
`items[m]`	0	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: 0\}$	15
`call key.cmp(0)`		$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: 0\}$	17
`return`	`Equal`	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: 0\}$	19
`return`	`Some(0)`	$\{\texttt{key}: 0, \texttt{items}: [0], \texttt{l}: 0, \texttt{h}: 1, \texttt{m}: 0\}$	21

Figure 1: An execution trace of the binary search procedure together with accumulated step consumption. The steps of the nested function calls to `items.len` and `key.cmp` have been omitted. It has been assumed that the bodies of these two functions take one step to execute.

depends on *individual* inputs, which is too fine-grained. We can express this schematically as

$$\text{input} \xrightarrow{\textit{operational semantics}} \text{trace} \xrightarrow{\text{resource semantics}} \text{resource-annotated trace.}$$

Or if we abstract away intermediate computations and focus on the final state of a trace, that is, if we focus on the output and the net resource cost, then the semantics are captured by the mappings:

$$\mathbf{OpSem}_{\mathsf{IO}} : \langle \text{program} \rangle \to \mathsf{Input} \to \mathsf{Output}$$
$$\mathbf{ResSem}_{\mathsf{IO}} : \langle \text{program} \rangle \to \mathsf{Input} \to \mathsf{Cost}.$$

Instead of such an explicit dependence on the input, we are interested in how resource consumption varies in relation to some *metric* $\|x\|$ of the input x, such as its size. For that, we need an *aggregate* resource semantics that specifies a cost aggregated across a class of inputs: worst-case, expected, amortized, etc., costs. In this thesis we focus exclusively on the worst-case cost. For the net cost at termination the worst-case semantics is the mapping:

$$\mathbf{WorstCaseResSem}_{\mathsf{IO}} : \langle \text{program} \rangle \to \mathsf{Metric} \to \mathsf{Cost}$$
$$\mathbf{WorstCaseResSem}_{\mathsf{IO}}(\mathbf{prog}) = n \mapsto \max\{\mathbf{ResSem}_{\mathsf{IO}}(\mathbf{prog})(x) \mid x \in \mathsf{Input} \wedge \|x\| \leq n\}.$$

The static analysis provides an *abstract semantics*, whose goal is to *over-approximate* with a reasonable precision the worst-case resource semantics. To a first approximation, for the net cost at termination that would be a mapping

$$\mathbf{WorstCaseResSem}^{\#}_{\mathsf{IO}} : \langle \text{program} \rangle \to \mathsf{Metric} \to \mathsf{Cost}$$

such that for all programs **prog** and all upper bounds on the input metric $m \in \mathsf{Metric}$

$$\mathbf{WorstCaseResSem}_{\mathsf{IO}}(\mathbf{prog})(m) \leq \mathbf{WorstCaseResSem}^{\#}_{\mathsf{IO}}(\mathbf{prog})(m).$$

Over-approximation, however, is not the only goal. The issue is that the usefulness of the solution depends also on the form in which it is expressed, i.e., how $\mathbf{WorstCaseResSem}^{\#}_{\mathsf{IO}}(\mathbf{prog})$ is coded. Without extra constraints, the problem has a trivial and useless solution. More specifically, if the analyzed program always terminates and only finitely many inputs are bounded by a finite value of the metric, then we can simply repeat the definition of worst-case cost. For example, we can express the worst-case number of steps that the binary search needs as a function on the number of items in its input with the program in Listing 3, which is too complicated to be useful.

Listing 3: Encoding of the worst-case cost of the binary search in Listing 2 as a program. The function `enumerate_binary_search_args(n)` enumerates all arguments `key`, `items` with `items.len()` $\leq$ `n` (there are finitely many since `i32` has only 32 bits). The function `res_sem_io_binary_search(key, items)` counts the steps taken by binary search for the given inputs, which can be automatically generated by simulating (instrumenting) the binary search.

```
fn worst_case_res_sem_io_binary_search(usize n) -> usize {
    let usize max = 0;
    for (key, items) in enumerate_binary_search_args(n) {
        max = cmp::max(max, res_sem_io_binary_search(key, items);
    }
    max
}
```

That is why, in addition to soundness, the analysis has to produce simple enough expressions. For example, for the cost of binary search we normally expect an expression in the form

$$A \cdot \log(n) + B$$

where n is the number of items being searched and A and B are constants. In fact, we often want to abstract the constants away too, for example, because they are irrelevant, or they depend on some external unspecified component. That is why, we shall also permit big $\mathcal{O}$ notation, e.g.,

$$\mathcal{O}(\log(n)).$$

2.2 Feasibility and language fragment

The general problem of determining the resource usage of a program is undecidable. This is very simple to establish by reducing the halting problem: the number of steps needed to execute a program is finite iff the program terminates, hence a sound and precise static analysis for finding this number would need to at least solve the halting problem. That is why, as standard, our focus is to develop a sound static analysis that is precise only for a limited fragment of the source programming language. In particular, we focus on Rust programs with *no loops* and *no recursion*, or at best, where invariants for loops and recursive functions are provided manually. Our efforts target features (present but not specific to Rust) such as mutability, higher-order functions and closures, and parametric polymoprhism, which are heavily used by Substrate pallets. We believe this approach is still useful in cases where the pallets are relatively simple at the surface, with complexity encapsulated in heavily reused library functions for which we provide manual specifications.

3 The Substrate environment

This section presents the Substrate framework, the technologies and concepts revolving around it, as well as the ecosystem it is intended to be integrated to: Polkadot.

3.1 Polkadot blockchain ecosystem

We can extract the building blocks of the Polkadot ecosystem as the Relay Chain [26], the parachains, and the parathreads (Fig. 2).

The Relay Chain is the central entity. It implements nominated proof-of-stake (NPoS) and enables only a minimal set of transactions because its primary goal is to ensure the coordination of the whole system. The complete protocol specification can be found in [20].

If we consider the Relay Chain to be a computer, the parachains can be compared to always running applications in physical memory and the parathreads are like applications that are stored on the disk and loaded when needed. Parachains and parathreads are specialized subsystems, they often implement their own blockchain, with their own tokens and consensus algorithm. Paraobjects have no constraint except that they must be able to produce proofs that can be validated by the validators assigned to the parachain/parathread.

From the Relay Chain point of view there are three actors:

- **validators**: validators usually have a lot of DOT at stake to have a chance to be elected in the NPoS algorithm. They produce blocks for the Relay Chain if they are elected in the current active validators set. They also collect proofs of state transition from the collators.
- **nominators**: any DOT owner can be a nominator, they will bind their tokens to a set of validators they trust. They are responsible for slashing the validator if it behaves against the protocol and are well rewarded if the validator behaves correctly.
- **collators**: they connect parachains to the Relay Chain. Collators are full nodes on the Relay Chain and on the parachain, they are responsible for collecting parachain transactions and producing state transition proofs for the validators. Collators are also responsible for cross-chain message passing for interoperability.

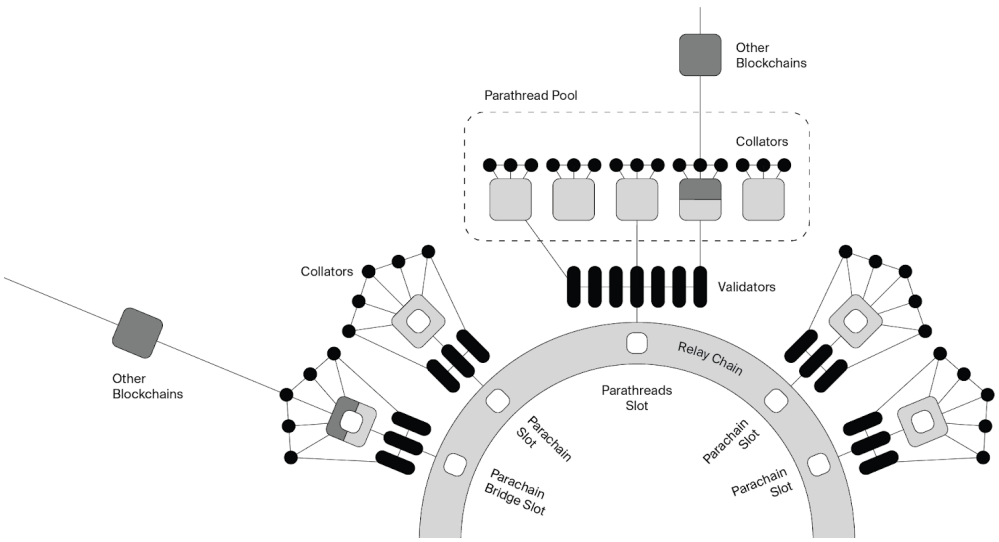

Figure 2: Polkadot architecture

Source: `https://polkadot.network/blog/parathreads-parathreads-pay-as-you-go-parachains/`

3.2 Substrate framework and FRAME library

Substrate [27] is born from the development of the Polkadot Relay Chain, both maintained by Parity Technologies[13] (paritytech), it can be seen as a blockchain SDK. Substrate-based blockchains can run as standalone systems or be included in the Polkadot ecosystem as parachains.

[13] `https://www.parity.io/`

The Substrate framework is a complete modular blockchain builder as it provides all the needed core components such as a database and network layers, a consensus engine, a transaction pool, as well as some modules for runtime development. The runtime is the business logic of the blockchain, a set of rules that define how the state can be modified. The Substrate client coordinates all these components to run a blockchain node, Fig. 3. Developers can choose to implement their blockchain from scratch with Substrate Core and use any language targeting WebAssembly since the client will execute a Wasm blob as its runtime. Going for the Substrate Core way allows the developer a lot of freedom when it comes down to implementation details but it is also quite tedious, especially if the runtime is not compatible with the node's abstract block creation logic. This is where Substrate Framework for Runtime Aggregation of Modularized Entities (FRAME) comes into play, it provides helper modules to interact with Substrate Core and greatly eases the task of building the runtime. FRAME library provides plug-and-play runtime modules, called pallets, and makes some macros available for developers to write custom pallets holding the logic they need for their blockchain. Not only it is simple to integrate new business logic, FRAME also makes it easy to choose between different database implementations (RocksDB, ParityDB), or different block construction algorithms (Aura, BABE [18], proof-of-work).

Note that there are currently two runtimes, on-chain Wasm runtime, and native runtime. The on-chain runtime is the "true" one since it is the one stored on the validated blockchain. The native runtime, usually compiled in `x86_64`, is an optimization coming from the fact that Wasm code execution is slower than native code execution, the Wasmi [28] execution (interpreter) is 10x slower and the Wasmtime [17] execution (compiler) is 2x slower. It turned out that this optimization did not deliver its promises for different reasons[14] and at the time of writing, there is the will at paritytech to get rid of the native runtime [15].

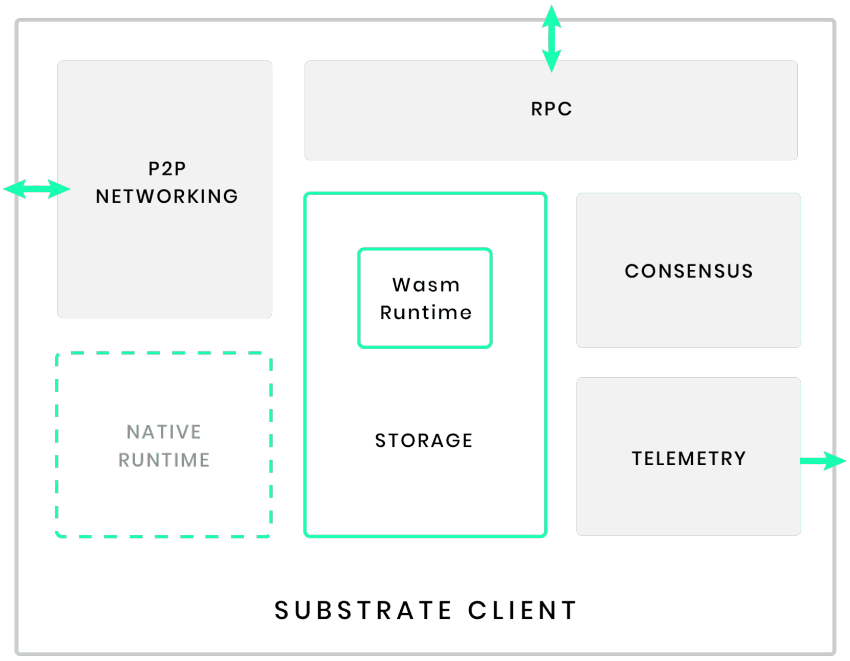

Figure 3: Substrate client architecture

Source: `https://docs.substrate.io/v3/getting-started/architecture/`

3.2.1 Substrate concepts and technicalities

Since Substrate comes with its own concepts and technical details, it is useful to define some of them here:

[14] `https://github.com/paritytech/substrate/issues/7288`
[15] `https://github.com/paritytech/substrate/issues/10579`

- **proposer**: a node authoring a new block, the node will propose this new block to the network.
- **extrinsics**: piece of data coming from outside the chain and that will be included in a block. Extrinsic can either be signed/unsigned transactions or inherents.
- **inherents**: special kind of extrinsics, piece of external data added by the block author (proposer). A trivial example is block timestamp.
- **structure of a block**: as in other blockchains the block is composed of a header and the list of what happened during the block execution, i.e. the list of transactions for Ethereum-like chains, the list of extrinsics for Substrate chains.
- **dispatchables**: functions implemented by pallets that can be triggered by the execution of extrinsics.

3.2.2 Building blocks of a pallet

Pallet development heavily relies on macros made available by the FRAME library [16]. Based on the code written by the developer, they will generate the necessary code that allows the pallet to be integrated in the blockchain's Runtime. The mandatory attribute macros are:

- `#[frame_support::pallet]` on the pallet module, allows the module to be used as a pallet by the Runtime
- `#[pallet::config]` on the `Config` trait of the pallet, here the developer needs to define generic types that are used in the pallet, like the `Currency` of the chain for example. Those types will be made concrete upon Runtime construction
- `#[pallet::pallet]` on the `Pallet<T>(_)` struct of the pallet, the macro will implement all the pallet information on top of this struct.

Other interesting and often used attribute macros are:

- `#[pallet::call]` on the `impl` block of the `Pallet<T>` struct, this indicate that all the functions implemented in the block are dispatchables, i.e. they are entry points in the pallet's logic. Since nothing comes for free, every call to a dispatchable must have a cost that users will have to pay in order to get their transactions executed. The cost of each dispatchable has to be known in advance and this is made possible by the use of so-called weights and the `#[pallet::weight(...)]` macro. `#[pallet::call]` macro will also generate the pallet-level `enum Call`, whose variants are the dispatchable functions of the pallet, and other queryable information concerning the dispatchables.
- `#[pallet::storage]` on runtime's storage abstractions like `StorageValue`[17] or `StorageMap`[18] do define them as storage fields.
- `#[pallet::event]` on the `Event` enum, events should be emitted when a notable action is done on the pallet. They are stored in the storage and can be listened to from outside the blockchain to trigger some other automated actions.
- `#[pallet::error]` on the `Error` enum, we can define custom errors in order to make the reasons for a failed dispatchable call more explicit.

Once the pallet is implemented, developers must write a weight function that should be representative of the dispatchable's computational cost. A tool has been developed along the FRAME library to help them in this task, the benchmarking module[19]. In order to use it, the developers have to write some benchmarking code for each dispatchable that will be fed to the FRAME's benchmarking module. It will basically fuzz the dispatchables and record the number of reads/writes to the database, the number of emitted events and the time of execution. It will then infer some weights from those values.

[16] `https://paritytech.github.io/substrate/master/frame_support/attr.pallet.html`
[17] `https://paritytech.github.io/substrate/master/frame_support/storage/types/struct.StorageValue.html`
[18] `https://paritytech.github.io/substrate/master/frame_support/storage/types/struct.StorageMap.html`
[19] `https://docs.substrate.io/main-docs/test/benchmark/`

The issue here is that the developers could have written bad benchmarks if the fuzzer did not trigger the most expensive path. If this is the case, the inferred weights may be way undervalued, allowing a cheap attack on the chain.

To sum up, the pallets are generic plug-and-play libraries. This property is very useful as we can reuse the implemented logic across different configuration environments. We can construct a small working toy example of a pallet, see Listing 4, where users can add numbers to the blockchain storage via the `add_number` dispatchable function that accesses the `StoredNumber` storage field. We can see the two generic types in the `Config` trait that will be made concrete when the Runtime is built. This way, the pallet can be reused across different runtimes with different parameters. Note that the pallet has not been benchmarked and the weights are random and do not represent the true cost.

Listing 4: Minimalist working pallet example, users can store up to `MaxSize` numbers in the storage.

```
#![cfg_attr(not(feature = "std"), no_std)]

pub use pallet::*;

#[frame_support::pallet]
pub mod pallet {
  use super::*;
  use frame_support::pallet_prelude::*;
  use frame_system::pallet_prelude::*;

  #[pallet::config]
  pub trait Config: frame_system::Config {
    type Event: From<Event<Self>> + IsType<<Self as frame_system::Config>::Event>;

    #[pallet::constant]
    type MaxSize: Get<u32>;
  }

  #[pallet::pallet]
  #[pallet::generate_store(pub(super) trait Store)]
  pub struct Pallet<T>(_);

  #[pallet::storage]
  pub(super) type StoredNumbers<T: Config> =
    StorageValue<_, BoundedVec<u32, T::MaxSize>, ValueQuery>;

  #[pallet::event]
  #[pallet::generate_deposit(pub(super) fn deposit_event)]
  pub enum Event<T: Config> {
    NewNumberStored { who: T::AccountId, n: u32, new_length: u32 },
  }

  #[pallet::error]
  pub enum Error<T> {
    StorageFull,
  }

  #[pallet::call]
  impl<T: Config> Pallet<T> {
    #[pallet::weight(42 + T::DbWeight::get().reads_writes(1,1))]
    pub fn add_number(origin: OriginFor<T>, n: u32) -> DispatchResult {
      let who = ensure_signed(origin)?;

      let new_length =
        <StoredNumbers<T>>::try_mutate(|stored_numbers| -> Result<u32, DispatchError> {
          stored_numbers.try_push(n).map_err(|_| Error::<T>::StorageFull)?;
          Ok(stored_numbers.len() as u32)
```

```
48          })?;
49
50          Self::deposit_event(Event::NewNumberStored { who, n, new_length });
51
52          Ok(())
53      }
54    }
55  }
```

3.3 Exploring the extrinsic's execution callpath

In order to have an idea of the path the analysis must take, it is crucial to understand how the calls are dispatched from the Substrate client to the final dispatchable function. In the following section, we will study in details the callpath of one extrinsic during the construction of a block. In a first step, we explore the callpath from the Substrate node's client to the boundary of the runtime. In a second step, we detail the callpath of the extrinsic execution from the runtime to the target dispatchable, this is the part we considered during our analysis.

From the client to the runtime. As we can see in the `basic_authorship.rs`[20] file when a `Client` tries to author a block, it will spawn a new `BlockBuilder`[21] instance, execute and push its own list of inherents, request extrinsics from the transaction pool, execute and push them in the block. The callpath for the extrinsics to reach the Wasm runtime during block construction with default execution strategy[22] is such as:

1. `BlockBuilder::apply_extrinsic_with_context`: this function is generated at compile time by the `decl_runtime_apis!` macro in `primitives::block-builder`
2. `RuntimeApiImpl::BlockBuilder_apply_extrinsic_runtime_api_impl`: this function is generated at compile time by the `impl_runtime_apis!` macro in `node::runtime`
3. `BlockBuilder::apply_extrinsic_call_api_at`: this function is generated at compile time by the `decl_runtime_apis!` macro in `primitives::block-builder`
4. `Client::call_api_at`
5. `LocalCallExecutor::contextual_call` with the following arguments:
   - `method`: `"BlockBuilder_apply_extrinsic"`
   - `arguments`: encoded extrinsic
   - `execution_manager`: `ExecutionStrategy::AlwaysWasm` (default for block construction)

   It will spawn a new `StateMachine` instance whose `executor` is a `NativeElseWasmExecutor`.
6. `StateMachine::execute_using_consensus_failure_handler`: will call the method `StateMachine::execute_aux` with `use_native` set to `false`.
7. `StateMachine::execute_aux`
8. `CodeExecutor::call`: it will create a `WasmInstance` of the runtime, there are actually two implementors of this trait: `WasmiInstance` (interpreter) and `WasmtimeInstance` (runtime). Since only the first one is listed in the documentation, we will stick with it.
9. `WasmiInstance::call_export`
10. `WasmiInstance::call` with the argument `method = InvokeMethod::Export("BlockBuilder_apply_extrinsic")`

[20] https://github.com/paritytech/substrate/blob/87052efd6be1bb07061a14dbb63a13ed7a81cfb2/client/basic-authorship/src/basic_authorship.rs#L341

[21] https://github.com/paritytech/substrate/blob/93cf073d7caac85f4f8f1fb894d1abf05362b917/client/block-builder/src/lib.rs

[22] https://docs.substrate.io/v3/advanced/executor/

11. `Wasmi::FunctionExecutor::call_in_wasm_module`

12. `ModuleRef::invoke_export`, `FuncInstance::invoke` and then `Interpreter::start_execution` will start executing the runtime by calling its `BlockBuilder_apply_extrinsic` method with the encoded extrinsic as its arguments.

    Note that the `BlockBuilder_apply_extrinsic` method does not exist as is in the Rust source code. It is exported by the Wasm binary and is the result of the compilation of the `api::dispatch` function generated by the `impl_runtime_apis!` macro. The runtime api's `dispatch` function does pattern matching on the method's name and dispatches the calls that way.

From the runtime to the dispatchable. We finally exited the client's code and entered the Wasm runtime, now it is time to dispatch the call. The entry target function is `BlockBuilder_apply_extrinsic` with the encoded extrinsic as its argument. Let's look at the callpath inside the blockchain's runtime:

1. `api::dispatch`: with the `"BlockBuilder_apply_extrinsic"` branch: this function is generated at compile time by the `impl_runtime_apis!` macro in `node::runtime`

2. `Runtime::apply_extrinsic`

3. `Executive::apply_extrinsic`

4. `Executive::apply_extrinsic_with_length`: At this point, the extrinsic is `CheckExtrinsic` and it implements `Applyable::apply`. Its field `function` has type `runtime::Call`, here it is the runtime-level `enum runtime::Call` generated by the `construct_runtime!` macro. It contains every callable pallet of the runtime.

5. `CheckedExtrinsic::apply`

   The two following functions are generated by `construct_runtime!` at compile time

6. `runtime::Call::dispatch`

7. `runtime::Call::dispatch_bypass_filter`: will pattern match on the runtime-level `enum Call` variant to determine to which pallet it should dispatch the call

8. `pallet::Call::dispatch_bypass_filter`: will pattern match on the pallet-level `enum Call` variant to determine which dispatchable function it should call. This function is generated at compile time by the `#[pallet::call]` macro.

To sum up, once the call enters the runtime, it passes through a two-steps dispatch process, the first one to the target pallet and the second one to the target dispatchable. We will use the knowledge of `pallet::Call::dispatch_bypass_filter` to show the tool the functions it must analyze.

4 Introduction to the Rust compiler

This section shows an overview of the Rust language and its native compiler, as well as how we can leverage the compiler driver made available by the compiler team. We will also present the Mid-level Intermediate Representation (MIR), an intermediate representation for Rust code which is used in the compilation process.

4.1 Rust and the Rust compiler

Rust is a powerful general-purpose language that is meant to be primarily for low-level systems programming, fast and memory safe. It achieves memory safety for references with the help of reference lifetimes, an ownership model, and a borrow checker, but no garbage collector which would add a resource overhead at runtime. For example, Rust will not allow the use of two mutable references at the same time. We chose to develop our tool in Rust mainly for the reason that Polkadot and Substrate are written in Rust, but also because the Rust compiler exposes an API that allows to register some callbacks that can be called at different phases of the compilation, see Fig 4. The different hooks are:

1. `after_parsing`: happens right after the parsing of token stream, but before macro expansion
2. `after_expansion`: happens right after macro expansion, Rust macros[23] are pieces of code that are able to write other code, and macro expansion is the generation of this new code
3. `after_analysis`: happens after MIR generation and optimization

The Rust compilation process works by incremental compilation and executes queries that are then memoized in a huge structure called `TyCtxt`[24]. It is the entry point for accessing compilation artifacts and for dealing with the AST, the HIR, the MIR, or the results of the typechecker for example.

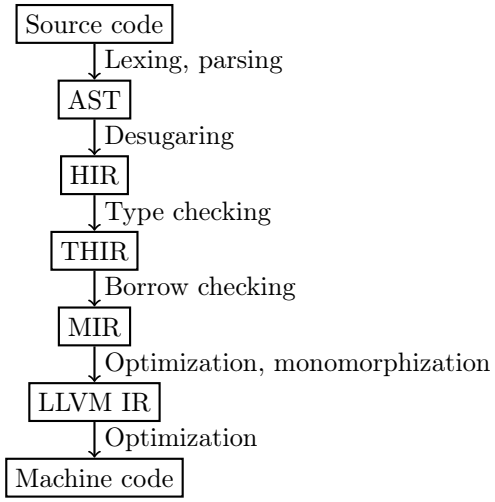

Figure 4: Phases of Rust compilation

The compiler is a huge and complex project that is still under active development[25] and we expect its API to be updated regularly to enable more features and be better documented with time.

4.1.1 Compiler driver

The Rust compiler is built like a big Lego, it is separated in many smaller crates, each representing a specific part or data structure used in the compilation process, which makes the compiler easier to understand and allows one to use part of it as libraries. One of the higher-level crates is called `rustc_driver`, it is responsible for gluing all the subcomponents together and running the

[23] `https://doc.rust-lang.org/book/ch19-06-macros.html`
[24] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/ty/struct.TyCtxt.html`
[25] `https://github.com/rust-lang/rust`

compiler.

A Rust package is called a *crate* and the language has a convenient tool to manage the different crates, that is `cargo`. This command will invoke the compiler for every crate dependency with the correct arguments. It is also possible to override the default compiler to redirect the calls to a custom compiler wrapper through the `RUSTC_WORKSPACE_WRAPPER` compilation flag.

In our case, the simplest way to interact with the compiler is to use the `rustc_driver` module. It only needs some compilation arguments and the set of callbacks, then it will automatically run the compilation as usual and invoke the given callbacks. To do so, we leveraged the power of `cargo`-generated arguments and `RUSTC_WORKSPACE_WRAPPER` to redirect the compilation to a custom compiler driver we wrote, which is responsible for invoking the compiler with the analysis callbacks when needed.

4.2 Rust's MIR

The (optimized) `MIR` [25] is a desugared representation of the Rust code that is way simpler to work with than the raw AST. The MIR is also used to run some optimization while still having some Rust-specific knowledge before the information is lost when the code is translated to LLVM-IR, hence the optimized MIR. It is built between the type checking and the LLVM-IR codegen compilation phases, it is generated per function and is based on the control-flow graph. Its building blocks are:

- **Locals**: memory locations allocated on the stack. They can represent local variables, function arguments or temporary variables. Locals are indexed from `0_` to `n_`. `0_` is always the `Local` containing the return value, `1_` to `p_` are the `p` function's arguments, if any, and `(p+1)_` to `n_` are the function's internal and temporary variables. It is important to mention that `Local`s can have projections. A projected `Local` is called a `Place`, there can be different kinds of projections, a good example is access to a `struct`'s field.
- **Basic blocks**: they are the nodes in the control-flow graph (CFG). Basic blocks are made of zero or more statements and exactly one terminator.

  - **statements**: actions with only one successor

  - **terminator**: actions with one or more successors, it is always the last element of a basic block

There are multiple kinds of `Statements`[26] and `Terminators`[27] implemented by the MIR. Some of them are not relevant to our analysis, like `StatementKind::StorageLive(_)` which is useful to the borrow checker, or `TerminatorKind::SwitchInt(x)` which, depending on the value of `x` redirects the execution flow to a basic block or another. We are only interested in primitive operations, type history and precision, and function calls. Table 1 lists the `StatementKind`s and `TerminatorKind` relevant for our resource analysis and quickly describes their role.

The majority of the computational costs happen in the `StatementKind::Assign(Place,Rvalue)`, which has the form `Place = Rvalue` in the MIR, where `Rvalue`[28] wraps different sorts of operations that are listed in Table 2.

Apart from the `Terminator` and `Statement` `struct`s, there are two other `struct`s that we used a lot during the analysis: `DefId`[29] and `Ty::ty`[30]. `DefId`s are identifiers for definitions across compilation artifacts, they can refer to types or functions definition for example. We used it to redirect the analysis to a given function pointed by `TerminatorKind::Call`, by getting its MIR from the function's `DefId`, or to query the type of a `DefId`. `Ty::ty` is the internal representation of the types in the MIR, after type checking and type inference.

Listing 5 illustrates the translation from Rust source code (Listing 5) to `MIR`. The program starts with an accumulator `accum`, a mutable reference to this variable will be passed to the function `loopy_add` when it is called in the terminator of basic block 0. `loopy_add` will iterate through the range of integers `[1,10[` and add each value to the accumulator, updating the initial `accum`

[26] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/syntax/enum.StatementKind.html`
[27] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.TerminatorKind.html`
[28] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.Rvalue.html`
[29] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_span/def_id/struct.DefId.html`
[30] `https://rustc-dev-guide.rust-lang.org/ty.html`

`StatementKind::Assign(Place,Rvalue)`	flow of value, may be responsible for unary/binary operations depending on the `Rvalue`. Also used for collecting more precise type information
`StatementKind::SetDiscriminant(Place,VariantIdx)`	`Place` has the type of an `enum`, the statement sets the index of variant
`TerminatorKind::Call`	a function call
`TerminatorKind::Goto`, `TerminatorKind::SwitchInt` `TerminatorKind::Return`, ...	`Terminator`s used in the dataflow framework for joining the domains

Table 1: List of `StatementKind`s and `TerminatorKind` relevant to our analysis.

`Use(Operand)`	yields the operand unchanged
`Repeat(Operand, Const)`	creates an array of constant length where each element is the value of the operand
`Ref(Region, BorrowKind, Place)`	creates a reference to the place, of the given `BorrowKind` (unique, shared, mutable,...)
`ThreadLocalRef(DefId)`	creates a reference to the given thread local
`AddressOf(Mutability, Place)`	creates a pointer to the place with the given mutability (mutable or unmutable)
`Len(Place)`	returns the length of the place, i.e. the length of an array
`Cast(CastKind, Operand, Ty)`	performs casting that can be done with the use of the `as` keyword
`BinaryOp(BinOp, (Operand, Operand))`	binary operations: $+, -, \times, \div, \%, \oplus, \wedge, \vee, \ll, \gg, ==, <, \leq, !=, \geq, >$,`ptr.offset`
`CheckedBinaryOp(BinOp, (Operand, Operand))`	same as `BinaryOp` but yields an additional `bool` indicating an error condition
`NullaryOp(NullOp, Ty)`	computes either the size of a value, or the minimum alignment of the type, depending on `NullOp`
`UnaryOp(UnOp, Operand)`	unary operations: $!, -$
`Discriminant(Place)`	computes the discriminant of the place
`Aggregate(AggregateKind, Vec<Operand>)`	creates an aggregated value like a tuple or a struct
`ShallowInitBox(Operand, Ty)`	transmutes `*mut u8` into shallow-initialized `Box<T>`
`CopyForDeref(Place)`	equivalent to a read from a place at the codegen level

Table 2: List of `Rvalue`s with their description.

since a *mutable* reference is passed.

Listing 5: Rust code snippet computing $\sum_{i=1}^{9} i$ in an over-engineered fashion.

```
fn loopy_add(accum: &mut usize) {
    for i in 1..10 {
        *accum += i;
    }
}

fn main() {
    let mut accum = 0;
    loopy_add(&mut accum);
}
```

Note on closures in MIR. The language also supports closures, these are anonymous functions that are able to capture their environment, see Fig. 17 and Section 6.4.4 for more details. The point here is that as for more usual functions, each closure will have its own unique associated MIR type, but it will also have an additional data structure that will hold information about the captured environment.

Limitations due to monomorphization. An important note is that monomorphization happens after MIR optimization, during code generation. Monomorphization is the process of resolving types for generic traits and functions so the compiler can generate specific code for each concrete type. This also means that in the MIR, when a generic function is called, the call resolves to the generic definition and not to the concrete definition. Also, in the case where some types are still generic, monomorphization and codegen will be delayed until all concrete types are known [31].

[31] `https://users.rust-lang.org/t/trait-functions-implementation-in-mir/76002/6`

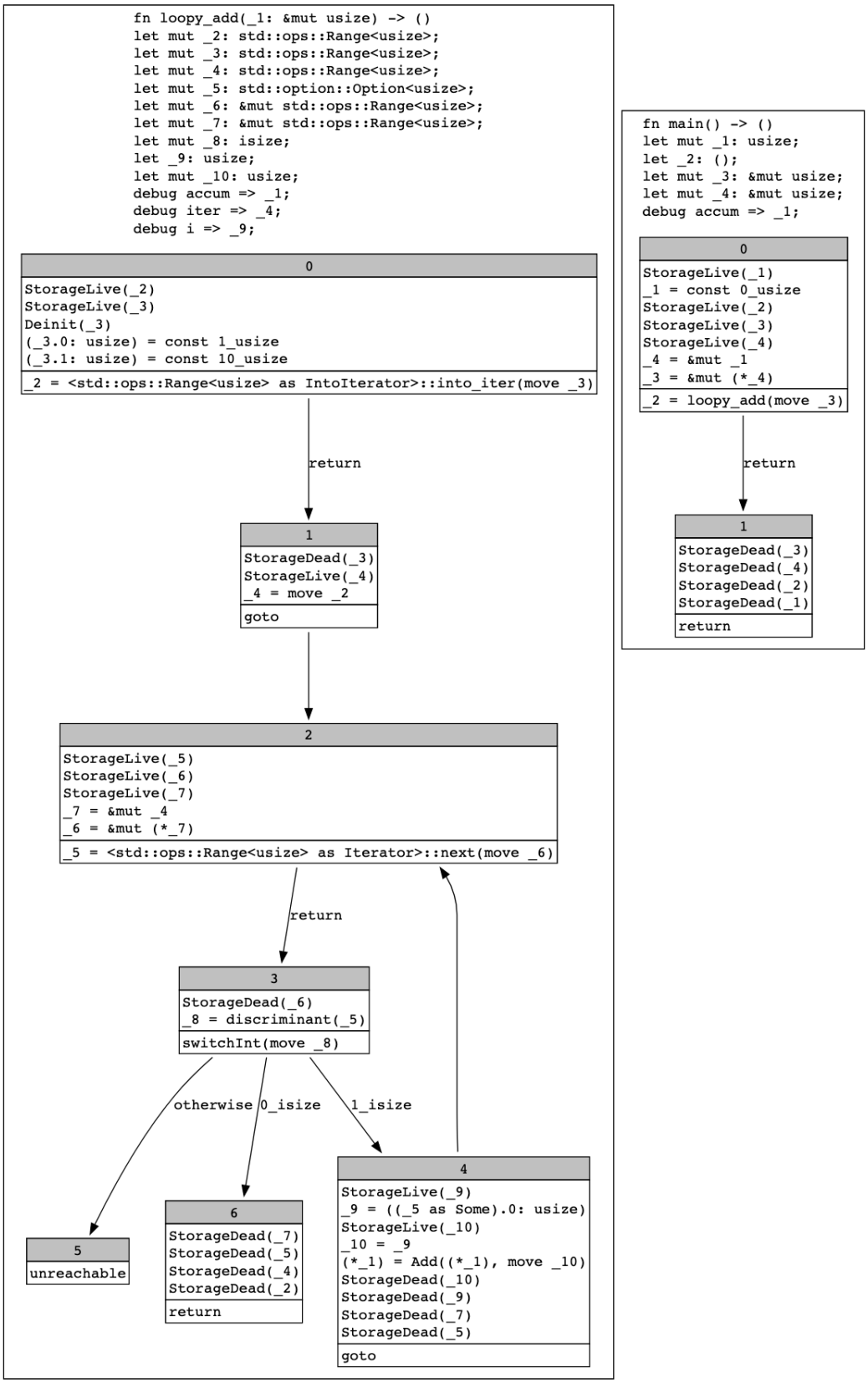

Figure 5: Generated MIR for the `main` and `loopy_add` functions of Listing 5.

5 Static analysis

Following the problem statement from Section 2, we perform static analysis on the code of Substrate pallets in order to infer resource bounds.

5.1 Static analysis background

Static program analysis is a field of the subtle art of program analysis. The most accurate way to analyze a program, e.g., to check whether it satisfies a given property, would be to run it with every possible input and check all the resulting behaviors. Of course, this is infeasible in general. Not only that but it is actually uncomputable. Indeed, Rice's theorem tells us that any nontrivial semantic property is undecidable. (A semantic property [15] is a property of program behaviors. In contrast, a syntactic property is a property of the code of the program, such as whether it contains loops.) That is why for (infinitely) many programs, static analysis has to flag false alarms, or miss true alarms, or both. Depending on this situation we distinguish:

- **sound analysis**: if the analysis concludes that a property P holds, then P indeed holds (the analysis never misses true alarms, that is, failure to satisfy the property);

- **complete analysis**: if a property P holds, then the analysis will conclude that P holds (the analysis never flags false alarms, that is, false claims that the property holds).

Ideally, we would like an analysis that is both sound and complete, but as we said, this is impossible in general. That is why analyses usually focus on soundness or on completeness. Since we aim for securing Substrate pallets against extreme worst-case behaviors, we design our analysis to be sound. That is, the analysis should always provide an upper bound on the resource cost.

5.1.1 Abstract semantics

The method we shall employ to design sound static analyses is the abstract interpretation of Cousot and Cousot developed in the late 70's [2]. Abstract interpretation phrases the soundness of the analysis as *over-approximation* of the operational semantics of a program: the static analysis is supposed to compute a *superset* (seen as a property) of the actual program behaviors. This over-approximation is called the *abstract semantics*, while the approximated operational semantics is called the *concrete semantics*. The interpretation bit comes from the way the abstract semantics is computed by mimicking the computation of the concrete semantics, both described next.

In Section 2 we already gave an example of concrete semantics: the set S of state sequences *traced* by the program. The computation of S takes place in the space, call it $\mathbb{D}$, of all possible sets of state sequences, and proceeds inductively. The induction is formalized by partially ordering the space $\mathbb{D}$ by subset inclusion, and defining S as the least fixed point of a monotone map $f : \mathbb{D} \to \mathbb{D}$ extracted from the program. Given an argument, this map returns the union of two sets: 1) the set of all traces consisting of just a single initial state, and 2) the set of all single-step extensions of traces in the argument. A step produces a new state by executing the next instruction scheduled by the program relative to the last state of the trace being extended. The function that captures how the instruction transforms this last state is called a *concrete transfer function*. For example, the concrete transfer function for $x = x * z$ updates the value of x in the state it receives as argument producing a new state. (Non-deterministic instructions have many-valued transfer functions.)

Abstract interpretation mimics this situation. Instead of applying the transformation in $\mathbb{D}$, the inductive computation proceeds in another partially ordered set $\mathbb{D}^\#$, while the monotone map f is replaced with another monotone map $f^\# : \mathbb{D}^\# \to \mathbb{D}^\#$. Just as the map f is constructed from applications of concrete transfer functions (corresponding to the instructions in the program), the map $f^\#$ is constructed from applications of *abstract transfer functions*. An abstract semantics is then defined as any fixed point $S^\#$ of $f^\#$, the least one being the most precise (but it could be more difficult to compute).

In practice, $\mathbb{D}^\#$ (just like $\mathbb{D}$) is constructed using simpler partially ordered sets. Consider, for example, the simple code snippet in Listing 6 where the abstract semantics of interest is the sign of each variable at the end of the `simple` function. In this case, $\mathbb{D}^\#$ is built by letting the program variables range not in the integers but in the elements of sign abstract domain (shown in Fig. 6).

Listing 6: Effects of the sign abstract transfer functions.

```
fn simple() {        // x ↦ ⊥, y ↦ ⊥, z ↦ ⊥
    let mut x = 3;   // x ↦ ≥ 0, y ↦ ⊥, z ↦ ⊥
    let y = 0;       // x ↦ ≥ 0, y ↦ = 0, z ↦ ⊥
    let mut z = -2;  // x ↦ ≥ 0, y ↦ = 0, z ↦ ≤ 0
    x = x * z;       // x ↦ ≤ 0, y ↦ = 0, z ↦ ≤ 0
    z = z * z;       // x ↦ ≤ 0, y ↦ = 0, z ↦ ≥ 0
}                    // x ↦ ≤ 0, y ↦ = 0, z ↦ ≥ 0
```

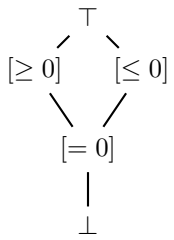

Figure 6: The sign domain.

The comments in the code show the effect of the abstract transfer function for each statement. The parts relevant for our example of the sign abstract transfer function can be described as follows:

- if variable declaration or no-op (`lhs = rhs`): x for $x \leftarrow \mathit{sign}(\texttt{rhs})$
- if assignment (`lhs = op(rhs)`):
  - if the operation is `+`:

$$\begin{cases} 0 & \text{if } \texttt{lhs} \to 0 \text{ and } \texttt{lhs} \to 0 \\ [\geq 0] & \text{if } \texttt{lhs} \to [\geq 0] \text{ and } \texttt{lhs} \to [\geq 0] \\ & \text{or if one of } \{\texttt{lhs}, \texttt{rhs}\} \to [\geq 0] \text{ and the other} \to 0 \\ [\leq 0] & \text{if } \texttt{lhs} \to [\leq 0] \text{ and } \texttt{lhs} \to [\leq 0] \\ & \text{or if one of } \{\texttt{lhs}, \texttt{rhs}\} \to [\leq 0] \text{ and the other} \to 0 \\ \top & \text{otherwise} \end{cases}$$

  - if the operation is `*`:

$$\begin{cases} 0 & \text{if } \texttt{lhs} \to 0 \text{ or } \texttt{rhs} \to 0 \\ [\geq 0] & \text{if } \texttt{lhs} \to [\geq 0] \text{ and } \texttt{rhs} \to [\geq 0] \\ & \text{or if } \texttt{lhs} \to [\leq 0] \text{ and } \texttt{rhs} \to [\leq 0] \\ [\leq 0] & \text{if one of } \{\texttt{lhs}, \texttt{rhs}\} \to [\leq 0] \text{ and the other} \to [\geq 0] \\ \top & \text{otherwise} \end{cases}$$

Now, in order to judge whether the abstract semantics over-approximates the concrete one, we use a monotone *concretization function* that relates the concrete domain to the abstract domain:

$$\gamma : \mathbb{D}^{\#} \to \mathbb{D}.$$

For example, the concretization function for the sign domain has the following definition

$$\gamma(\bot) = \varnothing \quad \gamma(\bot) = \mathbb{Z} \quad \gamma([= 0]) = \{0\} \quad \gamma([\leq 0]) = \{z \in \mathbb{Z} \mid z \leq 0\} \quad \gamma([\geq 0]) = \{z \in \mathbb{Z} \mid z \geq 0\}$$

The over-approximation criterion can then be stated as ($\sqsubseteq$ denotes the respective ordering):

- concrete semantics: least $S \in \mathbb{D}$ such that $f(S) \sqsubseteq S$;
- abstract semantics: any $S^{\#} \in \mathbb{D}^{\#}$ such that $f^{\#}(S^{\#}) \sqsubseteq S^{\#}$;
- over-approximation: $S \sqsubseteq \gamma(S^{\#})$.

Here, we used a more-general formulation with the so-called post fixed points ($f(S) \sqsubseteq S$) instead of fixed points. The least post fixed point (of a monotone f) is actually a fixed point, since by monotonicity $f(S)$ is a post fixed point too ($f(f(S)) \sqsubseteq f(S)$), and therefore $S \sqsubseteq f(S)$. Thus, compared to using fixed points, this is a generalization only of the abstract semantics. This generalization is useful, because post fixed points are somewhat easier to find than fixed points.

It turns out that the over-approximation required by soundness is guaranteed if the abstract monotone map $f^\#$ over-approximates the concrete one f in the sense that, for all $D^\# \in \mathbb{D}^\#$:

$$f \circ \gamma(D^\#) \sqsubseteq \gamma \circ f^\#(D^\#)$$

This way, the soundness of the analysis is reduced to designing an $f^\#$ that over-approximates f.

The partially ordered sets that appear in the various semantics are called *domains*. For our purposes, we shall relax the partial ordering to a pre-ordering, and also insist on the presence of a join operation. Hence, a domain is a structure L that consists of:

- a pre-ordering: $\sqsubseteq$ (reflexive and transitive relation)
- a top element: $\top$
- a bottom element: $\bot$
- a join operator: $join(x, y) = x \sqcup y$, such that for all $z \in L$: $x, y \sqsubseteq z \iff x \sqcup y \sqsubseteq z$.

5.1.2 Structuring the semantics

We need to express the maps f and $f^\#$ using transfer functions for the primitive instructions that appear in the program. In our case, the program is given by a control flow graph (CFG), and the primitive instructions are the elements of the basic blocks in the graph. In order to easily apply the transfer functions, it is useful to structure the elements of $\mathbb{D}/\mathbb{D}^\#$ in a very specific way. The idea is to factor a domain element into smaller pieces attached to each instruction in the CFG. For example, the concrete semantics consists of a set of traces. We decompose this set of traces into disjoint subsets, grouping them according to the next scheduled instruction as indicated by the last state in a trace. (Every state carries extra control information, e.g., a program counter, that determines the schedule in which instructions are executed.) This way, to apply the effect of the transfer function corresponding to an instruction, we simply look up the group of traces for that instruction, and extend them by applying the transfer function to the last state. The result is then joined into the group attached to each successor instruction in the CFG.

Similarly, we structure $\mathbb{D}^\#$ as the domain of tuples (ordered pointwisely)

$$\Lambda \to \mathbb{D}'^\#,$$

where the indices in Λ are called *program points* that correspond to the instruction occurrences in the CFG, and $\mathbb{D}'^\#$ is some other domain. To express $f^\#$, let E be the set of pairs (λ, λ') of indices for which the CFG permits control to flow from λ to λ'. More specifically, the elements of a basic block flow to their successor in the block, except for the terminator, which flows to the first elements of the successor basic blocks. Now, we assume that a transfer function $[\![e]\!] : \mathbb{D}'^\# \to \mathbb{D}'^\#$ is given for each such pair $e \in E$. With this we can define $f^\# : \mathbb{D}^\# \to \mathbb{D}^\#$ as follows

$$f^\#(D^\#) = \lambda' \mapsto \bigsqcup_{(\lambda,\lambda') \in E} [\![(\lambda, \lambda')]\!](D^\#(\lambda)).$$

That is, each program point λ' is updated with the join of all results of applying transfer functions to predecessor program points. This generalizes the above scheme for the concrete domain $\mathbb{D}$.

This way, $f^\#$ is constructed uniformly from the CFG, and the domain $\mathbb{D}'^\#$ together with the transfer functions $\mathbb{D}'^\# \to \mathbb{D}'^\#$ for every instruction. These transfer functions are usually defined at once for all possible instructions and then reused. This allows us to reason about soundness on the level of transfer functions: If f is expressed equivalently via a concrete counterpart $\mathbb{D}'$ and concrete transfer functions, and furthermore a concretization $\gamma' : \mathbb{D}'^\# \to \mathbb{D}'$ is given, then soundness follows from the abstract transfer functions over-approximating the respective concrete transfer functions.

5.1.3 Computing the semantics

The concrete and the abstract semantics were defined as post fixed points of suitable monotone functions. The existence of such post fixed points as well as their computation heavily depends on the structure of the underlying domains (and any additional properties of the monotone functions). The concrete domain is usually a complete lattice, in which case Tarski's fixed-point theorem

guarantees the existence of a least fixed point for every monotone map, hence the concrete semantics is well defined. Unfortunately, even for such well-structured domains, computing the semantics is impossible (i.e., it requires infinitely many computational steps).

If the domain $\mathbb{D}^\#$ has a finite height h, which bounds the length of all chains, then we can compute the least (post) fixed point of $f^\#$ as the last element of the (necessarily finite) chain

$$\bot \sqsubseteq f^\#(\bot) \sqsubseteq {f^\#}^2(\bot) \sqsubseteq {f^\#}^3(\bot) \sqsubseteq \cdots \sqsubseteq {f^\#}^h(\bot).$$

However, this is not the case even for relatively simple domains. Consider for example, the domain of integer intervals. Its elements are inequality constraints of the form

$$l_x \leq x \leq h_x,$$

where l_x and $h_x \in \mathbb{Z}^\infty_{-\infty}$. Intervals are ordered by inclusion. As evident from the picture in Fig. 7, the interval domain is not of finite height:

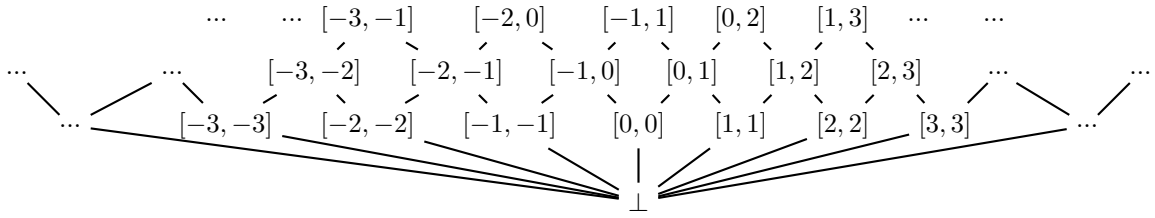

Figure 7: The domain of integer intervals.

The transfer function for addition of intervals can be defined as

$$[a, b] + [c, d] = [a + c, b + d].$$

If we try to compute the interval abstract semantics for the program in Fig. 8 using the above scheme we will not reach a (post) fixed point.

```
1 fn loopy_simple(n: i32) {
2     let mut i = 0;
3     while i < n {
4         i = i + 1;
5     }
6 }
```

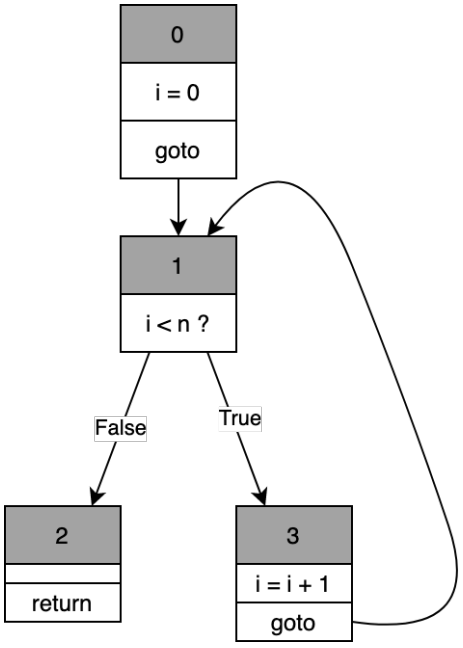

Figure 8: A function with an unbounded loop.

The reason for that is that: 1) there is no a priori bound on n, i.e., its value is $[-\infty, +\infty]$, and 2) the interval bound for i while computing the semantics will always be $[0, j]$ for $j < +\infty$, hence the sequence of reached abstract elements will be ever increasing.

To remedy such problems with convergence, one may introduce a **widening operator** [15] $\nabla : \mathbb{D}^\# \times \mathbb{D}^\# \to \mathbb{D}^\#$ that satisfies the following conditions:

1. over-approximation: $x \sqcup y \sqsubseteq x \nabla y$ for all $x, y \in \mathbb{D}^\#$;

2. termination: for any sequence $(x_i \in \mathbb{D}^\#)_0^\infty$, the sequence (x') stabilizes in finite time:

$$x'_0 = x_0 \qquad\qquad x'_{n+1} = x'_n \nabla x_n.$$

In case the original iteration does not converge, we can instead apply iteration with widening:

$$x_0 = \bot \qquad\qquad x_{n+1} = x_n \nabla f^\#(x_n)$$

This sequence necessarily converges, and moreover it does so at a post fixed of $f^\#$.

However, we shall not rely on widening to cope with convergence. Instead, we limit the fragment of programs which we analyze statically. We consider only programs without iterative constructs, such as loops or recursion, which are guaranteed to converge in finite time even if the domain is of infinite height. In case a part of the code contains such constructs, one can manually provide a specialized transfer function only for this part, and still analyze the code.

5.1.4 Interprocedural analysis

So far we did not discuss how to deal with one very important construct, namely function calls. There are multiple ways to do that, and we choose to handle them as closely as possible to the concrete semantics (which in case of recursion introduces problems with convergence). We consider a modified control flow graph obtained in the following way. We build this CFG given an initial function call to be analyzed. We take a copy of the CFG of the called function, and recursively apply the CFG construction procedure to all calls appearing there. Then we modify the call and return sites by connecting them to the respective entry and exit nodes of the corresponding recursively built CFG. This construction allows us to precisely analyze function calls, but of course, if there is a loop in the call graph, the construction will not terminate. The transfer function of calls and returns simply initializes the respective callee (for calls) and caller (for returns) variables.

Next we describe the various domains used that constitute our static resource analysis.

5.2 Event variants domain

During the analysis we will need to collect information about which variant of the `enum Event` can be called from a given `deposit_event` call site. More details will be given later in Section 6.4.7.

We first define the set `Variants`, that is able to capture the property that on a given `deposit_event` call site, the event's variant is not fixed and can be decided based on a branching structure at runtime.

We then define the following abstract domain E per function as a mapping from a `Location`[32] (`deposit_event` call site) to a `Variants`, see Listing 12:

$$E : \texttt{Location} \to \texttt{Variants}$$

The *join* operator takes x, the domain we want to update, and y, the domain we want to join into it, it is defined as follows:

$$join(x, y) = x \sqcup y \quad | \quad x, y \in E$$

with

$$x \sqcup y = \begin{cases} y[l] & \text{if } x[l] \text{ does not exist} \\ \texttt{Or}(x[l], y[l]) & \text{otherwise} \end{cases} \qquad \forall l \text{ in } y.keys$$

The bottom element $\bot_E$ being an empty `Location` $\to$ `Variants` mapping. Since the number of `Location` where `deposit_event` and the set of events available for a pallet are finite, the lattice is trivially of finite height and its top element $\top_E$ is the disjunction of all the mappings where the keys are `Location` is a `deposit_event` call site and their associated `Variants` is the set of events in the pallet.

The soundness of this analysis is guaranteed here since we over-approximate the concrete runtime behavior by keeping the set of all possible variants for each `deposit_event` call site.

[32] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/struct.Location.html`

5.3 Cost language

Concrete domain. The cost language is an abstraction of the concrete cost semantics itself. Since the cost can vary from 0 to ∞ (e.g., the time cost of a non-terminating program) the domain of costs consists of the natural numbers plus ∞, ordered as usual:

$$\mathbb{N}^{\infty} = \{0 < 1 < 2 < \cdots < \infty\}.$$

The concrete cost semantics is based on the cost of individual execution traces. Since we care for the worst-case executions, which is a kind of aggregate semantics, the cost semantics domain (not to be confused with the domain of costs above) is the function space

$$\mathbb{N}^{\texttt{CostParameter}} \to \mathbb{N}^{\infty}.$$

A function f in this space takes as argument a tuple of upper bounds on some pre-defined cost parameters (such as the length of a function argument), and returns the worst-case cost of a possibly partial execution that starts in a state in which the cost parameters satisfy the given upper bounds. In the terms used in Section 2.1, we have

$$\mathsf{Metric} = \mathbb{N}^{\texttt{CostParameter}} \qquad\qquad \mathsf{Cost} = \mathbb{N}^{\infty}.$$

The function space $\mathbb{N}^{\texttt{CostParameter}} \to \mathbb{N}^{\infty}$ is partially ordered pointwise:

$$f \sqsubseteq g \iff \forall \rho \in \mathbb{N}^{\texttt{CostParameter}}.\, f(\rho) \leq g(\rho).$$

Abstract domain. Instead of working directly with the concrete functions, our abstract domain represents them as expressions from a *cost language*. Going back to the example from Section 2, we represent the net time cost of the binary search implementation with the expression

$$\mathcal{O}(\texttt{Log}(\texttt{items.len()})),$$

where `items.len()` denotes the length of the items argument of the binary search function.

The first component of the cost language are program variables or symbols

$$\langle \texttt{Variable} \rangle \;::=\; \texttt{Id(id, span)} \mid \texttt{String}$$

We represent them in two ways: i) as a Rustc `id` plus a `span` (the region of code where the variable is defined),[33] or ii) simply as a string that allows us to identify the symbol `"MaxSize::get()"`.

From variables we can extract the various cost parameters

$$\begin{aligned}\langle \texttt{CostParameter} \rangle \;::=\;& \texttt{ValueOf(Variable)} \mid \texttt{SizeOf(Variable)} \mid \\ & \texttt{ReadsOf(Variable)} \mid \texttt{WritesOf(Variable)} \mid \\ & \texttt{SizeDepositedOf(Variable)} \mid \\ & \texttt{StepsOf(Variable)} \mid \texttt{LengthOf(Variable)}\end{aligned}$$

Each cost parameter is something we cannot compute in advance: `ValueOf` represents the value returned by a function, e.g., `ValueOf("MaxSize::get()")`[34], `SizeOf` stands for the size of a type whose size is unknown. `ReadsOf, WritesOf, SizeDepositedOf` and `StepsOf` represent the bytes that are read and written, the size of the events and the number of steps executed by some functions that cannot be analyzed. It is important to note that `StepsOf` does not account for the sizes of the function's arguments and we assume uniform upper bounds for calling such a function.

Finally, a cost expression abstracts a function element in the concrete domain:

$$\begin{aligned}\langle \texttt{CostExpr} \rangle \;::=\;& \\ & \mid \texttt{Parameter(CostParameter)} \\ & \mid \texttt{Infinity} \\ & \mid \texttt{Scalar(u64)} \\ & \mid \texttt{Add(Vec<CostExpr>)} \\ & \mid \texttt{Max(Vec<CostExpr>)} \\ & \mid \texttt{ScalarMul(u64,CostExpr)} \\ & \mid \texttt{ParameterMul(CostParameter,CostExpr)} \\ & \mid \texttt{Log(CostExpr)} \\ & \mid \texttt{BigO(CostExpr')}\end{aligned}$$

[33] https://doc.rust-lang.org/nightly/nightly-rustc/rustc_span/span_encoding/struct.Span.html

[34] https://paritytech.github.io/substrate/master/frame_support/traits/trait.Get.html#tymethod.get

Cost expressions include constants from the domain of costs, parameters, and various combinations: sum, multiplication by a scalar, multiplication by a cost parameter, maximum, and logarithm. We also include the `BigO` operator that abstracts a function f to its class $\mathcal{O}(f)$. Here, `CostExpr'` denotes the fragment of the language without `BigO` itself, i.e, `BigO` cannot be nested.

To define the meaning of cost expressions, we need to define how expressions concretize as functions in the function space $\mathbb{N}^{\texttt{CostParameter}} \to \mathbb{N}^\infty$ (ordered pointwisely). However, because of the `BigO` operator, an expression cannot concretize to just one function, but it needs to concretize to a set of functions. Moreover, we also need to define how these subsets are compared. In total, we need a domain $\mathbb{C}$ whose elements are sets of functions, and a concretization function

$$\gamma_{\texttt{CostExpr}} : \texttt{CostExpr} \to \mathbb{C}.$$

The concretization function essentially extends the familiar operations in the cost language to operate not on single functions but on sets of functions. This is easily done by interpreting such sets as non-deterministic choices of values and extending the operations to all possible such choices. For example, one intuitive way to define addition (which we will update slightly below) is

$$S_1 + S_2 = \{f_1 + f_2 \mid f_1 \in S_1, f_2 \in S_2\}.$$

Defining the partial ordering is more difficult. We know that expressions not containing `BigO` stand for single functions. We would like to preserve the pointwise ordering on these functions, and more generally, to embed the partially ordered set $\mathbb{N}^{\texttt{CostParameter}} \to \mathbb{N}^\infty$ into $\mathbb{C}$, and even preserve the top and bottom elements. For example, the top element of the function space is the constant function returning ∞, which should concretize to the top element of $\mathbb{C}$. Another constraint, which comes from the `BigO` operation, is that $\mathbb{C}$ must include all sets in the form $\mathcal{O}(f)$.

These constraints can be satisfied by letting $\mathbb{C}$ equal the non-empty *down-closed subsets*:

$$\mathbb{C} = \{S \subseteq \mathbb{N}^{\texttt{CostParameter}} \to \mathbb{N}^\infty \mid S \neq \varnothing, (\forall f.\, f \in S \iff \exists g \in S.\, f \leq g)\}.$$

This becomes a lattice when ordered by subset inclusion, with join given by set union:

$$S_1 \sqcup_{\mathbb{C}} S_2 = S_1 \cup S_2.$$

The constraints are satisfied since every $\mathcal{O}(f)$ is down-closed, and the function space embeds via

$$g \mapsto \{f \in \mathbb{N}^{\texttt{CostParameter}} \to \mathbb{N}^\infty \mid f \leq g\}.$$

Finally, the concretization function can readily be defined by structural induction as follows:

$$\begin{aligned}
\gamma_{\texttt{CostExpr}}(\texttt{Parameter}(p)) &= \{f \mid \forall \rho.\, f(\rho) \leq \rho[p]\} \\
\gamma_{\texttt{CostExpr}}(\texttt{Infinity}) &= \{f \mid \forall \rho.\, f(\rho) \leq \infty\} \\
\gamma_{\texttt{CostExpr}}(\texttt{Scalar}(s)) &= \{f \mid \forall \rho.\, f(\rho) \leq s\} \\
\gamma_{\texttt{CostExpr}}(\texttt{Add}(e_1, \ldots, e_n)) &= \{f \mid \exists g_i \in \gamma_{\texttt{CostExpr}}(e_i).\, \forall \rho.\, f(\rho) \leq \textstyle\sum_i g_i(\rho)\} \\
\gamma_{\texttt{CostExpr}}(\texttt{Max}(e_1, \ldots, e_n)) &= \{f \mid \exists g_i \in \gamma_{\texttt{CostExpr}}(e_i).\, \forall \rho.\, f(\rho) \leq \max_i g_i(\rho)\} \\
\gamma_{\texttt{CostExpr}}(\texttt{ScalarMul}(s, e)) &= \{f \mid \exists g \in \gamma_{\texttt{CostExpr}}(e).\, \forall \rho.\, f(\rho) \leq s \cdot g(\rho)\} \\
\gamma_{\texttt{CostExpr}}(\texttt{ParameterMul}(p, e)) &= \{f \mid \exists g \in \gamma_{\texttt{CostExpr}}(e).\, \forall \rho.\, f(\rho) \leq \rho[p] \cdot g(\rho)\} \\
\gamma_{\texttt{CostExpr}}(\texttt{Log}(e)) &= \{f \mid \exists g \in \gamma_{\texttt{CostExpr}}(e).\, \forall \rho.\, f(\rho) \leq \log g(\rho)\} \\
\gamma_{\texttt{CostExpr}}(\texttt{BigO}(e)) &= \{f \mid \exists g \in \gamma_{\texttt{CostExpr}}(e).\, f \in \mathcal{O}(g)\}.
\end{aligned}$$

One readily verifies that for every $e \in \texttt{CostExpr'}$ (an expression not containing `BigO`) the concretization $\gamma_{\texttt{CostExpr}}(e)$ has a maximum element and this element equals the expected interpretation of e as an element of the function space $\mathbb{N}^{\texttt{CostParameter}} \to \mathbb{N}^\infty$. This follows by structural induction from the monotonicity of the language operations w.r.t. to the pointwise ordering on the functions.

The domain of cost expression has a join operation defined trivially:

$$e_1 \sqcup_{\texttt{CostExpr}} e_2 = \texttt{Max}(e_1, e_2),$$

which gives rise to the pre-order (reflexive and transitive but not necessarily anti-symmetric):

$$e_1 \sqsubseteq_{\texttt{CostExpr}} e_2 \iff \gamma_{\texttt{CostExpr}}(\texttt{Max}(e_1, e_2)) = \gamma_{\texttt{CostExpr}}(e_2).$$

As expected, the abstract join over-approximates the concrete join, i.e., for all $e_1, e_2 \in \texttt{CostExpr}$:

$$\gamma_{\texttt{CostExpr}}(e_1) \cup \gamma_{\texttt{CostExpr}}(e_2) \subseteq \gamma_{\texttt{CostExpr}}(\texttt{Max}(e_1, e_2)),$$

which ensures that $\gamma_{\texttt{CostExpr}}$ is monotone. Indeed, if $e_1 \sqsubseteq_{\texttt{CostExpr}} e_2$, then

$$\gamma_{\texttt{CostExpr}}(e_1) \subseteq \gamma_{\texttt{CostExpr}}(e_1) \cup \gamma_{\texttt{CostExpr}}(e_2) \subseteq \gamma_{\texttt{CostExpr}}(\texttt{Max}(e_1, e_2)) = \gamma_{\texttt{CostExpr}}(e_2).$$

5.4 The analysis domain

The analysis domain is divided into two subdomains. The first one, the counting domain C, is responsible for collecting the cost of a function. The second subdomain is here to help the counting domain to provide a more precise analysis, it is responsible for tracking some type information about each of the `Local`.

5.4.1 `Local`'s type information tracking

When analyzing a function call during interprocedural analysis, we need to know the type context from which the function is called to have a more precise result. In order to achieve this goal, we track the type of each `Local` throughout the analysis. To that end we use the mapping `LocalsInfo` : `Local` $\rightarrow$ `LocalInfo`. A `LocalInfo` contains:

- `length_of`: this is an optional field, it will hold the length of the `LocalInfo` if it is a dynamically sized vector, we reuse the cost language and track it as a `Cost`. The length of a vector is shared between its references so if it gets updated in a part of the code, the change is reflected in the root vector. This may break the invariant that the input set of the analysis should not change but since the tool does not support loops and recursion it is not problematic,
- `ty`: a vector containing the history of Rust's MIR type `Ty::ty` of the `Local`. We need this type history because Rust contains generic types that are instantiated later during monomorphization. But since the MIR cannot be monomorphized for our analysis, we keep the history of such instantiations, from the most imprecise to the most precise. For example, the first type can be `F: impl FnOnce` and the second, most recent and precise type, will be the type of a concrete closure, not only "something that is a closure".
- `members`: the list of its fields if it is a `struct`, or the list of its variants if it is an `enum`, for simplicity "fields" and "variants" will be grouped under the term "members". Each member is itself a `LocalInfo`.

Note on references and aliasing. For the sake of typing model simplicity, we chose to abstract away the references so the analysis works directly with the underlying types. To do so, we peel the references of the `Ty::ty`, e.g. we recursively inspect the `ty` field of `TyKind::Ref(_, ty, _)` and extract it when we access the underlying type.This diverges from the concrete Rust semantics but does not affect the correctness of the analysis since the only link we need to track between aliases is the length of vectors, which is tracked with the `length_of` field.

Joining `LocalsInfo`. The $join_{\texttt{LocalsInfo}}(x, y)$ operator is defined as the $join_{\texttt{LocalInfo}}(x_l, y_l)$ operator applied to each entry l of the `LocalsInfo` mapping. We then need to join each component of the `LocalInfo` individually.

Joining `length_of`. If the `LocalInfo` does not represent a vector, then there is nothing to join. For `LocalInfo` that are vectors, the operator that will join the `length_of` field is simply the maximum and is trivially sound:

$$join_{\texttt{length\_of}}(x, y) = max(x, y)$$

From this definition we can infer the bottom and top values for `length_of`: $\bot_{\texttt{length\_of}} =$ `CostExpr::Scalar(0)` and $\top_{\texttt{length\_of}} =$ `CostExpr::Infinity`.

Joining `ty`. The second join operation, for the `ty` field, needs to take into account the history of the type precision.

$$join_{\texttt{ty}}(x, y) = longest\_common\_prefix(x, y)$$

The soundness argument for $join_{\texttt{ty}}$ is that if the two histories are the same there is nothing to join, and we keep the common type history if they diverge. The bottom element for `ty` is the MIR type at the start of the analysis, it is taken from the caller context if the `Local` is one of the callee function parameters or taken from the MIR body's `Locals` information otherwise. The top

element is not defined but also not needed since the tool does not support loops and recursion so the history has a finite length.

Joining `members`. Finally, the `members` are joined recursively one by one:

$$join_{\texttt{members}}(x, y) = join_{\texttt{LocalInfo}}(x_m, y_m) \quad \forall m \in \texttt{members}$$

The soundness of this last join operator is guaranteed since the soundness of $join_{\texttt{LocalInfo}}$ relies on the soundness of $join_{\texttt{length\_of}}$ and $join_{\texttt{ty}}$ which have been shown to be sound as well.

5.4.2 Counting domain

We want the pallet developer to be able to relate the tool's output with the inferred weights given by the FRAME's benchmarking module. To do so we will track the same four metrics:

- size of reads/writes on the database. Substrate actually implements a cache layer between the pallet's execution and the database changes, so that if a field is accessed twice, only one access is done on storage. We ignore this optimization during the analysis.
- size and variants of the emitted events
- algorithmic time complexity

This is why our abstract counting domain C consists of four cost expressions, accounting for database reads and writes, size of the deposited events and the number of steps:

$$C : (\texttt{CostExpr} \times \texttt{CostExpr} \times \texttt{CostExpr} \times \texttt{CostExpr})$$

We define the *join* operator:

$$join(x, y) = max_C(x, y) \quad | \quad x, y \in C$$

where max_C is the pointwise maximum which is used to join the summaries of two basic blocks during the intraprocedural analysis. When encountering function calls during the analysis we also need to account for the cost of the called function and this is done through the composition of individual cost of function call statements. In the concrete semantics the composition would be the addition operation, and the symbolic addition in the abstract semantics:

$$compose_{inter}(x, y) = add_C(x, y) \quad | \quad x, y \in C$$

where $compose_{inter}$ is the pointwise addition that is used to join two function summaries, after a function call analysis for example.

The bottom and top elements for C are trivially defined as:

$\bot_C = (\texttt{CostExpr::Scalar(0)}, \texttt{CostExpr::Scalar(0)}, \texttt{CostExpr::Scalar(0)}, \texttt{CostExpr::Scalar(0)})$

$\top_C = (\texttt{CostExpr::Infinity}, \texttt{CostExpr::Infinity}, \texttt{CostExpr::Infinity}, \texttt{CostExpr::Infinity})$

Here again the analysis is sound because we over-approximate the concrete domain by using the *max* operator during intra-procedural analysis and the *add* operator during the inter-procedural analysis. The analysis running on this domain is sound provided that the transfer functions, or transformers, are also sound. We will provide more details about transformers in Section 6.4.8.

6 Implementation

In this chapter, we will describe the implementation details, the choices we made as well as some technical challenges we overcame.

First of all, we chose to run the analysis on the pallet's level because we wanted to access information available in the HIR (High-level Intermediate Representation) and it is only generated for the currently compiled crate[35].

Secondly, the analysis is done on the generated optimized MIR. The choice of running the analysis on the MIR was motivated by the following aspects:

- since some optimizations were already made, we are closer to the final behaviour than with any other Rust intermediate representation,
- we carefully compared the different representations of the code, HIR, MIR, LLVM-IR and WebAssembly. While LLVM-IR and WebAssembly are the closest to the real execution, a lot of information is lost in the process because all the structures, field accesses or operations on vectors for example are abstracted away and the output would not be very informative for a developer. We also considered the HIR for the analysis since it is close to the initial logic. The problem is that it is only available for the top-level crate being compiled. This also means that type checking and inferring did not happen yet and we may lack information, e.g. the size of some types,
- we can still easily map the MIR to the Rust source code if we need to understand more precisely how a function works,
- the compiler also uses this intermediate representation to run some internal static analysis,
- type checking and inference were completed so we have more insight about the types we encounter,
- there already exists a dataflow analysis framework embedded in the compiler that we can use for static analysis.

The major drawback of using the MIR at the pallet level is that it is still a generic library whose concrete types are set when the runtime is constructed. It means that, as mentioned in Section 4.2, monomorphization cannot be done and we cannot use `Instance`[36], which would have allowed us to generate monomorphized MIR.

6.1 Cost Model

In this section, we show how the different operations encountered during the analysis are taken into account into the cost expression.

6.1.1 Precise modeling with the MIR

When the MIR is available for a function, the cost model for each `StatementKind::Assign(Place, Rvalue)` of a basic block is straightforward and we can assign a cost to the operations (`Rvalue`s) coming directly from the MIR, like addition, multiplication, division, and other binary operations which are grouped in `rustc_middle::mir::syntax::BinOp` enum[37], or the `Repeat(element, X)` operator which is responsible for filling an array `X` times with `element`. The cost expression associated to those basic statement operations are listed in Table 3, the description of the different `Rvalue`s are shown in Table 2. Function calls (`TerminatorKind::Call`) also have a cost of `Scalar(1)` to account for the call computational overhead.

6.1.2 Approximation with specifications

Due to missing monomorphization, the reasons are explained in Section 6, the MIR may not have been generated for some functions. If the MIR is not available for the function to be analyzed, we need to write manual specifications. This is where the transformer is not guaranteed to be sound. For some implementations, like storage access functions, it is possible to have sound and relatively

[35] `https://rustc-dev-guide.rust-lang.org/hir.html#out-of-band-storage-and-the-crate-type`
[36] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/ty/instance/struct.Instance.html`
[37] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/syntax/enum.BinOp.html`

Operation (Rvalue)	Cost
Use(Operand)	Scalar(1)
Repeat(Operand, times)	Scalar(times)
Ref(Region, BorrowKind, Place)	Scalar(1)
ThreadLocalRef(DefId)	Scalar(1)
AddressOf(Mutability, Place)	Scalar(1)
Len(Place)	Scalar(1)
Cast(CastKind, Operand, Ty)	Scalar(1)
BinaryOp(BinOp, (Operand, Operand))	Scalar(1)
CheckedBinaryOp(BinOp, (Operand, Operand))	Scalar(1)
NullaryOp(NullOp, Ty)	Scalar(1)
UnaryOp(UnOp, Operand)	Scalar(1)
Discriminant(Place)	Scalar(1)
Aggregate(AggregateKind, operands)	Scalar(operands.len())
ShallowInitBox(Operand, Ty)	Scalar(1)
CopyForDeref(Place)	Scalar(1)

Table 3: List of Rvalues their cost expression in our cost model.

precise specifications. But for other functions like std::ops::Mul::mul[38] for example, we do not have access to the concrete implementation and the actual cost heavily depends on it. For primitive types like usize, we could account for it as a constant cost, but for more exotic types like vectors or matrices, the cost of a multiplication relies on the concrete implementation. When such cases occurred, we decided to account for them in the manner they would happen the most, usually by a constant cost in the number of steps, hence the soundness caveats.

When specifying more complex functions, we tried to recreate their costs as best as possible. While it is not possible to be as precise as if we had the MIR, it is still possible to make some sound approximations, like for std::vec::Vec::<T, A>::push[39] where we account for a possible growth of the allocated memory region. The true implementation will double the allocated memory when the maximal capacity is reached, so each push to the vector will have an amortized cost complexity, but our over-approximation accounts for a complexity of $\mathcal{O}(L)$, where L is the current symbolic length of the vector.

6.2 Internal representation of the types

In order to compute some costs, e.g. number of bytes read/written, we also need the size of some types. We wrote an internal representation for such types, Type, to be able to easily query their size. They are built from ty::Ty, the representation of types in the MIR.

We added special support for algebraic data types (ADT) that are often used in Substrate, these are Option and BoundedVec. The latter is a vector along with a maximum size, which is often used in storage fields. Since we want the worst case, its size is computed from the maximum number of elements it can hold, multiplied by the size of one of these elements. For example, the storage field defined at line 24 in our example will have a size expressed as VALUEOF(MaxSize::get()) * 4. One clear drawback of using the upper bound size for the BoundedVec used by storage fields is the loss of precision compared to an analysis where we would have better modeling of the storage fields and consider the on-chain state as an additional parameter to the function currently being analyzed. Despite being not very precise, it still makes sense to consider the upper bound size since we over-approximate. This is due to the refund mechanism implemented by Substrate, the function's weights should consider the worst-case storage accesses and refund the users at the end of the dispatchable execution, so they only pay for what they used in the end.

The size of those types is expressed with the use of the Cost type, defined in Section 5.3, they are computed in bytes where the concrete size is known at compile time, otherwise, we keep their size symbolic.

6.3 Cost language

We now present the implementation of the cost language defined in Section 5.3, its implementation in Rust can be found in Listing 7. The smallest unit a Cost can have is either Scalar or

[38] https://doc.rust-lang.org/std/ops/trait.Mul.html#tymethod.mul
[39] https://doc.rust-lang.org/std/vec/struct.Vec.html#method.push

`CostParameter`. To preserve soundness, i.e. we are over-approximating, the cost `Scalar(1)` represents the highest constant cost encountered during the analysis. `Add`, `ScalarMul`, `ParameterMul` and `Max` are the operations we can use to build a more complex cost expression. `BigO` is used to abstract away the constants and express the asymptotic complexity.

Listing 7: Semantics of the cost language.

```
pub(crate) struct Variable {
    pub id: u32,
    pub span: Option<Span>,
}

pub(crate) enum Cost {
    Scalar(u64),
    Parameter(CostParameter),
    Add(Vec<Cost>),
    ScalarMul(u64, Box<Cost>),
    ParameterMul(CostParameter, Box<Cost>),
    Max(Vec<Cost>),
    BigO(Box<Cost>),
    Log(Box<Cost>),
}

pub(crate) enum CostParameter {
    ValueOf(String),
    SizeOf(String),
    ReadsOf(String),
    WritesOf(String),
    SizeDepositedOf(String),
    StepsOf(String),
    LengthOf(Variable),
}
```

Since some of the cost expressions can become quite cumbersome, we implemented the `+` and `max` operation such that they reduce the length and complexity of the expression.

For the `add` operation, we pattern match on the left-hand (`lhs`) and right-end (`rhs`) sides to check whether it is possible to merge them without adding a `Add` node as shown in the left part of Listing 9. The optimizations are:

- if either `lhs` or `rhs` is `Infinity`, we return `Infinity`
- if either `lhs` or `rhs` is zero, we return the other element
- the addition of two `Scalars`, `Scalar(a) + Scalar(b)` will be `Scalar(a+b)`
- `ScalarMul(x, a)` are aggregated:
  - $((x) * a) + a => ((x + 1) * a)$
  - $a + ((x) * a) => ((x + 1) * a)$
  - $((x) * a) + ((y) * a) => ((x + y) * a)$
- when adding a `Scalar` to an already existing `Add` node, the `Scalars` are aggregated
- otherwise, we pack `lhs` and `rhs` in a new `Add` node and we flatten the whole vector. Flattening the vector will open any nested `Add` node and expand the `Scalars` and `ScalarMuls`. Then, we extract common elements, and recursively aggregate them.

For the `max` operation we flatten and reduce each side and try to extract out of the `Max` node the elements with the same value on the left-hand and right-end sides as they will not play a role in deciding which side represents the biggest value. Then we compare the remaining elements and try to decide which is the max, otherwise we create a new `Max` node. An example is shown in the right part of Listing 9. Some of the optimizations are:

- zeros are ignored

- if either `lhs` or `rhs` is `Infinity`, we return `Infinity`
- if the element we want to add to the `Max` node is already present in the vector we do nothing since $max(a, a, b) = max(a, b)$
- `max(Max(v_1),Max(v_2))` will end up as maximum of the union of `v_1` and `v_2`
- whenever possible, we try to "resolve" the max, i.e. the common elements are extracted from the vector's members and will be taken out of the `Max` node. If the remaining elements are comparable, we take the biggest, otherwise we keep them in a smaller `Max` node, this is demonstrated in the right-hand side example of Fig. 9.

```
Example 1:
----------
Concrete(20)
    +
Concrete(22)
=> Concrete(41)
instead of
Add([
  Concrete(20),
  Concrete(2)
])

Example 2:
----------
ConcreteMul(
  4,
  Symbolic(
    SizeOf("someADT")
  )
)
    +
Symbolic(
  SizeOf("someADT")
)
=> ConcreteMul(
     5,
     Symbolic(
       SizeOf("someADT")
     )
   )
instead of
Add([
  Symbolic(
    SizeOf("someADT")
  ),
  ConcreteMul(
    4,
    Symbolic(
      SizeOf("someADT")
    )
  )
])
```

```
Example 1:
----------
max([
  Add([
    Concrete(42),
    Symbolic(
      SizeOf("ADT1")
    ),
    Symbolic(
      SizeOf("ADT2")
    )
  ]),
  Add([
    Concrete(42),
    Symbolic(
      SizeOf("ADT1")
    ),
    Symbolic(
      SizeOf("ADT3")
    )
  ])
])
=> Add([
     Concrete(42),
     Symbolic(
       SizeOf("ADT1")
     ),
     Max([
       Symbolic(
         SizeOf("ADT2")
       ),
       Symbolic(
         SizeOf("ADT3")
       )
     ])
   ])
```

Figure 9: Examples of simplification of the cost expression for addition (left) and `max` (right) operations

6.4 Pallet static resource analysis

The tool is open source and can be found on GitHub `https://github.com/simon-perriard/saft`. It is named `saft` after "Static Analyzer for FRAME palleTs" and can be run as a `cargo` subcommand[40] with `touch src/*.rs && cargo saft --release` from inside the pallet's crate, the `touch` command forces the pallet's recompilation. The tool will output the static resource analysis for each dispatchable of the pallet under analysis.

Note that `saft` does not support loops and recursion, yet. It will detect such patterns and mark the analysis of the dispatchable as invalid.

Although there is no automatic support for loops, users can still write manual specifications for the functions that contain such structures. They can simply add their specifications in the `try_custom_dispatch` function and the tool will apply this custom transformer instead of trying to analyze it, detect a loop and fail.

6.4.1 Hooking into the compiler

The `cargo` subcommand is mainly a wrapper around the command `RUSTC_WORKSPACE_WRAPPER="/path/to/saft-driver" RUSTCFLAGS="-Z always-encode-mir" cargo check --lib` with the remaining arguments generated by the `cargo saft --release` command. The first environment variable tells `cargo` to use `saft-driver` to compile the files that are in the workspace, the second one asks `rustc` to make the MIR available whenever possible for every dependencies.

The compilation of the crate and its dependencies is then handed over to the usual `cargo` pipeline and whenever it is the turn of our pallet's crate of interest, it is compiled with `saft-driver`. `saft-driver` is itself a wrapper around the Rust compiler. It will invoke and run a compiler instance provided by `rustc_driver::RunCompiler` with empty callback by default, and with our custom callback when the `lib.rs` implementing the pallet must be compiled.

As quickly explained in Section 4.1, `rustc_driver` allows us to hook into the compilation process at multiple stages. Since we are interested in the optimized MIR, we only used the last callback, `after_analysis`. We can then inject our `saft::extract_juice` in the callback.

At this point, our callback handler `saft::extract_juice` has access to all the memoized compilation queries, up to MIR optimization, and will start the analysis.

6.4.2 Gathering of pallet information

We now need to collect the following information about the pallet:

- the storage fields, their types and sizes
- the `DefId`s of the dispatchables to analyze

We first construct a list of the fields' names, for each storage field there is a `struct` named `_GeneratedPrefixForStorageFIELD_NAME` so we iterate through the HIR `Item`s[41] and we store the different `FIELD_NAME`s.

Secondly, we iterate again through HIR `Item`s to target the type definition where the storage field is declared, line 24 in our example, and we compare their names with the ones of the list from above. If the name matches, we then extract the type that is behind the type alias, i.e. we are interested in `StorageValue<_, BoundedVec<u32, T::MaxSize>, ValueQuery>;` to compute its size, not in the type alias `StoredNumbers<T: Config>`. From there we record the kind of storage field (`StorageValue, StorageMap,...`), the type alias `DefId`, the type of the values that are stored and type(s) of the key(s) if it is a kind of `Map`, both as one of our internal `Type` representation.

Finally, we need to collect the `DefId`s of the dispatchables. Like before, we start by getting a list of the dispatchables names. We look for the pallet's `Call` enumeration to retrieve the name of its variants, which are the dispatchables names. From Section 3.3 we know that the calls are dispatched to the target functions by `pallet::Call::dispatch_bypass_filter`, so the task is now to symbolically execute the HIR body of that dispatch function for each of the dispatchable name until we reach the call to the target dispatchable, from where we can get its `DefId`.

We now have our own representation of the pallet with the information that will be useful for the analysis, i.e. what are the storage fields and what are the dispatchables.

[40] `https://doc.rust-lang.org/book/ch14-05-extending-cargo.html`
[41] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_hir/hir/struct.Item.html`

6.4.3 MIR's data flow analysis

During compilation, the compiler already performs some data flow analysis [23] (the code that is used to this end can be found in `rustc_mir_dataflow`), which makes it easy to run an analysis, provided that we define a correct domain and transfer function. We will leverage this module as it is included in the compiler and provides a good framework for our analysis.

Data flow analysis has been developed with the concepts of abstract interpretation in mind. We first need to design an abstract domain that will represent the problem we want to solve, as well as a transfer function. The analysis works by following the CFG, in a forward or backward manner. The algorithm visits each basic block and applies the transfer function to each statement and terminator in order. If a basic block has multiple predecessors, the transformed output domains of those predecessors are joined ($\sqcup$) to create the input domain for the basic block to be visited. The analysis iterates through the CFG until a fixpoint is reached. If the program contains loops or recursion, termination with an unbounded complete lattice and monotonic transfer function is not ensured. This is the reason why the abstract domain needs to be a join-semilattice, i.e. the domain must implement `JoinSemiLattice`[42], so the analysis may reach a fixpoint at some point, $f(\top) = \top$ at worse.

The data flow analysis implemented in the compiler uses a less naive algorithm, which is called maximal fixpoint algorithm (MFP). It is similar to the one described above but tries to be smarter about it. Instead of going through the whole set of successors of a block when its input have changed, it keeps a list, called worklist, of the edges where data has changed. Let us consider a basic block with successor(s), if for two different inputs the transfer function outputs the same domain, the analysis will not revisit the successors since their input domain would not change.

To create a new analysis with this framework, it suffices to define a struct containing all the information we will need during the process, the struct must then implement the following traits: `Analysis`[43] and `AnalysisDomain`[44]. Here we specify the actual domain we want to use, with the condition that it must be a partially ordered set that has a least upper bound property for any arbitrary pair of elements of the set, in other words a join-semilattice. Several such domains are made available by the compiler, but it is also possible to build custom ones. We also need to implement the mandatory functions that will be called during the MIR analysis: `apply_statement_effect`, `apply_terminator_effect` and `apply_call_return_effect`, basically this is where we will call the transfer function. Then we call the provided functions `into_engine`, `iterate_to_fixpoint` and `into_result_cursor`, resp. to instantiate the engine capable of solving the dataflow problem, run the analysis until it stabilizes, and access the results of the analysis.

6.4.4 Analysis of closures

Closures[45] are anonymous functions that can capture their environment in so-called `upvars`. They are executed through the use of a data structure containing a pointer to the anonymous function body and the `upvars` that are passed along the function's parameters packed in a tuple to one of the following: `FnOnce::call_once`, `FnMut::call_mut` or `Fn::call`, which are responsible for calling and executing the closure with the correct parameters.

In the example snippet Listing 8, the closure stored in `c` will capture the `captured` variable from its environment and this translates in its MIR assigning the `captured` variable in a similar manner than we would access a field of a `struct` using a projection, see lines 51 and 19 of Listing 9. We can consider `[closure@src/main.rs:9:13: 12:6]` as an ADT carrying the environment of the closure it is bound to. Still in Listing 9, when the closure is executed at line 10, this ADT and the argument(s) of the closure are passed to `call_once` that will call and execute the closure's code, starting at line 17.

Note that `closure_caller` is not yet aware of the concrete type of its argument, this will be inferred later during monomorphization, a similar function with the concrete type `[closure@src/main.rs:9:13: 12:6]` will be generated. So if we want to analyze this closure in good conditions at its call site we need to pass the calling context from `main` to `closure_code` when we analyze it. We explain how the tools tracks the calling context in the next section.

[42] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_mir_dataflow/lattice/trait.JoinSemiLattice.html`

[43] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_mir_dataflow/framework/trait.Analysis.html`

[44] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_mir_dataflow/framework/trait.AnalysisDomain.html`

[45] `https://doc.rust-lang.org/book/ch13-01-closures.html`

Listing 8: The closure stored in `c` captures a value from its environment and will multiply it by a factor provided only when the closure is called.

```
fn closure_caller<F: FnOnce(usize) -> ()>(closure: F) {
    closure(6);
}

fn main() {

    let captured = 7;

    let c = |mul_factor| {
        let res = captured * mul_factor;
    };

    closure_caller(c);
}
```

Listing 9: Generated `MIR` for Listing 8, the live ranges (`StorageLive/StorageDead`) of the `Locals` have been removed for readability.

```
fn closure_caller(_1: F) -> () {
    debug closure => _1;
    let mut _0: ();
    let _2: ();
    let mut _3: F;
    let mut _4: (usize,);
    bb0: {
        _3 = move _1;
        (_4.0: usize) = const 6_usize;
        _2 = <F as FnOnce<(usize,)>>::call_once(move _3, move _4) -> bb1;
    }
    bb1: {
        return;
    }
}

fn main::{closure#0}(_1: &[closure@src/main.rs:9:13: 12:6], _2: usize) -> () {
    debug mul_factor => _2;
    debug captured => (*((*_1).0: &usize));
    let mut _0: ();
    let _3: usize;
    let mut _4: usize;
    let mut _5: usize;
    scope 1 {
        debug res => _3;
    }
    bb0: {
        _4 = (*((*_1).0: &usize));
        _5 = _2;
        _3 = Mul(move _4, move _5);
        return;
    }
}

fn main() -> () {
    let mut _0: ();
    let _1: usize;
    let mut _3: &usize;
    let _4: ();
    let mut _5: [closure@src/main.rs:9:13: 12:6];
    scope 1 {
        debug captured => _1;
        let _2: [closure@src/main.rs:9:13: 12:6];
        scope 2 {
            debug c => _2;
        }
    }
    bb0: {
        _1 = const 7_usize;
        _3 = &_1;
        (_2.0: &usize) = move _3;
        _5 = _2;
        _4 = closure_caller::<[closure@src/main.rs:9:13: 12:6]>(move _5) -> bb1;
    }
    bb1: {
        return;
    }
}
```

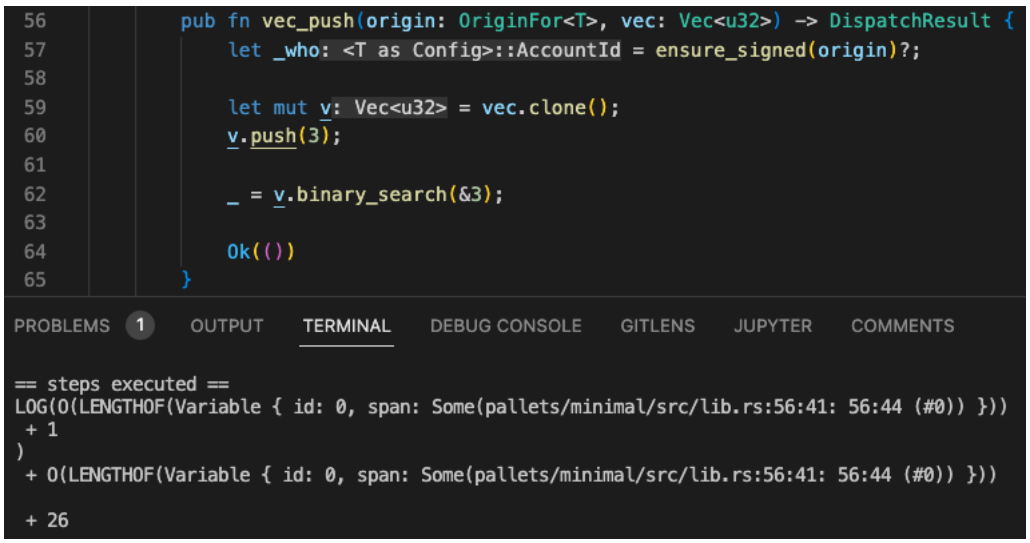

Figure 10: Thanks to the `length_of` field, the cost expression can precisely refer to a symbolic length.

6.4.5 Tracking the calling context

As mentioned before, since monomorphization did not happen yet and we are doing interprocedural analysis, we need to pass the calling context from the caller to the callee function we want to analyze. To do so, we maintain a mapping `LocalsInfo` : `Local` → `LocalInfo` and we define a new recursive struct `LocalInfo`, see Listing 10, which represents the most precise type information we can have for a `Local` at a given point in the analysis. Our internal representation of type information for a `Local` has the ability to represent simple types by using only `ty`, but also more complex types like `struct` or `enum` thanks to its `members`. If it is a dynamically sized vector, its length can be shared through its references by means of the `Rc<RefCell<_>>`, a shared reference-counted pointer to a location in the heap.

For simplicity, the pointers, references, as well as mutability are abstracted away in `LocalInfo` and we only keep the underlying type. This poses no issue for the soundness of the analysis because the elements listed above are not relevant for the types, but only for their concrete values. One exception to this are the sizes of the dynamically sized vectors we keep track of. The base vector and its references share the same reference to its size, so when one of them gets updated, the change is reflected everywhere for this vector. If the vector is cloned, its length will be copied too, but in a new `Rc<RefCell<_>>`, so the new vector is now living independently from its source vector. `LocalInfo` considers those vectors and references as mutable but this is sound thanks to the fragment of the Rust we support, which will ensure that two mutable references to the same object cannot exist at the same time. This is particularly useful when the complexity depends on the length of some vector: we can point to which vector's length we refer to, see Fig 10 for an example, where the cost expression does not simply point to the length of a variable that has the type `Vec`, we can precisely track it down to be related to the function's argument's length.

Whenever the analysis encounters a function call, we create a new `CalleeInfo` and we will inject the `LocalInfo`s we have from the current caller context's `LocalsInfo` as the type arguments of the callee function. The callee function analysis will first fill its own `LocalInfo` mapping as described above, then the `Local`s representing the function's arguments will be overridden by the information coming from the caller. We can then analyze the called functions with the most precise calling context possible. Then, to update the type information, we apply symbolic store, i.e. if considering Rust's MIR basic operation, every time we encounter a `StatementKind::Assign`[46] that has a `Use` or `Ref` as an `Rvalue`, we update the type information of the left-hand side operand with the one of the right-hand side operand.

This technique is limited to the calling context for the arguments of a function and does not replace monomorphization. It cannot infer concrete types for generic traits or functions and does

[46] `https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.StatementKind.html#variant.Assign`

not allow the tool to generate MIR for such functions.

Listing 10: `LocalInfo` encapsulates the most precise type and size information the analysis can get at a given point in the program.

```
pub(crate) struct LocalInfo<'tcx> {
    pub length_of: Rc<RefCell<Option<Cost>>>,
    ty: Vec<Ty<'tcx>>,
    members: Vec<LocalInfo<'tcx>>,
}
```

6.4.6 Wrapping the calling context

Since we aim to provide more precise results by leveraging the interprocedural analysis, we need to extract the calling context out of each function's call site so that the tool can pass it to the transformer responsible to analyze the call. We encapsulate all the relevant information in a `struct` called `CalleeInfo`, see Listing 11.

Listing 11: `CalleeInfo` captures information about the caller's context at the `callee_def_id` call site to pass it to the callee analysis.

```
pub(crate) struct CalleeInfo<'tcx> {
    pub location: Option<Location>,
    pub args_type_info: Vec<LocalInfo<'tcx>>,
    pub caller_args_operands: Option<Vec<Operand<'tcx>>>,
    pub destination: Option<Place<'tcx>>,
    pub callee_def_id: DefId,
    pub substs_ref: SubstsRef<'tcx>,
}
```

The location points to the call site of the callee in the caller's MIR. Arguments type info is an ordered list of `LocalInfo` copied from the caller, one per function argument, this is maybe the most important element of the calling context. From `caller_args_operands` it is possible to retrieve the caller's `Places` that were used to call the function, the destination points to the caller's `Place` where the result of the call will be stored.

The optional fields are needed in cases where a closure cannot be analyzed directly at its call site and we need to simulate the call out-of-context, more details will be given in Section 6.4.8.

6.4.7 Event variants analysis

We implement here the abstract domain defined earlier in Section 5.2. As shortly stated before, the goal of this first pre-analysis is to collect information about which variant, also called discriminant, of the `Event` enum is called at which `deposit_event` call site. It is needed because we cannot know when visiting the call site which variant will be deposited. From our study of a subset of pallets, the events are usually instantiated in the same function where `deposit_event` is called and are not passed as a parameter to any other function, so we considered that an intraprocedural analysis should be sufficient to cover the vast majority of cases.

We implemented the `Variants` set as shown in Listing 12. The domain E is implemented as a `Hashmap<Location, Variants>` and the $join(x, y)$ operator will check whether $x[k]$ is defined $\forall k \in y.keys$. If it is not defined, it will add $y[k]$ as is, and it will wrap and insert it as $Or(x[k], y[k])$ otherwise.

Listing 12: The `Variants` enum captures the fact that the variant of the deposited event could be decided by a branching structure.

```
pub(crate) enum Variants {
    Variant(VariantIdx),
    Or(Box<Variants>, Box<Variants>),
}
```

During the analysis, we keep track of a mapping `Local` $\rightarrow$ `VariantIdx` that gets updated

whenever a statement is of kind `StatementKind::SetDiscriminant`[47], in a symbolic-store-like style. The domain gets updated when the analysis encounters a call to `deposit_event` in a block terminator.

We keep the analysis summary (E) of each function in a mapping `DefId` $\rightarrow E$ as we will use it in the cost analysis.

6.4.8 Cost analysis

Now that we have collected all the needed information about storage fields, dispatchables and event variants, it is time to start the cost analysis. We perform intra- and inter- procedural analysis, with calling context passing.

We define our `CostAnalysis`, Listing 13, which is fed with the information we gathered during the pallet information gathering and event variants analysis phases. It implements the `rustc_mir_dataflow`'s `Analysis` and `AnalysisDomain`, where the domain is the abstract counting domain defined in Section 5.4.2.

Listing 13: The `CostAnalysis` data structure contains all the data and fields needed for the analysis of a dispatchable.

```
pub(crate) struct CostAnalysis<'tcx, 'inter> {
    tcx: TyCtxt<'tcx>,
    pallet: &'inter Pallet,
    events_variants: &'inter HashMap<DefId, EventVariantsDomain>,
    def_id: DefId,
    caller_context_args_type_info: Vec<LocalInfo<'tcx>>,
    pub analysis_success_state: Rc<RefCell<AnalysisState>>,
    fresh_var_id: FreshIdProvider,
}
```

We had to handle multiple situations when considering the analysis of the MIR. The simplest cases are where we can directly infer the cost from the MIR statements, see Table 3, then there are function calls which need to be accounted for. We want to analyze a function at its call site, denoted by a `TerminatorKind::Call` MIR node, so we can provide it with its calling context. Function call analysis is done following the pattern showed below:

1. we wrote specifications for the function,

2. we check whether some MIR is available for the function or not, to check for this property we used the provided method `TyCtxt::is_mir_available`[48]. It is then possible to get the MIR body through the API call to `TyCtxt::optimized_mir`[49].

3. if none of the above applied, the tool will raise an error informing the user that no specifications nor MIR was found, along with some calling context information.

Accounting for functions with specifications . For functions whose MIR is unavailable or functions we don't want the tool to analyze by itself, like some standard library functions, generic functions, storage access functions, or some custom functions containing loops, we dispatch the analysis to manual specifications we wrote. We emphasize that the manual specifications may introduce some soundness issues depending on the operation that had to be manually specified, as explained in Section 6.1. Some of those functions may contain side effects that need to be considered when writing the specifications, like `StorageValue::try_mutate` (see Listing 14), which, in addition to its own cost execution, calls a closure. This is why a careful inspection of every function requiring manual specifications is needed in order to recreate its costs and effects as precisely as possible. The list of manual specifications we implemented is displayed in Appendix A.

[47] https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.StatementKind.html#variant.SetDiscriminant

[48] https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/ty/context/struct.TyCtxt.html#method.is_mir_available

[49] https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/ty/context/struct.TyCtxt.html#method.optimized_mir

Accounting for functions with available MIR. Such functions are analyzed at their call site with their calling context taken from the current caller's `LocalsInfo`. We spawn a new instance of `CostAnalysis` where we inject the calling context for the function parameters we get from the corresponding `CalleeInfo` and we run the analysis. Once the analysis of the called function reaches a fixpoint and terminates, the result of the analysis $\in C$ is added ($compose_{inter}$) to the current abstract domain and the `length_of` of the `LocalInfo` corresponding to the return value of the callee overwrites the one from the caller's `destination`'s `Place`.

Accounting for closures cost. Closures costs can happen in two different ways:

1. closure can be analyzed at its call site: during the analysis of MIR's terminator we can encounter calls to either:
   `{FnOnce::call_once, FnMut::call_mut or Fn::call}(closure_struct,packed_args)`

   In such cases, the tool will handle the closure's execution itself, where we will simulate the action of the functions listed above. We create a new `CalleeInfo` to mimic the calling context of the closure where we extract the function pointer and unpack the arguments. Then we run the analysis on the closure's body to compute its cost. Here we can replace the `location` and `destination` by the ones of the call to one of the current calling context. In other words, we treat the call as if it was a direct function call to the closure's body and abstract away the call to `FnOnce::call_once`, `FnMut::call_mut` or `Fn::call`.

2. closure is analyzed from the specifications: it may happen that we are aware that a closure has to be executed in some specifications we are writing, and it can be that the closure will be executed multiple times, think of a mapping or filtering of a vector for example. When such a case arises, the specifications must run the closure's analysis internally to get its cost. To this end, a calling context `CalleeInfo` must be created in the specifications, see Listing 14 for an example. The last argument of `transfer_function.fn_call_analysis`, a boolean variable used, can be set to indicate whether the analysis must be run in isolation, i.e. whether the cost of the closure must be accounted for at the end of its analysis (`run_in_isolation=false`) or not (`run_in_isolation=true`). This is useful if we only need the standalone cost of the closure, it then allows the tool to extract and modify it before adding the updated cost to the analysis domain. E.g. when we need to multiply the closure's cost with the length of a vector for a mapping operation.

Example of manual specifications. Listing 14 shows the specifications for the `StorageValue::try_mutate` storage access function. This function will query the storage field, apply a closure and write back the modified data. In our specifications we first account for the algorithmic complexity of the function, which will be proportional to the size of what is stored. The cost mainly comes from the decoding and encoding of the data. Then we add the cost of reading the database, mimic a calling context for the closure and run the analysis on it. Next we account for the cost of the closure and finally, we add the cost of writing the field back to the database.

Listing 14: Manual specification for `StorageValue::try_mutate`. A less technical explanation of the actions of this transformer is available in Section 6.4.8.

```
...
"frame_support::pallet_prelude::StorageValue::<Prefix, Value, QueryKind,
    OnEmpty>::try_mutate" => {

    // decoding/encoding depends on the actual length of what is stored
    transfer_function
    .state
    .add_steps(cost_to_big_o(storage_field.get_size(transfer_function.tcx)));

    // storage access
    transfer_function
        .state
        .add_reads(storage_field.get_size(transfer_function.tcx));

    // Account for closure call
    let closure_adt = callee_info.args_type_info[0].clone();
    let (closure_fn_ptr, closure_substs_ref) =
        if let TyKind::Closure(def_id, substs_ref) = closure_adt.get_ty().kind
    () {
            (*def_id, substs_ref)
        } else if let TyKind::FnDef(def_id, substs_ref) = closure_adt.get_ty().
    kind() {
            (*def_id, substs_ref)
        } else {
            unreachable!();
        };

    // Closure accepts only one argument which is of type Value
    // Rust was able to infer the type in the closure's substs ref
    // so no need to specialize more the args_type_info
    let closure_call_simulation = CalleeInfo {
        location: None,
        args_type_info: Vec::new(),
        caller_args_operands: None,
        destination: None,
        callee_def_id: closure_fn_ptr,
        substs_ref: closure_substs_ref,
    };

    transfer_function.fn_call_analysis(closure_call_simulation, false);

    // storage access
    transfer_function
        .state
        .add_writes(storage_field.get_size(transfer_function.tcx));
    Some((*transfer_function.state).clone())
}
...
```

If no specification nor MIR is available for a function, the tool will mark the analysis as failed and will output the name of the function that could not be analyzed, along other information like the `LocalInfo` of its arguments to help the user write new manual specifications.

Finally, the analysis summary for a given dispatchable is displayed in the terminal as shown in Fig. 11. In case of an unsuccessful analysis, either it met a loop or a recursion, the output will inform the user about which of those issues it encountered and what is the function responsible for it. Note that it may happen that some `O(SIZEOF(symbolic_struct_name))` appear in the final expression, this can be explained by the fact that some operations are done on `structs` whose

size is only known symbolically, examples of such use cases are Substrate's encoding/decoding of a `struct`.

```
Extracting pallet information... Done
Extracting event variants... Done
The following dispatchables will be analyzed :
add_number

*******************************
Summary for dispatchable pallet::Pallet::<T>::add_number
== bytes read ==
VALUEOF(MaxSize::get()) * 4

== bytes written ==
VALUEOF(MaxSize::get()) * 4

== bytes deposited ==
SIZEOF(pallet::Event::NewNumberStored)

== steps executed ==
O(VALUEOF(MaxSize::get()))
 + 52

*******************************
```

Figure 11: The tool's output after analysis of the pallet defined in Listing 4.

7 Results

Tests setup:

- Rust toolchain: `nightly-2022-06-10`
- Substrate:
  `https://github.com/paritytech/substrate/tree/64d77551cb9ae0a9572a955d8a678d0de41af4bf`

7.1 Results from initial goal

We first tested `saft` on 5 pallets pointed to us by the Web3 Foundation, those are:

- `balances` pallet: manage the system's default currency, allows to transfer tokens for example.
- `identity` pallet: naming system managed by so-called `Registrars`, users can ask for `Registrars` to verify their identity, provided some fees.
- `multisig` pallet: submodule that can accept actions, as `Extrinsincs`, that need to be approved by a `threshold` number of signers to be dispatched.
- `utility` pallet: helper pallet enabling users to dispatch calls in batches or dispatch a call with an origin that is derived from the original user address.
- `vesting` pallet: this module provides a way to implement a vesting mechanism following a linear curve

We have run benchmarks of the tool for those pallets, see Table 4. The benchmark consisted of 100 runs of cost analysis on each dispatchable and measured the time it took. Then we averaged those timing over the 100 runs to have a representative time for each dispatchable.

Fig. 11 shows that for simple dispatchables we can easily relate the code with the tool's output. This is good but in fact, we are interested in the relation between the Substrate's weight benchmarking output and the tool's output. We can compare the weights specifications of `pallet::Identity::add_registrar` with the tool's output for the same dispatchable in Fig. 12.

We can observe that compared to `saft` output which gives bounds on the number of bytes that are modified on storage as well as the general algorithmic complexity, the benchmark module only considers the number of database accesses and number of deposited events, not their sizes.

However, the two tools are not in a competition and they need to be working together so that the pallet developers can automatically get some concrete weights on one side, and check that those complexities make sense compared to the worst case on the other side.

7.2 Results from extended goal

After having developed the tool with primarily the five pallets of Section 7.1, we considered that it was interesting to know how the tool would scale with more pallets, for which we did not write any targeted manual specifications. The results of this benchmark is shown in Table 5.

The tool is able to analyze a good quarter of the dispatchables contained in the pallets listed above without adding more specifications. From these results, one can say that there are still quite a lot of missing specifications but since they can be reused across different analysis, the burden of writing specifications decreases exponentially, i.e. one added specifications can unlock the analysis of multiple dispatchables.

The different source code for the pallets can be found on the Substrate's GitHub repository : `https://github.com/paritytech/substrate/tree/master/frame`.

Analysis results on other pallets of the FRAME library	
Dispatchable	Avg time over 100 runs or reason for analysis failure
`alliance::add_unscrupulous_items`	loop detected
`alliance::announce`	8ms
`alliance::init_members`	loop detected
`alliance::remove_announcement`	12ms

alliance::remove_unscrupulous_items	loop detected
alliance::set_rule	2ms
other 9 dispatchables of the alliance pallet	missing specs
assets::clear_metadata	9ms
assets::create	16ms
assets::destroy	loop detected
assets::force_asset_status	22ms
assets::force_clear_metadata	7ms
assets::force_create	5ms
assets::force_set_metadata	22ms
assets::freeze_assets	4ms
assets::set_metadata	50ms
assets::set_team	12ms
assets::thaw_asset	4ms
assets::transfer_ownership	16ms
other 13 dispatchables of the assets pallet	missing specs
all 3 dispatchables of the atomic-swap pallet	missing specs
babe::plan_config_change	1ms
other 2 dispatchables of the babe pallet	missing specs
all 2 dispatchables of the bags-list pallet	missing specs
bounties::accept_curator	14ms
bounties::approve_bounty	9ms
bounties::extend_bounty_expiry	9ms
bounties::propose_bounty	21ms
bounties::propose_curator	11ms
other 4 dispatchables of the bounties pallet	missing specs
all 7 dispatchables of the child-bounties pallet	missing specs
contracts::remove_code	6ms
other 5 dispatchables of the countries pallet	missing specs
all 6 dispatchables of the conviction-voting pallet	missing specs
democracy::cancel_proposal	loop detected
democracy::cancel_referendum	<1ms
democracy::clear_public_proposals	<1ms
democracy::emergency_cancel	7ms
democracy::external_propose	3ms
democracy::external_propose_default	<1ms
democracy::external_propose_majority	<1ms
democracy::fast_track	20ms
democracy::propose	20ms
democracy::veto_external	20ms
other 15 dispatchables of the democracy pallet	missing specs
election-provider-multi-phase::set_emergency_election_result	4ms
election-provider-multi-phase::set_minimum_untrusted_score	<1ms
other 3 dispatchables of the election-provider-multi-phase pallet	missing specs
elections-phragmen::elections-phragmen	2ms
other 5 dispatchables of the elections-phragmen pallet	missing specs
grandpa::note_stalled	<1ms
other 2 dispatchables of the grandpa pallet	missing specs
im-online::heartbeat	missing specs
indices::claim	4ms
indices::force_transfer	1ms
indices::free	7ms
indices::freeze	8ms
indices::transfer	15ms
lottery::set_calls	loop detected
lottery::stop_repeat	<1ms
other 2 dispatchables of the lottery pallet	missing specs
all 7 dispatchables of the membership pallet	missing specs
nicks::clear_name	3ms
nicks::force_name	8ms

nicks::kill_name	missing specs
nicks::set_name	22ms
node-authorization::claim_node	7ms
node-authorization::transfer_node	15ms
other 7 dispatchables of the node-authorization pallet	missing specs
all 13 dispatchables of the nomination-pools pallet	missing specs
preimage::unnote_preimage	11ms
other 3 dispatchables of the preimage pallet	missing specs
recovery::set_recovered	1ms
other 8 dispatchables of the recovery pallet	missing specs
remark::store	missing specs
all 6 dispatchables of the scheduler pallet	missing specs
scored-pool::change_member_count	<1ms
scored-pool::submit_candidacy	7ms
other 3 dispatchables of the scored-pool pallet	missing specs
session::purge_keys	loop detected
session::set_keys	missing specs
state-trie-migration::continue_migrate	missing specs
state-trie-migration::control_auto_migration	<1ms
state-trie-migration::force_set_progress	2ms
state-trie-migration::set_signed_max_limits	<1ms
other 2 dispatchables of the state-trie-migration pallet	loop detected
sudo::set_key	7ms
sudo::sudo	15ms
sudo::sudo_as	21ms
sudo::sudo_unchecked_weight	15ms
system::fill_block	1ms
system::remark	1ms
system::kill_storage	loop detected
system::set_storage	loop detected
other 5 dispatchables of the system pallet	missing specs
tips::retract_tip	7ms
other 5 dispatchables of the tips pallet	missing specs
all 3 dispatchables of the transaction-storage pallet	missing specs
treasury::approve_proposal	2ms
treasury::propose_send	7ms
treasury::spend	7ms
other 2 dispatchables of the treasury pallet	missing specs
all 4 dispatchables of the whitelist pallet	missing specs

Table 5: Results of the tool's benchmarks on some of the other pallets of the FRAME library, the timing average is done over 100 rounds.

Analysis results on the initial 5 pallets	
Dispatchable	Avg time over 100 runs or reason for analysis failure
balances::force_transfer	9ms
balances::force_unreserve	1ms
balances::set_balance	22ms
balances::transfer	6ms
balances::transfer_all	6ms
balances::transfer_keep_alive	6ms
identity::add_registrar	8ms
identity::add_sub	18ms
identity::cancel_request	76ms
identity::clear_identity	loop detected
identity::kill_identity	loop detected
identity::provide_judgment	173ms
identity::quit_sub	8ms
identity::remove_sub	19ms
identity::rename_sub	9ms
identity::request_judgement	166ms
identity::set_account_id	11ms
identity::set_fee	8ms
identity::set_fields	9ms
identity::set_identity	109ms
identity::set_subs	loop detected
multisig::approve_as_multi	555ms
multisig::as_multi	553ms
multisig::as_multi_threshold_1	30ms
multisig::cancel_as_multi	41ms
utility::as_derivative	15ms
utility::batch	loop detected
utility::batch_all	loop detected
utility::dispatch_as	15ms
utility::force_batch	loop detected
vesting::force_vested_transfer	31ms
vesting::merge_schedules	55ms
vesting::vest	48ms
vesting::vested_other	50ms
vesting::vested_transfer	30ms

Table 4: Results of the tool's benchmarks on the 5 initial goal pallets, the timing average is done over 100 rounds.

```
# <weight>
- `O(R)` where `R` registrar-count (governance-bounded and code-bounded).
- One storage mutation (codec `O(R)`).
- One event.
# </weight>
```

```
********************************
Summary for dispatchable pallet::Pallet::<T>::add_registrar
== bytes read ==
VALUEOF(MaxRegistrars::get()) * SIZEOF(RegistrarInfo)

== bytes written ==
VALUEOF(MaxRegistrars::get()) * SIZEOF(RegistrarInfo)

== bytes deposited ==
SIZEOF(pallet::Event::RegistrarAdded)

== steps executed ==
O(VALUEOF(MaxRegistrars::get()))
 + 72

********************************
```

Figure 12: Weights complexity computed by the benchmark module and the tool's output for the dispatchable `pallet::Identity::add_registrar`.

8 Conclusion

8.1 Objectives

We presented `saft`, the first static analysis tool in the Substrate ecosystem, dedicated to the extraction of the worst case execution resource consumption out of arbitrary pallets. The tool uses a semi-formal approach to tackle the problem, we automated the process with symbolic execution and abstract interpretation as much as possible but some manual specifications were needed at some point. The tool starts by building a minimal internal representation of the pallet and then uses a combination of symbolic execution and abstract interpretation to build some cost expressions that will represent the number of bytes that are read/written from/to the storage, the number of bytes deposited by the event emitted by the dispatchable execution, as well as the algorithmic worst case complexity of the execution.

It provides a useful side information to the developers so they can check that the weights they assigned to their dispatchables take the worst case into account, and help them fine-tune their benchmark if not.

Moreover, if a motivated developer finds any use for another static analysis The tool can also be used as a base framework for implementing other kinds of pallet-level program analysis.

8.2 Discussion

In this section, we come back on some implementation choices we made.

Monomorphization. The initial choice that we made of running the analysis on the pallet level was a good start for grasping the complexity of both Rust MIR and the Substrate architecture. However, we stumbled on the fact that when the MIR is still generic, on the Runtime in our case, monomorphization does not happen and some of the MIR cannot be made available. This lead us to the need for manual specifications and with it, a loss of precision and sometimes dangling soundness guarantees.

Even with those imperfections in mind, we were usually able to relate the tool's output with the weights benchmarking provided by Substrate, so the analyzer is still able to capture the complexity of the different dispatchables in a meaningful manner and provide useful input to the developers

to help them fine-tune their benchmarks.

References abstraction. As mentioned in 6.4.5, references and pointers are replaced by their underlying type for simplicity. This is fine in the context of our simple analysis but does not reflect the concrete semantics of Rust. This would be problematic for more complex and precise analysis. Ideally, the types and memory models should be as close as possible to the concrete semantics, but due to the limited 6 months span of this project, we decided to stick to a simpler type modeling.

Loops and recursion support. Fully automatic loop and recursion support was not implemented in the tool, mainly because of time constraints, but also because a lot of the dispatchables from the five initially supported pallets do not use such control flow structures. We wanted to put the main effort into providing a meaningful analysis that would work with the majority of the pallets before adding more complex features to the tool, like loop invariants resolution. However, partial support for loops is available through the use of manual specifications, where the user can set a rather precise bound from the context.

Results. The initial goal of achieving the analysis of the five listed pallets in Section 7.1 was reached. We decided not to write specifications for the 6 dispatchables still containing a loop because it was detected in the entry function, so the result of the analysis would be directly the specifications, which does not make much sense for (semi-)automatic analysis.

8.3 Future Development

We strongly believe that with more development, the tool can really contribute to the ecosystem. The first development we have in mind would be to achieve more precise bounds, with a lesser need for manual specification. This can be done through Runtime level analysis, with more symbolic execution, or another way to allow the compiler to enable full monomorphization of the MIR. Another optimization would be to consider the on-chain state as a function input parameter as well, and use symbolic execution to track the changes.

In a second step, the tool could implement a generic support for loops and recursion. We can imagine that it could allow the users to annotate the source code in order to help the analysis resolve loops invariants.

Finally, more research can be done on the tool and the development of a formal execution cost model and try to statically infer the weights, without using benchmarking. This could greatly benefit the ecosystem by removing the burden of assigning weights to the developer, by relying on a formal cost model.

Personal challenges. During the 6 months of this thesis I ran into multiple challenges of all sorts. The biggest ones were to get a full picture of the extrinsics execution path and to get familiar with the compiler. Substrate's architecture is quite complex and heavily relies on macros and polymorphic function calls that are not always trivial to follow. Of course, I could not explore in details the whole compiler, but the MIR part is already big enough to be a challenge on its own. A lot of time has been put into understanding the compiler's API and the different data structures that are used in the MIR in order to efficiently leverage them in the analysis.

On the more technical side, some other challenges were the cost language optimization, in particular the optimization for the `Max` operation where we wanted to "resolve" it as much as possible to lower the complexity of the final output. The implementation of the types and calling contexts tracking was not trivial either as the underlying structure had to mimic the structure of the types and the transformers had to be carefully designed. A last example of a challenge I encountered was of course Rust itself, whose limitations references and lifetimes can give a hard time if we are not careful.

I learned a lot during this thesis, it was a very interesting project where I had to discover and understand different parts (Rust, Substrate, static analysis, ...) and make them work together. I am glad to have reached the state in which the project currently is, since we can put the choices we made in perspective and point to solutions to improve them. Moreover, the current state allows interesting developments in the future.

References

[1] Ben Wegbreit. "Mechanical Program Analysis". In: *Commun. ACM* 18.9 (Sept. 1975), pp. 528–539. ISSN: 0001-0782. DOI: 10.1145/361002.361016. URL: https://doi.org/10.1145/361002.361016.

[2] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[3] Mads Rosendahl. "Automatic Complexity Analysis". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 144–156. ISBN: 0897913280. DOI: 10.1145/99370.99381. URL: https://doi.org/10.1145/99370.99381.

[4] C. Ferdinand. "Worst case execution time prediction by static program analysis". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004, pp. 125–. DOI: 10.1109/IPDPS.2004.1303088.

[5] Pedro B. Vasconcelos and Kevin Hammond. "Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs". In: *Implementation of Functional Languages*. Berlin, Heidelberg: Springer, 2005, pp. 86–101. ISBN: 978-3-540-27861-0. DOI: 10.1007/978-3-540-27861-0_6. URL: https://doi.org/10.1007/978-3-540-27861-0_6.

[6] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. "SPEED: Precise and Efficient Static Estimation of Program Computational Complexity". In: POPL '09 (2009), pp. 127–139. DOI: 10.1145/1480881.1480898. URL: https://doi.org/10.1145/1480881.1480898.

[7] Azadeh Farzan and Zachary Kincaid. "Compositional recurrence analysis". In: *2015 Formal Methods in Computer-Aided Design (FMCAD)*. 2015, pp. 57–64. DOI: 10.1109/FMCAD.2015.7542253.

[8] Zachary Kincaid et al. "Compositional Recurrence Analysis Revisited". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 248–262. ISBN: 9781450349888. DOI: 10.1145/3062341.3062373. URL: https://doi.org/10.1145/3062341.3062373.

[9] Petar Tsankov et al. "Securify: Practical Security Analysis of Smart Contracts". In: CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 67–82. ISBN: 9781450356930. DOI: 10.1145/3243734.3243780. URL: https://doi.org/10.1145/3243734.3243780.

[10] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: A Static Analysis Framework For Smart Contracts". In: *CoRR* abs/1908.09878 (2019). arXiv: 1908.09878. URL: http://arxiv.org/abs/1908.09878.

[11] Mark Mossberg et al. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts". In: *CoRR* abs/1907.03890 (2019). arXiv: 1907.03890. URL: http://arxiv.org/abs/1907.03890.

[12] Víctor Pérez et al. "Cost Analysis of Smart Contracts Via Parametric Resource Analysis". In: Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 7–31. ISBN: 978-3-030-65474-0. DOI: 10.1007/978-3-030-65474-0_2. URL: https://doi.org/10.1007/978-3-030-65474-0_2.

[13] Anton Permenev et al. "VerX: Safety Verification of Smart Contracts". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1661–1677. DOI: 10.1109/SP40000.2020.00024.

[14] Vineet Rajani. "A type-theory for higher-order amortized analysis". PhD thesis. 2020. DOI: http://dx.doi.org/10.22028/D291-30877.

[15] Rival Xavier and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. The MIT Press, 2020. ISBN: 9780262043410.

[16] Emil Jørgensen Njor and Hilmar Gústafsson. "Static Taint Analysis in Rust". MA thesis. Aalborg University, 2021. URL: https://projekter.aau.dk/projekter/files/421583418/Static_Taint_Analysis_in_Rust.pdf.

[17] Bytecode Alliance. *WebAssembly runtime*. URL: https://github.com/bytecodealliance/wasmtime. (accessed: 15.06.2022).

[18] Handan Kilinc Alper. *Blind Assignment for Blockchain Extension (BABE)*. URL: https://research.web3.foundation/en/latest/polkadot/block-production/Babe.html. (accessed: 10.05.2022).

[19] Meta Experimental. *MIRAI*. URL: https://github.com/facebookexperimental/MIRAI. (accessed: 20.06.2022).

[20] Fabio Lama, Florian Franzen, and Syed Hosseini. *Polkadot Protocol Specification*. URL: https://github.com/w3f/polkadot-spec/releases/download/v0.2.0-beta2/polkadot-spec_v0.2.0-beta2.pdf.

[21] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: https://bitcoin.org/bitcoin.pdf. (accessed: 20.06.2022).

[22] Rust compiler team. *clippy*. URL: https://github.com/rust-lang/rust-clippy. (accessed: 20.06.2022).

[23] Rust compiler team. *MIR dataflow analysis*. URL: https://rustc-dev-guide.rust-lang.org/mir/dataflow.html. (accessed: 15.06.2022).

[24] Rust compiler team. *miri*. URL: https://github.com/rust-lang/miri. (accessed: 20.06.2022).

[25] Rust compiler team. *Rust MIR rfc*. URL: https://github.com/rust-lang/rfcs/blob/master/text/1211-mir.md. (accessed: 20.06.2022).

[26] Parity Technologies. *Polkadot*. URL: https://github.com/paritytech/polkadot. (accessed: 15.06.2022).

[27] Parity Technologies. *Substrate*. URL: https://github.com/paritytech/substrate. (accessed: 15.06.2022).

[28] Parity Technologies. *WebAssembly Interpreter*. URL: https://github.com/paritytech/wasmi. (accessed: 15.06.2022).

[29] Dr. Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. URL: https://ethereum.github.io/yellowpaper/paper.pdf. (accessed: 20.06.2022).

A List of implemented manual specifications

core::slice::
binary_search_by
binary_search_by_key
cmp::SliceContains::slice_contains

custom specifications
pallet_multisig::ensure_sorted_and_insert

frame_system::
ensure_none
ensure_root
ensure_signed
pallet::Pallet::block_number
pallet::Pallet::extrinsinc_index
pallet::Pallet::deposit_event

parity_scale_codec::
Encode::using_encoded
Decode::decode

sp_io::
hashing::blake2_256

sp_runtime::traits::
CheckedMul::checked_mul
Convert::convert
One::one
StaticLookup::lookup
StaticLookup::unlookup
Saturating::saturating_add
Saturating::saturating_mul
Saturating::saturating_sub
TrailingZeroInput::new
Zero::is_zero
Zero::zero

frame_support::
BoundedVec::get_mut
BoundedVec::remove
BoundedVec::retain
BoundedVec::try_insert
BoundedVec::try_push
dispatch::Dispatchable::dispatch
dispatch::GetDispatchInfo::get_dispatch_info
dispatch::UnfilteredDispatchable::dispatch_bypass_filter
pallet_prelude::StorageDoubleMap::get
pallet_prelude::StorageDoubleMap::insert
pallet_prelude::StorageDoubleMap::remove
pallet_prelude::StorageDoubleMap::try_get
pallet_prelude::StorageMap::contains_key
pallet_prelude::StorageMap::get
pallet_prelude::StorageMap::insert
pallet_prelude::StorageMap::mutate
pallet_prelude::StorageMap::remove
pallet_prelude::StorageMap::take
pallet_prelude::StorageMap::try_get
pallet_prelude::StorageMap::try_mutate
pallet_prelude::StorageMap::try_mutate_exists
pallet_prelude::StorageValue::append
pallet_prelude::StorageValue::decode_len
pallet_prelude::StorageValue::exists
pallet_prelude::StorageValue::get
pallet_prelude::StorageValue::kill
pallet_prelude::StorageValue::mutate
pallet_prelude::StorageValue::put
pallet_prelude::StorageValue::set
pallet_prelude::StorageValue::try_append
pallet_prelude::StorageValue::try_get
pallet_prelude::StorageValue::try_mutate
StorageMap::get
StorageValue::get
traits::Currency::transfer
traits::EnsureOrigin::ensure_origin
traits::EnsureOrigin::try_origin
traits::fungible::Inspect::reducible_balance
traits::Get::get
traits::LockableCurrency::remove_lock
traits::LockableCurrency::set_lock
traits::OriginTrait::set_caller_from
traits::ReservableCurrency::can_reserve
traits::ReservableCurrency::repatriate_reserved
traits::ReservableCurrency::reserve
traits::ReservableCurrency::reserved_balance
traits::ReservableCurrency::slash_reserved
traits::ReservableCurrency::unreserve
traits::VestingSchedule::add_vesting_schedule
traits::VestingSchedule::can_add_vesting_schedule

std::
alloc::Allocator::allocate
alloc::Allocator::deallocate
alloc::Allocator::exchange_malloc
alloc::handle_alloc_error
clone::Clone::clone
cmp::Ord::cmp
cmp::PartialEq::eq
cmp::PartialOrd::partial_cmp
convert::AsRef::as_ref
convert::From::from
convert::Into::into
convert::TryInto::try_into
default::Default::default
intrinsincs::add_with_overflow
intrinsincs::assert_inhabited
intrinsincs::assume
intrinsincs::min_align_of_val
intrinsincs::mul_with_overflow
intrinsincs::ptr_guaranteed_eq
intrinsincs::saturating_add
intrinsincs::size_of_val
intrinsincs::transmute
intrinsincs::unlikely
iter::IntoIterator::into_iter
iter::Iterator::collect
iter::Iterator::enumerate
iter::Iterator::filter
iter::Iterator::filter_map
ops::Add::add
ops::BitOr::bitor
ops::Deref::deref
ops::Div::div
ops::Fn::call
ops::FnMut::call_mut
ops::FnOnce::call_once
ops::FromResidual::from_residual
ops::Index::index
ops::Index::index_mut
ops::Mul::mul
ops::Rem::rem
ops::Sub::sub
ops::SubAssign::sub_assign
ops::Try::branch
result::unwrap_failed
slice::SliceIndex::get
slice::to_vec
vec::Vec::insert
vec::Vec::
vec::Vec::new
vec::Vec::push