# The Experiment Orchestration Toolkit (ExOT)

Bruno Klopott[⋆§] Philipp Miedl[†§] Lothar Thiele[‡§]

## Abstract

*Researchers are required to support their claims with experimental evidence and provide results that are reproducible, comparable and exhaustive. However, the effort required to conduct exhaustive experimental analyses, or reproduce and compare different results, has proven to be high. To tackle these issues, we present the Experiment Orchestration Toolkit (ExOT).*

*In this whitepaper, we give a detailed overview of the design and implementation strategies used during the development of the first public version of ExOT. ExOT is designed to be easily extended and can be used to easily include a variety of different platforms in a measurement setup. It helps to automate the process of setting up, executing and analysing measurements. All components of the ExOT project that were developed at ETH Zürich are publicly available under the 3-clause BSD license.*

## 1. Introduction

The Experiment Orchestration Toolkit (ExOT) project emerged from research conducted at the Computer Engineering Group at the Computer Engineering and Networks Laboratory at ETH Zürich. Various versions of ExOT were used to conduct the experiment for multiple publications [2], [3], [13], [14], [24], which allowed the framework to evolve into a flexible measurement toolkit.

The basic design of ExOT assumes a measurement setup with five components:

- The measurement environment consisting of different *zones*.
- The *source* application which actively interacts with the measurement environment.
- The *sink* application which observes the measurement environment.
- The *jammer* applications, which tamper with the measurement environment to provide the possibility for controlled external influences onto the measurement.
- The *experiment engine* for data processing and experiment orchestration.

These five components and their relation are illustrated in Figure 1.

ExOT is designed to be easily extendable, allowing it to be applied in a broad range of measurement tasks. The main goals were to *reduce the engineering* burden placed on the researchers to conduct *extensive and scalable measurements campaigns*, to support the reproducibility, comparability and expressiveness of measurement results.

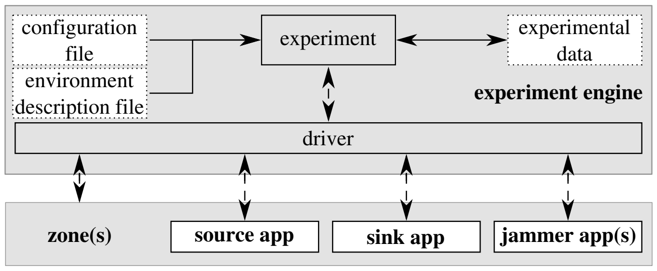


Figure 1: Block diagram of Experiment Orchestration Toolkit (ExOT), illustrating the relation of the different components.

In this paper, we give detailed insights into the design of the first public version of ExOT, which is an extensive software package compromised by an application library, an application compilation suite, and the experiment engine. The application library and the compilation suite speed up the implementation of source, sink and jammer applications for different processor architectures. The experiment engine mimics the layered structure of a communications channel, and simplifies and systematises the information flow and the analyses.

In Section 2 we present implementation details of the application library and in Section 3 we illustrate how applications are build and which testing facilities are included in ExOT. Section 4 gives an overview of the Android integration of ExOT and in Section 5 we present the implementation strategy for the experiment engine. In Section 6 we present small examples of the usage of ExOT, list possible future extensions in Section 7 and give some concluding remarks in Section 8.

## 2. The application library

The *Kahn Process Networks* model was chosen as the conceptual underpinning of the application library. The model allows for great extendability and reusability, because the individual nodes of the network are self contained, and communicate only through well defined interfaces. Any number of nodes can be introduced without any impact to the existing functionality and the model is simple yet expressive. Processes in *process*

*bruno.klopott@alumni.ethz.ch
†miedlp@ethz.ch
‡thiele@ethz.ch
§ETH Zurich, Computer Engineering and Networks Laboratory (TIK), Gloriastrasse 35, Zurich, Switzerland

*network* nodes can communicate only via unbounded first-in, first-out queues, but can perform computation of any degree of complexity[1]. The message passing semantics are rather straightforward, requiring blocking read accesses and non-blocking writes to the queues. This means that a process attempting to read from an empty queue will be suspended until a token is available. Such a model can describe systems that process streams of data, which can run sequentially or concurrently. Further details about the model of computation, its extensions, and implementation requirements are presented by Geilen and Basten [38, ch. 2], Allen [32, ch. 3] and Vrba [35].

## 2.1. Related work

A number of potential candidates for core *process network* support were identified: (i) Computational Process Networks (CPN) by Allen [32], (ii) RaftLib++ [27], (iii) FastFlow [30], (iv) Yapi [39], and (v) Threading Building Blocks (TBB) by Intel [31]. Nornir [35] was also investigated, but the implementation has not been made publicly available. Task-based models were also briefly explored; a survey is given in [34].

After reviewing these related libraries we decided to implement ExOT from scratch due to limitations of these libraries or them being outdated or unmaintained. However, a few observations were influential on the subsequent design process:

- *Templatable input and output ports.* CPN, TBB, RaftLib++, and Yapi use template parameters to indicate which data type is used by the input and output interfaces. In TBB, the parameters are passed to node declarations, e.g., `tbb::flow::function_node<int, int>`. In RaftLib++ the input/output interfaces contain template member functions, e.g. `output.addPort<int>(/*...*/)`. In Yapi and CPN, a separate interface class template is used, e.g., `Out<int>`. Such an approach seems certainly superior to casting a `void*` argument, and better than configuring data types at runtime.
- *Using strong types.* Some of the libraries seemed to be using code constructs specific to C in their C++ codebases. Most notably, some used generic pointers to `void` for "carrying" data between nodes. Such an approach provides no type safety, and might result in unexpected errors at runtime. Stronger typing can help prevent many errors at compile time and make software less ambiguous. Moreover, modern C++ provides avenues for polymorphic types and generic containers.
- *Abstracting the creation of queues from the user.* The libraries that did not require the explicit creation of queues seemed much friendlier. For example, in TBB the `tbb::flow::make_edge(node, node)` function is used, FastFlow uses consecutive calls to `pipeline.add_stage()`, and RaftLib++ provides a rather strange operator overloading (`map += producer >> consumer;`).
- *Function nodes.* Another aspect that distinguished TBB from the rest was the ability to quickly define nodes with a `function_node` class template. This feature makes it particularly useful for quick exploration and testing.
- Using pointers and references to objects and settings structures instead of string descriptions for configuring the *process network*. The libraries which rely on the latter required many more steps before the nodes and the network were usable.

[1]Data-flow graphs "have computational capability equivalent to a universal Turing machine" [42, p. 32].

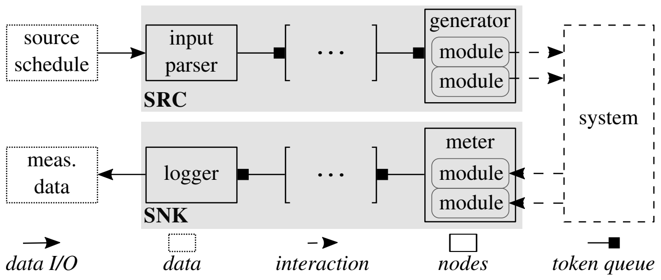


Figure 2: Process Networks model applied to the Experiment Orchestration Toolkit (ExOT) application design.

## 2.2. Application design

Figure 2 shows how the *process networks* concept could be applied to realise a measurement. The components of the channel model can be naturally expressed as *process network* nodes and connected with queues that can carry any data type, including heterogeneous or variable-size containers. The ellipses in the diagram indicate that other nodes could be introduced, as long as they conform to the input data type requirements of the dependent node.

The core of the library aims to provide the building blocks necessary for using the *process network* model.

**Nodes** Initially, the library will only provide support for single input and single output, which meets the needs of all existing covert channel applications. Three complementary classes of nodes are defined: consumers, producers, and processors. As the names suggest, they differ in the type of interfaces they provide, the latter combining both input and output interfaces. Similarly to GNU Radio and the frameworks listed in Section 2.1, the nodes contain a single executable process.

**Interfaces** Since queues are generic data containers, encapsulating interfaces are required to enforce the formalism of the *process networks*. The library will provide an abstraction of the underlying queues or communication channels, that ensures that only single-directional access is allowed. To allow for some deadlock avoidance, the interfaces in the library are extended with the ability to "time out". If the attempt to access the queue is not successful after a specified amount of time, it will be given up and indicated with a return status. If unsuccessful, there will be no side effects, i.e., the queue will not be modified.

**Connectors** In order to not burden the programmer with the creation of queues and bootstrapping the interfaces, the library will provide facilities to connect nodes together. The task of the connectors will be to verify the compatibility between nodes, create appropriate queues, and provide them to the nodes' interfaces.

**Executors** Executors will abstract the task of running the nodes' processes. These might in the future be used by bespoke schedulers, but initially will aim to execute the processes on threads that are scheduled by other entities, like the operating system.

### 2.3. Software design

From a software engineering perspective, the chosen design patterns and programming idioms support extendability, and provide an easy to use application programming interface (API).

To achieve reusability and extendability, the library aims to take the full advantage of the generic programming capabilities of C++. The language provides what is known as "templates", which allow creating classes and functions that can operate on different data types. For example, we may declare a function template:

```
template <typename L, typename R>
auto function(L arg_l, R arg_r);
```

An entity with such a declaration does not have any hard-coded types. Instead, the function template is instantiated when needed (implicitly or explicitly), and the compiler generates the actual function [see 19, *temp.spec*]. For example a call to `function(static_cast<int>(1), static_cast<long>(2))` will instantiate a function implicitly with template parameters `L` and `R` being `int` and `long`. With modern C++ we can also place constraints on the template arguments. For example, the instantiation of the function template above can be restricted to arithmetic types with:

```
template <typename L, typename R,
  typename =
  ↪  std::enable_if_t<std::is_arithmetic<L>::value
  &&
  std::is_arithmetic<R>::value>>
auto function(L arg_l, R arg_r);
```

Templates are even more powerful when applied to classes and their member functions, and combined with other language facilities, like inheritance. The following list provides some of the programming idioms that guided the development of the library.

**Policy-based design** This design technique makes use of templates to allow "assembling a class with complex behaviour out of many little classes, each of which takes care of only one behavioural or structural aspect" [40, p. 45]. The library will strive to make use of this idiom whenever some orthogonal functionality can be decomposed into smaller structures.

**"Template template parameters"** To facilitate using multiple template parameters, some of which also being template entities, the library makes use of "template template parameters". These constructs also help with policy-based design [40, p. 76]. Examples can be found throughout the Standard Template Library (STL); for example, the `std::vector` is a class template with a template parameter for the value type, but also a parameter for an allocator, which itself is a class template. "Template template parameters" allow propagating types, making the API cleaner. In the library, this idiom will be used to pass value types to containers, which then will be passed to interfaces operating on them. Since this idiom is quite difficult to describe, an application example is given in Section 2.5.1 (for the code sample shown in Listing 1).

**RAII** The behaviour known as "resource acquisition is initialisation" is used throughout the library to ensure that access to a particular resource is held during an object's lifetime. Notably, the library will use reference-counted smart pointers for managing dynamically allocated queues. Thanks to that, there will be no risk of ending up with a "dangling pointer", since the shared queue will not be destroyed as long as there is any entity holding a reference to it.

**SFINAE** "Substitution failure is not an error" is a rule that applies to function templates. With so-called "type traits" and compile-time polymorphism it is possible to provide conditional overloading of function templates via `std::enable_if` (e.g., a single print function that has different overloads for different types), check for the existence of specific member functions, or to provide static checks of matching types. In the library, it will be used to provide generic functionality while avoiding unnecessary abstraction through class hierarchy.

**Meta-programming & variadic templates** Templates can also be used to "generate" code at compile time, in a much type-safer manner than using preprocessor definitions. That also includes function and class templates that work with variable number of different types that need not share a common base class. This idiom can be particularly powerful when combined with inheritance in class templates, allowing the functionality of multiple smaller classes to be joined together.

### 2.4. Library structure

The software has been arranged in descriptive namespaces:

**exot::framework** Defines the application library core functionality, including the process network nodes (consumers, producers, and processors), input and output interfaces to communication queues/channels, concurrent single-producer, single-consumer queues, thread-safe state holder, node and pipeline connectors, and executors.

**exot::utilities** Includes a range of general-purpose and helper functions and classes, including time keeping facilities, thread attributes, synchronisation primitives, template meta-programming facilities, file-system utilities, bit manipulation helpers, command line parsing, logging, type traits, input and output stream overloads, and workers.

**exot::components** Contains reusable complete process network components, such as load generators, loggers, and input schedule parsers.

**exot::modules** Includes modules used for *sink* meters, arranged by measured physical quantities (frequency, power, temperature).

**exot::primitives** Contains functions and classes that access low-level subsystems, such as model specific registers, time stamp counters, and platform-specific helpers.

### 2.5. Application library core functionality

In this subsection, we present the application library core functionality, implemented in the *exot::framework* namespace.

#### *2.5.1. Node structure*

The most fundamental building blocks of the applications are the process network nodes. Three classes of nodes are defined: *consumer*, *producer*, and *processor*. As the names suggest, the *consumer* node has the ability to only read data (consume tokens) from an input interface, the *producer* node provides a write-only interface, and the *processor* node combines the two. No bidirectional communication over a single interface is possible.

The nodes are realised using class templates with "template template parameters", using the technique described earlier in Section 2.3. The declaration of such a class is shown in Listing 1. The `Token` type is passed on as a template parameter to the `Container`, which then both are passed on to the `Reader` template parameter.With such construction, one only needs to write `IConsumer<int, queue, queue_reader>` instead of the more verbose `IConsumer<int, queue<int>, queue_reader<int, queue<int>>>`. Node classes derive from virtual, empty base classes (e.g. `TConsumer`), which are necessary to make type traits more usable with the class templates.

```
template <typename Token, template <typename...>
↪  typename Container,
          template <typename, template <typename...>
          ↪  typename> typename Reader>
class IConsumer : public virtual Node, public virtual
↪  TConsumer {
 public:
  using consumer_type = IConsumer<Token, Container,
   ↪  Reader>;
  using interface_type = Reader<Token, Container>;

  IConsumer() = default;
  explicit IConsumer(typename
   ↪  interface_type::container_pointer input_queue)
      : in_(input_queue){};
  virtual ~IConsumer() = default;

  void set_input(typename
   ↪  interface_type::container_pointer input);
  typename interface_type::container_pointer
   ↪  get_input() const;

 protected:
  interface_type in_;
};
```

Listing 1: Consumer node interface

Each node class template contains an interface object, which can either be initialised in the constructor, or configured after instantiation using `set_input` and `set_output` functions. The end user will rarely, if ever, need to use the non-default constructor. Classes deriving publicly from the these base node templates can then access the inherited interface object thanks to the `protected` access specifier.

The class template for the *processor* node has multiple inheritance from both *consumer* and *producer* node classes. One important feature of *processor* classes is their ability to bridge potentially disparate domains; hypothetically speaking, the consumer side could be connected to a network interface, and the producer side to a regular queue.

A UML class diagram which illustrates the relationships between the various class templates, regular, abstract and interface classes, is presented in Figure 3. Such an organisation of software components allows for easy extendability and reuse. Since their most often changed and crucial aspects are template parameters, it is trivial to declare nodes that deal with various token data types. If a new container or an interface is designed, to use it with the node, one only needs to pass them as template parameters. For further convenience template aliases are provided for default containers and interfaces, such that the user only needs to supply the token types used by the node.

The classes above are generic and do not yet define any

```
template <typename Token, template <typename...> class
↪  Container> class
QueueReader : public IReader<Token>, public
↪  Reader<Token, Container>;

template <typename Token, template <typename...> class
↪  Container> class
QueueWriter : public IWriter<Token>, public
↪  Writer<Token, Container>;
```

Listing 3: Queue interfaces

executable elements. To implement some functionality, a user can derive from the `IProcess` interface class that defines a process-oriented execution model. The interface contains a pure virtual member function that needs to be overwritten by implementing classes. In addition to the `void process()` function the interface class adds the commonly used shared pointer to the global execution state. This is also the only place where an object, the `GLOBAL_STATE`, is declared with the `extern` specifier for external linkage.

### 2.5.2. Interfaces

Communication between process network nodes happens through interface class templates. They provide the set of functionality required for satisfying the requirements of the process network model, that is *blocking reads*, and *non-blocking writes* (as long as queues are declared unbounded). The two pure virtual interface class templates shown below define the functions required for input (Figure 4) and output interfaces (Listing 2). The interfaces enforce the process network formalism and allow for different calling styles of the underlying containers (some use the combination of `front`, `pop` and `push` methods, while others define `enqueue` and `dequeue`). Similar layers of indirection are present in other process network implementations [35, p. 73, 41, p. 19].

Additional `Reader` and `Writer` base class templates allow reuse of constructors and provide common functionality of setting and getting the shared pointers to containers used as the 'transmission medium' between nodes. The concrete interfaces, the class templates `QueueReader` and `QueueWriter`, implement the reader and writer virtual interfaces for queue-like objects, and derive from the base classes above (Listing 3).

The extended queue interfaces additionally implement alternative semantics and define member function templates such as `try_read_for(token, timeout)`, as shown in Listing 4. Analogous functions are provided for the writer interface. The important distinction from the basic interface is the addition of a return type (`bool` instead of `void`), which indicates whether the operation was successful and allows for less conventional, defensive

```
inline bool try_read(Token &token);
template <typename Rep, typename Period>
inline bool try_read_for(Token &token,
                         const
                          ↪  std::chrono::duration<Rep,
                          ↪  Period> &timeout);
template <typename Clock, typename Duration>
inline bool try_read_until(
    Token &token, const std::chrono::time_point<Clock,
    ↪  Duration> &time);
```

Listing 4: Extended queue interface

code constructs.

### 2.5.3. Queues

The underlying communication channel has to abide by the process network formalism. The queue provided in the C++ standard libraries does not meet the requirements. First of all, `std::queue` is non-blocking. An empty queue can be "popped", and calls to the `front()` method can return data from uninitialised memory when the queue is empty. Moreover, `std::queue` is not thread-safe. Even with the queues having only two users, concurrent access has to be free of race hazards.

The library provides a concurrent queue suitable for single producer-consumer scenarios that mirrors the interface of `std::queue`, and its more complex extension that additionally implements the extended semantics. The queues are provided as class templates, allowing the use of different token data types and synchronisation primitives. Moreover, the queues can have bounded capacity.

The implementation uses two thread synchronisation constructs: mutual exclusion locks, and condition variables. Their combination allows efficient locking and monitoring the status of certain boolean conditions. A lock protect access to private class variables in each of the public interface functions. Two condition variables are used for waiting and notifying on empty and full queues.

All read operations acquire the lock and wait until the queue is not empty, typically using the code construct listed below:

```
std::unique_lock<mutex_type> lock(queue_mutex_);
queue_empty_cv_.wait(lock, [this] { return !empty_();
↪  });
```

An analogous mechanism is provided for write operations, which will block on a full queue. Waiting using a condition variable is roughly equivalent to `while(!predicate) lock.lock();`.

If the read or pop operation was called on an empty queue, the calling thread will efficiently wait until notified by another thread performing a write operation.

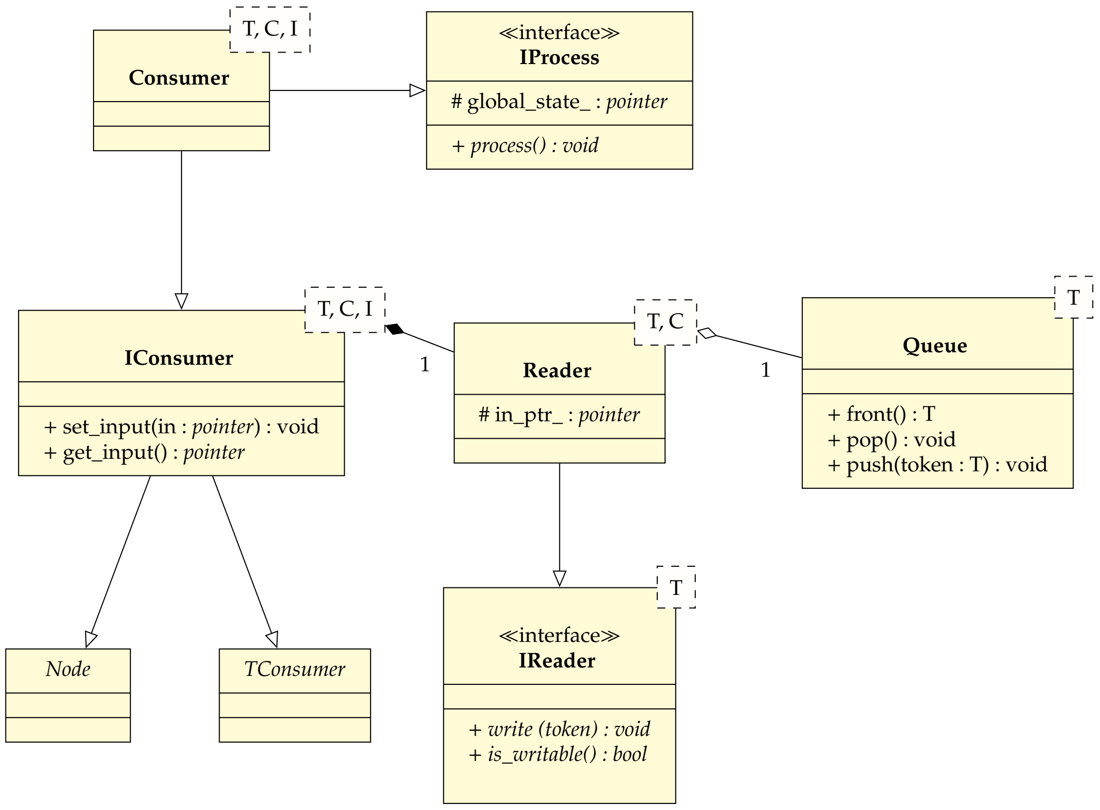


Figure 3: UML class diagram showing relationships in core framework's nodes

```
template <typename Token>
struct IReader {
  virtual ~IReader() = default;
  virtual void read(Token &) = 0;
  virtual bool is_readable() = 0;
};
```

Figure 4: listing
Reader interface

```
template <typename Token>
struct IWriter {
  virtual ~IWriter() = default;
  virtual void write(const Token &) = 0;
  virtual void write(Token &&) = 0;
  virtual bool is_writable() = 0;
};
```

Listing 2: Writer interface

Notifications are provided by two internal functions `notify_waiting_on_empty_()` and `notify_waiting_on_full_()`.

The extended interface provides `try_pop` and `try_push` functions, which do not block if the queue is empty or full; if the preconditions are not satisfied, the queue will not be modified and the functions will return `false`.

The usefulness of the condition variables becomes apparent in functions that attempt to read or write only for a specified period of time. The Listing 5 shows an excerpt from the source code, which defines a `try_pop_for` function. It's a function templates, which can be called with any duration object from the standard library, e.g. `try_pop_for(token, std::chrono::seconds{2})`.

```
template <typename Rep, typename Period>
bool try_pop_for(reference value, const duration<Rep,
↪  Period> &timeout) {
  std::unique_lock<mutex_type> lock(queue_mutex_);
  if (queue_empty_cv_.wait_for(lock, timeout, [this] {
  ↪  return !empty_(); })) {

    // ...

    lock.unlock();
    notify_waiting_on_full_();
    return true;   // if successful
  } else {
    return false;  // if timed out
  }
}
```

Listing 5: Accessing the queue with a timeout

### *2.5.4. State*

State objects in the library can be used to track local and global state. They contain a set of atomic boolean variables and functions to access and manipulate them, for example `start()` and `is_started()`. The objects are meant to be used through shared pointers, therefore the `State` class derives in a CRTP[2] fashion from `std::enable_shared_from_this<State>` and provides a member function to get a shared pointer to itself.

The only global variable with static storage duration used in the project is the pointer to a state object:

```
static State::state_pointer
↪ GLOBAL_STATE{std::make_shared<State>()};
```

Thanks to the static storage specifier the pointer can be used in C-style signal handlers. Currently, the library defines the following global state signal handlers:

```
static void interrupt_handler(int) {
↪ GLOBAL_STATE->stop(); }
static void terminate_handler(int) {
↪ GLOBAL_STATE->terminate(); }
static void start_handler(int) {
↪ GLOBAL_STATE->start(); }
```

### *2.5.5. Connectors*

One of the limitations of some of the reviewed process network implementations was the need to manually create and connect queues to nodes. To remove this burden from the users, a method for connecting compatible nodes together is provided in the library.

A function template `details::connect` takes two nodes as arguments, verifies that they have compatible interfaces and token types, constructs a queue and passes the shared pointer to neighbouring nodes. The verification uses custom type traits and happens at compile time thanks to `static_assert`.

Using a variadic function template, shown in Listing 6, any number of nodes can be connected. There is a base function that takes only two arguments, and a recursive function that takes any number of arguments. Thanks to template pack expansion a call to `connect(Node1, Node2, Node3)` will produce a sequence of calls `connect(Node1, Node2); connect(Node2, Node3);`. Additionally, the library provides a structure wrapper that allows setting the desired queue capacity[3], and a `pipeline` function. The `pipeline` function uses type traits and a compile-time for-loop to additionally verify that the first passed node is a producer, and the last one is a consumer.

### *2.5.6. Executors*

Executors provide a uniform way to execute processes. The base class provides a function template `spawn`, which can be used to execute any callable object, optionally with arguments. An example is listed below:

[2]Curiously Recurring Template Pattern, in which the class derives from a class template, using itself as a template argument.

[3]With the `Connector` class one can connect nodes with a queue of size 10 by calling `Connector(10).connect(Node1, Node2, Node3);`.

```
template <typename Left, typename Right>
void connect(Left &&left, Right &&right) {
  details::connect(std::forward<Left>(left),
  ↪ std::forward<Right>(right));
}

template <typename Left, typename Right, typename...
↪ Rest>
void connect(Left &&left, Right &&right, Rest &&...
↪ rest) {
  connect(std::forward<Left>(left), right);
  connect(std::forward<Right>(right),
  ↪ std::forward<Rest>(rest)...);
};
```

Listing 6: Variadic function template for connecting nodes

```
template <typename Callable, typename... Args>
void spawn(Callable &&callable, Args &&... args);
```

The concrete executor classes that are provided in the library at the moment are meant for executing callable objects on system and user-space threads. For example, the `ThreadExecutor` class will spawn each provided object in a separate system thread, and is well suited for executing process network nodes. It additionally provides a way of spawning an object on a specialised thread, with configurable pinning, scheduling policy and priority.

The other available executor facilitates the use of user-space threads, *fibers*, from the project **Boost/fiber**, and provides convenience functions for spawning *fibers* and adding *fiber* pool worker threads. Both executors also provide functions to query how many worker threads have been instantiated, and to wait for the completion of and join spawned system/user-space threads.

## 2.6. Utilities

A wide range of utility functions and classes are provided in the *exot::utilities* namespace of the application library. These provide both essential functionality, like time keeping or synchronisation mechanisms, and auxiliary helpers, like string formatting and type traits. We describe the most important of those utilities in the remainder of this subsection.

### *2.6.1. Timing*

Sleeping and estimating time offsets has been extracted into a `TimeKeeper` class template. The class gives a straightforward interface, with member functions `begin()`, `sleep(chrono::duration)`, and `update_offset()` providing the bulk of the functionality. Additionally, the class has a `run_every` function, which can invoke a callable object with arguments periodically until some predicate is false. For example, to perform and output some measurement every 10 milliseconds until the global state is terminated, one can run:

```
tk.run_every(std::chrono::milliseconds{10},
             [&]() { return
             ↪  !GLOBAL_STATE.is_terminated(); },
             [&]() { std::cout << meter.measure(); })
```

The time keeper can use different clock sources, different sleep functions, and also provides mean and standard deviation of intervals and offset. The `<exot/utilities/timing.h>` header also provides a `std::chrono`-like interface to the POSIX `nanosleep` function.

The library provides different timing primitives from literature, the ability to introduce serialisation to any clock type via a `fenced_clock` Curiously Recurring Template Pattern (CRTP) class template, and ExOT clock sources based on system's monotonic clock and performance counters. Moreover, it significantly simplifies the timing of any callable type with the help of a `timeit` function template. To support faster operation, the `TimeKeeper` utility has been provided the ability to use busy-loop sleeping.

### 2.6.2. Barrier synchronisation primitive

The barrier implements a rendezvous point for a configurable thread count. Threads arrive at the barrier and are allowed to pass it only once the specified number threads have reached it. The threads wait on the barrier efficiently, i.e. there is no busy waiting, and threads are idle. To allow for greater usability and portability, the older barrier primitive from the POSIX thread library has been replaced with a solution inspired by Boost libraries and C++ standard proposals [10], and uses just the C++ standard library. The advantage of the ExOT barrier is that it can be reused multiple times and can be used on the Android system with API support below level 24[4].

The ExOT barriers are implemented with a combination of locks and condition variables, both of which are template parameters, allowing for the use of different primitives. The barrier is initialised with a desired thread count. As threads arrive at the barrier, they increment an internal current thread count. If the count is less then the desired one, the entering thread waits on a condition variable. If the entering thread is the last to enter the barrier (incremented current count is equal to the desired count), it uses the condition variable to notify all waiting threads.

### 2.6.3. JSON interface

The application library has introduced generic support for JavaScript Object Notation (JSON)-based configuration. The format was chosen due to the simplicity of the markup language and very good software support with a library by Lohmann [6] that awaits standardisation.

[4]The functions `pthread_barrier_init` and `pthread_barrier_wait` are marked `__INTRODUCED_IN(24)` in the `<pthread.h>` header file in the Android's system root. API level 24 is not yet publicly available.

```
struct MyConfigurableClass :
↪  configurable<MyConfigurableClass> {
  std::vector<std::string> MyStrings;   //! A vector
  ↪  of strings
  std::map<std::string, double> MyMap;  //! A map of
  ↪  pairs string->double
  const char* name() const { return
  ↪  "MyConfigurableClass"; }
  void configure() {
    bind_json_to_data("MyStrings", MyStrings);
    bind_json_to_data("MyMap", MyMap);
  }
};

/* Later used as... */
MyConfigurableClass instance;
instance.set_json(the_json); instance.configure();
/* Or using a helper function... */
configure(the_json, instance);
```

Listing 7: Demonstration of configurable classes

It features strongly-typed support for all fundamental types and most data structures from the STL, as well as ease of providing overloads for functions performing serialisation and de-serialisation of user-defined types.

The support for JSON-configurable classes has been provided using the programming paradigm of CRTP. In a more common class hierarchy, we would use a base class with virtual or pure virtual functions, which are then implemented in derived classes. With this paradigm, however, the base class is a class template that takes the deriving class as a template argument, and inherits from it. Such structure allows a base class to access the derived class and simplifies multiple inheritance from other configurable classes. The simplicity of this solution is demonstrated in Listing 7. The `bind_json_to_data` instructs which key is to be mapped to which member variable.

### 2.6.4. Command line parsing

The command line interfaces in the library rely heavily on the very idiomatic header-only library `clipp` by 'muellan/clipp' [16]. One of the most helpful features of that library is the automatic generation of help messages, allowing for good extendability of applications and reducing the programmer's burden.

A separate class `CLI` builds upon the parsing facilities provided in the `clipp` library, and allows easily adding individual components configurations to a master configuration, adding description and example sections to the printed help message[5], and takes care of parsing the command line arguments and notifying about parsing errors. Applications using the JSON interface for configuration will provide a command line interface based on clipp as outlined in 8.

[5]Additional sections are automatically wrapped to 80 columns.

```
 $ ./generator_utilisation_mt
Parsing error: 4 missing/incorrect, 0 blocked, 0 conflicts

SYNOPSIS
        ./x86_64/generator_utilisation_mt ((--json_file <json file>) | (--json_string <json
            string>))

        ./x86_64/generator_utilisation_mt -h

OPTIONS
        (--json_file <json file>) | (--json_string <json string>)
                    JSON-based configuration

        -h, --help  print help message

CONFIGURATION
        [logging]
        | provide_platform_identification           should provide platform info? |bool|
        | debug_log_filename                        the debug log filename |str|, optional
        | app_log_filename                          the app log filename |str|, optional
        | log_level                                 the log level |str|, one of "trace", "debug", "info", "warn",
        ↪  "err", "critical", "off"
        | async_size                                async logger buffer size |uint|, power of 2
        | async                                     use the async logger? |bool|
        | async_thread_count                        async logger thread count |uint|
        | timestamp_files                           should timestamp the log files? |bool|
        | rotating_logs                             should rotate the logs? |bool|
        | append_governor_to_files                  should append the frequency governor to filenames? |bool|
        | rotating_logs_size                        the size of rotating logs in MiB |uint|
        | rotating_logs_count                       the maximum number of rotating logs to keep |uint|
        [schedule_reader]
        | input_file                                the input schedule file |string|, e.g. "input.sched"
        | reading_from_file                         reading from file? |bool|, reads from stdin if false
        | read_as_hex                               read hexadecimal values? |bool|
        | cpu_to_pin                                schedule reader pinning |uint|, e.g. 7
        [generator]
        | cores                                     cores to pin workers to |uint[]|, e.g. [0, 2]
        | should_pin_workers                        should pin the workers? |bool|, default 'true'
        | worker_policy                             scheduling policy of the workers |str, policy_type|, e.g.
        ↪  "round_robin"
        | worker_priority                           scheduling priority of the workers |uint|, in range [0, 99], e.g.
        ↪  99
        | host_pinning                              generator host core pinning |uint|, e.g. 5
        | should_pin_host                           should pin the host? |bool|, default 'true'
        | host_policy                               scheduling policy of the host |str, policy_type|, e.g.
        ↪  "round_robin"
        | host_priority                             scheduling priority of the host |uint|, in range [0, 99], e.g.
        ↪  99
        | start_check_period                        state change detection update period |uint, µs|, e.g. 100
        | use_busy_sleep                            should use busy sleep loop? |bool|
        | busy_sleep_yield                          should yield thread in busy sleep loop? |bool|
        [generator]
        | cores                                     cores to run workers on |uint[]|, e.g. [1, 2, 3]
```

Listing 8: Example for the command line interface of an application generated with the application library. Only a JSON file or a JSON string can be provided via the commandline. The help text lists all possible JSON configuration parameters of the application.

*2.6.5. File system utilities*

Since many of the application components access various pseudo-files provided by `procfs`, `devfs` and `sysfs`, the library provides functions for listing directories and for searching directories using regular expressions. Both recursive and non-recursive methods are given. For example, to search for all `thermal_zone` temperature access pseudo-files, the user needs only a single function invocation, `grep_directory_r("/sys/devices/virtual/thermal", ".*zone\\d+/temp$")`, which will return a vector of paths to the files (e.g., "…/thermal_zone0/temp").

*2.6.6. Meta-programming helpers and type traits*

The library provides several compile-time helpers, which are used in other parts of the library. These include a compile-time `for`-loop, which can also be used as a replacement for code generation preprocessor directives, and functions to apply functions to heterogeneous `std::tuple` containers. For example, to create 256 invocations of a callable object with signature `void(size_t)`, one can use `const_for<0, 256>([](auto I) { fun(I); })`. The template will be instantiated at compile time and expand to a sequence of calls `fun(0); fun(1); ...; fun(255);`.

Another functionality that does not have any direct application, but is used throughout the library are custom type traits. Type traits, which provide information about types and evaluate predicates at compile time, are heavily used in the C++ standard library. In the case of this work, they are used to check if a type is iterable or const-iterable, if a type is a tuple, or if a node has an input or output interface. These can later be used in `static_assert` statements, or to conditionally enable certain templates.

*2.6.7. Formatting and logging support*

To facilitate the process of formatting readings for logging, the library provides overrides for formatting any iterable or tuple-like objects to `basic_ostream` objects. An example of a signature of a function that overloads the `<<` operator is show in Listing 9. What this achieves is that all types `T`, which return true for `is_iterable<T>::value`, will be allowed to use this overloaded function. In practical terms, the user no longer has to write loops to output vectors. For example, this becomes a valid statement with the overload above: `std::cout << std::vector<int>{1, 2, 3, 4};`. Possibly the greatest utility comes from an overload for tuples, since iterating over tuples is not possible. The overload uses the `const_for` loop above to access tuple elements, allowing for printing of heterogeneous containers. All individual elements are comma separated in the output.

Similar overloads are provided for reading tuples and `chrono::duration` objects from `basic_istream` objects. This functionality is used in the *schedule reader* described later (Page 10).

*2.6.8. Logging*

Since logging is used universally across the whole library and in all applications, it has been extracted into a separate class, which has a uniform configuration interface. The `Logging` class is responsible for creating log files, setting log levels (e.g., critical, debug), and makes sure that the log files are readable. The functionality relies on the `spdlog` library [12], which provides a modern and highly configurable interface, and is also compatible with Android logging facilities. The logging library is also thread-safe and very fast: 4,328,228 messages per second can be logged to a file in a single-threaded mode on an Intel i7-4770 processor [12]. For convenience, most components and modules try using loggers defined in the global logger registry, both for application and debug logging.

*2.6.9. Thread parameters*

The library also features a `ThreadTraits` class which provides a portable way of setting the affinity and scheduling of threads. Most notably, a solution has been found for setting affinity on the Android platform, which was not possible in the legacy codebase. The Android-compatible way of pinning threads relies on the sys-call to `SYS_gettid` and using the POSIX `sched_setaffinity` function from the `<sched.h>` header.

*2.6.10. Workers*

Several worker classes are available in the library. A worker can wrap some callable object in a loop that checks the global execution state. Most notably, a worker using policy-based design principles is provided, which can take different synchronisation and threading policy/mixin classes, allowing easy extendability. For example, `Worker<BarrierSynchronisation, SpecialisedThreads>` declares a worker that will be pinned to a CPU and will invoke the callable object between two barriers.

## 2.7. Components

In this section, we present the reusable complete process network components, implemented in the *exot::components* namespace.

*2.7.1. Schedule reader*

The schedule reader is a producer class template that parses values line-by-line either from a file, or from the standard input (in contrast to standard input only in the legacy framework). For ease of use and reusability, the schedule reader forms a token from each line of input, as long as proper overloads were provided. At the moment any tuple type can be formed into a valid token. Thanks to such design, the schedule reader can be adapted to different components on the receiving end, with only minimal involvement of the user:

```
late <typename T, typename CharT, typename CharTraitsT>
name std::enable_if<exot::utilities::is_iterable<T>::value && !exot::utilities::is_const_iterable<T>::value,
↪  std::basic_ostream<CharT, CharTraitsT> &>::type & operator<<(std::basic_ostream<CharT, CharTraitsT> &ostream,
↪  const T &range);
```

Listing 9: Output stream operator overload

```
using loadgen = components::loadgen_mt;
using reader = components::schedule_reader<typename
 ↪  loadgen::token_type>;
```

As this example demonstrates, the developer does not even need to explicitly declare the token type, and can just use the internal types of other components. In addition, the schedul reader support reading tokens that contain iterable types (such as vectors) and fixed-size arrays.

*2.7.2. Logger*

The logger node is a very simple, but universal addition to the library. It is a class template that takes a token type as a template parameter. Thanks to the provided output stream overloads, it will format the token as a string and write it to the application log, separating individual values with commas.

*2.7.3. Function nodes*

The library also provides nodes that take a callable object as an argument to the constructor. These can be conveniently used for simpler operations and testing, and wrap the invocations of the callable in loop that monitors the global state. Three class templates are provided: `FunctionConsumer`, `FunctionProducer`, and `FunctionProcessor`, which take callable objects with signatures `void(token)`, `token(void)`, and `token(token)` respectively. Unlike C-like function pointers, the callable types can be more complex capturing and/or mutable lambdas. For example, to produce a sequence of 10 tokens the user of the library can write:

```
using token_type = std::tuple<int, std::string>;
int counter{0};

FunctionProducer<token_type> node([&]() mutable ->
 ↪  token_type {
  if (++counter > 10) GLOBAL_STATE->stop();
  return token_type{counter, std::string{"Token #"} +
   ↪  counter};
});

node.process();
```

*2.7.4. Platform identification*

In order to keep track of important platform characteristics alongside measurement log files the library implements a large number of identification functions. Among others, they are responsible for obtaining processor details, processor package topologies and frequency scaling settings.

*2.7.5. Adapter nodes*

The last type of components in the library allows bridging domains using different concurrency primitives: system and user-space threads. These nodes simply forward tokens, in a way that does not cause race issues.

*2.7.6. Meter host*

The host allows combining individual meter modules into a readily usable process network component. First, the `MeterHost` is class template, where meter modules are provided as variadic template parameters. The `MeterHost` then inherits publicly from each module, thanks to template pack expansion, as shown below:

```
template <typename Duration, typename... Meters>
class meter_host
    : public Meters...,
      public framework::IProcess,
      public
        ↪  framework::Producer<meter_token_type<Duration,
        ↪  Meters...>>
```

It's token type is also determined automatically from the inherited modules, using an alias template `meter_token_type`:

Secondly, all module configuration structures are combined and appended with host-specific settings, also using multiple inheritance and template pack expansion. The host has its own settings class, `struct settings : Meters::settings...;`, and own `configure` function, which incorporates module-specific equivalents. The host's settings class is then passed to modules' constructors; each module only accesses its relevant part of the structure.

The host also combines all meter module measurement functions, and the results of their reported variables' names and units. Thanks to that the meter host's process is very compact:

```
auto until = [this]() { return
 ↪  !global_state_->is_stopped(); };
auto action = [this]() { out_.write(measure()); };
timer_.run_every(conf_.period, until, action);
```

*2.7.7. Meter modules*

To facilitate the declaration and use of metering facilities, the module class type has been introduced. The UML class diagram in Figure 5 shows the relationship between the meter host and the modules. A meter module has only a few requirements:

- It must have a settings structure, `struct settings`, which can provide defaults and can be empty;
- It must have a constructor that takes the settings structure by reference;
- It must have a `measure()` function;
- It should provide a `vector<string> names_and_units()` function if the user wants to facilitate the process of adding headers to log files;
- The meter module should follow the Resource Acquisition Is Initialisation (RAII) idiom, and should be perfectly usable after instantiation.

There are no hard requirements imposed on the what kind of values a meter module can return via the `measure()` function. It is perfectly valid to have one meter return a `std::tuple<std::string, int, double>` and another return `std::vector<long>`. Although the meter function can return variable length arrays, it might be a better idea to keep the size constant after initialisation to obtain a well formed output.

The use of meter modules is enabled by the meter host component, described on Page 11. The modules can also be used independently of the host, for example in a reactive load generator.

*2.7.8. Generator host component and module base classes*

The generator host and the generator modules are built in a similar fashion to the meter host and meter modules. The generator host works in conjunction with generator modules, which define an interface composed of `decompose_subtoken`, `validate_subtoken`, and `generate_load`. The structure of both relies on type traits rather than class hierarchies, therefore generator modules can choose the types of each of the member types at their discretion. Moreover, to further simplify the creation of many similar generator and metering modules the library introduces base classes for those that require shared memory or interpret their input as bitsets.

## 2.8. Primitives

In addition to generic utilities, the library also provides more platform-specific utilities, arranged in the separate *exot::primitives* namespace. At the moment, these mostly include functionality specific to Intel-based platforms.

*2.8.1. Model specific register access*

The library provides a class for reading and writing *model specific registers* (MSR), which provides some improvements over the methods used in legacy code. In line with the overall aim of making most of the classes conform to the RAII idiom, the class initialises access to MSRs upon instantiation. A number of checks are performed to make sure that the arguments supplied in the constructor are sensible and that all registers can be accessed. Moreover, in addition to `read` and `write` functions, the class provides methods `read_first`, `read_any`, and `read_all`, the latter returning a vector of readings. These functions, which have their write access counterparts, are mostly meant for users who use the default constructor of the class.

*2.8.2. Time stamp counter clock*

Quite an interesting addition to the library is a clock that explicitly uses the time stamp counter (TSC) on Intel processors as the time source. The class provides the same interface like the clocks found in the header of the C++ standard libraries. Thanks to that all the facilities in the library (operations on and between time point and duration objects), handling of timing measurement with any clock implementing, that implements the standard interface, is very simple, as shown below:

```
auto begin = tsc_clock::now();
// ...
auto elapsed =
    duration_cast<microseconds>(
        begin - tsc_clock::now());
```

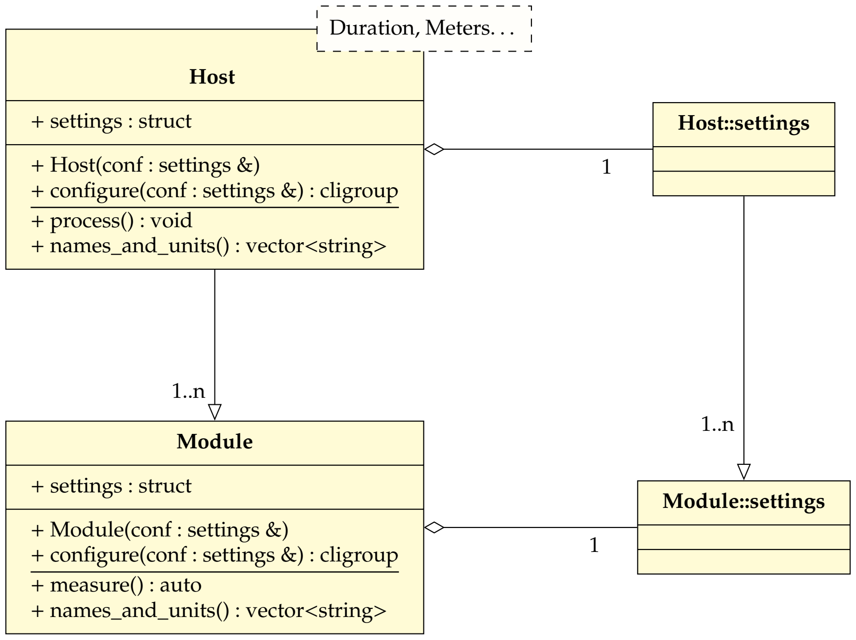
Using the TSC as the clock source can be quite beneficial, but there is always the need to estimate the frequency of the monotonic counter. To maintain a clean and usable interface, in `tsc_clock` the estimation happens upon the first invocation of the `tsc_clock::now()` function. Every subsequent invocation can use the estimate immediately, because the relevant function-local variables have have static storage duration and are initialised the first time the control flow passes their declarations.

*2.8.3. Memory mapping support*

On the Linux Operating System (OS) and other unix-based systems shared memory is enabled by, among others, `mmap`. A ExOT `MMap` class implements a safe and convenient wrapper that follows the programming idiom of RAII. It allows mapping file-backed and anonymous memory into the virtual address space of the calling process, including anonymous and shared huge pages allocations. The support for huge pages is necessary in situations that require contiguous physical memory that is aligned on the huge page size supported by the processor. For example, with 2 MB contiguous memory one can very easily find addresses that map to the same cache set without the need for allocating a large memory buffer.

*2.8.4. Cache parameters discovery*

The library provides a `CacheInfo` class, which performs automatic detection of specific cache's properties (such as ways of associativity or cache line size). A `CPUCacheInfo` class discovers all caches of a particular processor core. Moreover, both can be manually configured with any iterable type holding unordered maps of cache properties.



Duration, Meters. . .
Host
+ settings : struct
+ Host(conf : settings &)
+ configure(conf : settings &) : cligroup
+ process() : void
+ names_and_units() : vector<string>
Host::settings
1
1..n
1..n
Module
+ settings : struct
+ Module(conf : settings &)
+ configure(conf : settings &) : cligroup
+ measure() : auto
+ names_and_units() : vector<string>
Module::settings
1

Figure 5: UML diagram showing relationship between meter host and modules

### *2.8.5. Virtual-to-physical address translation*

The library features an `AddressTranslator` functor class that translates a virtual address into a physical one using the *pagemap* exposed for each process in Linux'es *procfs*. The pagemap is the Linux kernel's interface that allows programs running in the user space to find out the mapping between virtual memory pages and physical frames.

### *2.8.6. Custom allocators and alignment helpers*

To benefit from the ease-of-use of STL containers while being able to control their memory allocation, the library provides an array of custom allocators. For example, the `AlignedAllocator` can be used to align a container on a specific boundary and the `HugePagesAllocator` will allocate anonymous huge page memory to store the data. The allocators are passed as the second template argument to many containers, for example: `std::vector<int, HugePagesAllocator<int>`. Additionally, the library provides a type wrapper that makes it easier to make types with explicit alignment. The `aligned_t<T, A>` class template can be used to produce a type `T` that has the desired alignment `A`, for example via `make_aligned<64>(1ull)`. In this example the resulting type would have an alignment of 64 bytes, and array of such types would have each element located in a different cache lines on most platforms. Silarly, one could define a type that is aligned at virtual memory page size boundaries of 4 KB.

### *2.8.7. Cache slice hash function*

The library also provides the cache slice selection hash function as reverse engineered by Maurice, Le Scouarnec, Neumann *et al.* [28] for older Intel architectures.

## 3. Compilation suite and testing

The development criteria for ExOT included readability, support for cross-compilation, the Android platform, and testing, reproducibility of builds, ease of instrumentation with analysis tools. The tools that were considered were the Python-based SCons and Waf, Meson, Bazel, Buck, and CMake. CMake, Bazel and Meson were given particular attention because they allow creating toolchain descriptions, useful when compiling for different targets from different host machines. Although Bazel and Meson seem to have a more consistent and friendlier syntax, CMake was chosen as the build system for the project. The primary motivations were more robust support for C and C++ projects, longer history, and official support in the Android build system. Moreover, CMake does not build the software itself, but rather generates other build systems' files, including Unix Makefiles, which can be used on different operating systems and in Integrated Development Environment (IDE). Compared with GNU Make, CMake provides better convenience, superior correctness and easier scalability [33, p. 146, p. 262].

The library is structured into four directories: *include/exot*, *src*, *vendor* and *test*. The *include/exot* directory contains folders with header files, which reflect the namespace organisation described earlier. To use or add a piece of library code, a developer can include the files in a meaningful way, without caring about the relative path, e.g. `<exot/utilities/barrier.h>`, because the include directories are exported with the library target. The *src* folder contains all source files used for building object code. *Vendor* contains the external third-party dependencies of the library. The need for the *vendor* folder unfortunately arises from the difficulty of importing external CMake-enabled code. The dependencies do not have to be committed into the repository; they are set up as git submodules, and a call to `git submodule init` will pull the repositories and point to specific commits. This ensures that only stable features of vendor libraries are used.

The library build targets are governed by the top-level `CMakeLists.txt` file. It defines the project `exot-library` and the main build target, `exot`, which a static library `libexot.a`.The target is configured with a range of compile features and options, depending on the build configuration type: Release, or Debug. The third-party libraries, most of which include are CMake projects, are included in the top-level configuration. Thanks to that, all dependant targets will inherit the compiler and build settings from the main project. This avoids the situation in which an imported library, possibly from system paths, has been compiled using a different compiler, standard library or ABI. Having an application linked to different standard libraries is strongly discouraged.

Moreover, there are custom targets to auto-format code to achieve uniform style, and to enable static analysis checks using `clang-tidy`. When enabled, the static analyser will be invoked during each build, and a `tidy-all` target will be created, which lints all library code. In the Debug configuration custom targets are also provided with LLVM sanitisers enabled; for example, to compile the target `exto` with the memory sanitiser, one can use an auto-generated target `exot-san-memory`, or `exot-san-thread` for the thread sanitiser. Moreover, a Developer can use the provided CMake function `target_enable_sanitiser(target sanitiser)` to enable those for any target. To improve the library user's experience, some helpful messages are printed during configuration. For example, the linked libraries and compile flags are printed out in the console.

To use the library, a developer can use the library repository in their sources, and importing the library target via the `add_subdirectory` CMake function. The submodule mechanism described above can be used to import a specific version of the library to produce applications. The developer can create separate configurations which do not 'pollute' the codebase, by running:

```
cmake -DCMAKE_TOOLCHAIN_FILE=path/to/toolchain.cmake \
      -DCMAKE_BUILD_TYPE=Release -B build/Release -H.
cmake -DCMAKE_TOOLCHAIN_FILE=path/to/toolchain.cmake \
      -DCMAKE_BUILD_TYPE=Debug -B build/Debug -H.
```

All necessary build files and all build artefacts are contained in the directories specified with the `-B` flag. Then, to build separate binaries for a target `exot-target`, the user can navigate to those folders and use the `make` program, or from the project's root run:

```
cmake --build build/Release --target exot-target
cmake --build build/Debug --target exot-target
```

An important feature of the ExOT build process is the addition of toolchain files, which describe the essentials required for compilation. For example, `x86_64-linux-clang-libcxx.cmake` could define a Clang compiler and use *libc++* instead of *stdlibc++* as the standard library. The `android.toolchain.cmake` from the Android's Native Development Kit can also be used as a toolchain. Most of the executables produced using the compilation suite use the LLVM's Clang compiler. It seems superior to GCC in terms of ease of cross-compilation [43, 17][6]. To make the builds reproducible, the compilation suite and the application library use the Nix package manager to isolate the build process from the system. Recently the Nix package manager has also been recognised in the scientific community as a tool to facilitate the reproducibility of experimental software [18].

For testing purposes the modern and fast testing suite *doctest* is used in this project [8]. The build file also provides an executable target `exot-test`, which creates a test runner and combines all individual test suites and test cases from the *test* directory. The executable has a rich calling interface, which allows listing available tests and running specific cases. The creation of unit tests is still an ongoing process.

### 3.1. Cross-compilation and build reproducibility

The Docker-based compilation suite encapsulates all the required software and toolchains necessary for compiling and cross-compiling the library and the applications built on top of it in a docker container. Regardless of which host machine is used the same compiled binaries will be produced. The container can be used very easily by anyone familiar with a command line driven workflow. A special script is used to spawn the container, which mounts the current directory on the host inside it and sets file permissions to the same user and group ID as the calling user. This approach solves the common issue with container-based workflows, where all commands run as the root user, resulting in produced files being inaccessible on the host computer due to insufficient permissions. Preliminary support has been provided for using the environment via an Secure Shell (SSH) connection, for example in an IDE.

## 4. Android applications

As ExOT is designed to allow the integration of many different devices into the measurement environment, we provide utilities for Android devices. In this section, we

[6]The Android project also relies on LLVM's Clang by default.

outline the different applications and wrappers that are implemented as part of ExOT.

### 4.1. Intent proxy

We provide the *intent proxy* service, which acts as an interface between the ExOT experiment engine and any Android app. The intent proxy will scan the extras of each received intent it for keywords such as component or action and will assemble a new intent based on this information, which is forwarded to the application defined in the received intent. The intent proxy also allows us to control the type and keyword of the extras, appended to the forwarded intent. Detailed information on the intent structure can be found in the exot wiki.

As the adb intent interface is limited, the intent proxy allows us to sent all kinds of data to any application running on a smartphone. Therefore, ExOT allows to integrate any application into a measurement setup, and control it via the intent interface of the respective application.

### 4.2. Application wrapper

Based on an Android NDK[7] wrapper, we provide an Android service that allows to encapsulate functionality implemented in the ExOT application library to allow for a fast measurement application development, see Figure 6. In addition, this base service defines an intent interface which allows it to be controlled by the ExOT experiment engine out-of-the-box. The base service can also be integrated into applications with a user interface, to allow interactive measurement campaigns.

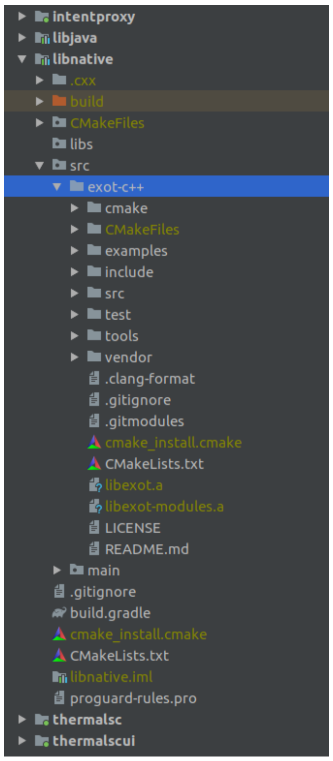
### 4.3. Example apps

We provide two example applications, *thermalsc* and *thermalscui* to illustrate the functionality of the Android integration of ExOT. Both applications measure the utilisation, operating frequency, temperature and current foreground application of the device. thermalsc is a background service that can be integrated into a measurement setup using the ExOT experiment engine, while thermalscui provides a simple user interface for an interactive measurement campaign.

## 5. The experiment engine

In this section we present the experiment engine of ExOT, used for data processing and experiment orchestration. We base our data processing design on a layered information flow model, illustrated in Figure 7. Similar to the well known OSI model, information travels from the highest layer to the lowest, and then up to the highest again.

[7]https://developer.android.com/ndk/



```
▶ intentproxy
▶ libjava
▼ libnative
  ▶ .cxx
  ▶ build
  ▶ CMakeFiles
    libs
  ▼ src
    ▼ exot-c++
      ▶ cmake
      ▶ CMakeFiles
      ▶ examples
      ▶ include
      ▶ src
      ▶ test
      ▶ tools
      ▶ vendor
        .clang-format
        .gitignore
        .gitmodules
        cmake_install.cmake
        CMakeLists.txt
        libexot.a
        libexot-modules.a
        LICENSE
        README.md
    ▶ main
    .gitignore
    build.gradle
    cmake_install.cmake
    CMakeLists.txt
    libnative.iml
    proguard-rules.pro
▶ thermalsc
▶ thermalscui
```

Figure 6: Android integration workspace of ExOT. The C++ application library is integrated as a submodule in the directory “exot-c++”.

| *Layer Name* | *Layer Functions* | |
|---|---|---|
| *6 - Generate/Verify* | generate input | calculate metrics |
| *5 - Source Coding* | bits to symbols | symbols to bits |
| *4 - Line Coding* | symbol to trace | trace to symbols |
| *3 - Raw Data Processing* | output formatting | raw data to trace |
| *2 - I/O Module* | write schedule files | read meas. files |
| *1 - Applications* | system interaction | system observation |
| *0 - System* | system interaction | |

Figure 7: Complete information flow model. Information travels from the highest to the lowest layer, is used to interact with the system and travels up to the highest layer.

Layer 6 describes how input data is generated and how metrics are calculated from the measurement data. In layer 5 and 4, the source and line coding is defined, which is used to compress and shape the data stream depending on the channel specifications. Layer 3 describes the data format required by the applications, while layer 2 defines file I/O. The two bottom layers describe the source (generator) and sink (meter or observer) applications and the system.

Layers 2 to 6 are implemented as Python packages in experiment engine, which has the following advantages: (i) there is no need for recompilation when a new data processing scheme is tested, (ii) the implementation is platform-independent, and (iii) data checks and debugging are easy to perform. The different layers can be combined arbitrarily, increasing the code reusability among different evaluations. Furthermore, experiments can use pass-through layers to skip processing steps or do not use the complete information flow stack. For example, an experiment may only use the stack up to layer 2 by taking static data and convert it to source and the sink application input and configuration files.

### 5.1. Overview of the experiment engine

The simplified Unified Modelling Language (UML) class diagram in Figure 8 shows some of the major relationships in the experiment engine and hints at the functionality enabled in the major components. As before, an instance of an `Experiment` class is responsible for the generation of experiments, data and path management, and instantiation of processing layers.

It is driven by a configuration file, described in Section 5.2. The experiment instance can define multiple experimental phases. Each phase holds a number of *runs*. For example, a phase might include experimental runs for a range of symbol rates. An experimental *run* is the building block of experiments, and encapsulates a number of parameters that remain invariant. These runs are instances of a `Run` class, which is responsible for **ingesting** (processing the input bit stream through the *encode* parts of each of the layers) and **digesting** (processing the measurements from the target platforms through the *decode* parts of each of the layers) the data. They also define how the execution has to be carried out on a target platform.

The experiment engine is typically used as shown in Listing 10. First a config is loaded from a file of the Tom's Obvious, Minimal Language (TOML) format (line 1). The `PerformanceExperiment` instance is created using that configuration (line 2). All required `Run` instances are instantiated by calling the `generate` method (line 3). The run objects are arranged in a dictionary with a tree-like structure with experimental phases as first level children. The tree is accessed via the `phases` property. To perform the encode path processing, the `digest` method is called on all runs—leaves in the hierarchy accessed via `phases` (lines 4-5). The experiment is serialised to disk with the `write` method (line 6). Once it is available as files, it can be executed on a target platform of choice (line 7). The execution involves sending and fetching data as well as orchestrating the execution for each run. Once it completes, the data produced on the target platform is available locally. To read and analyse the data, we perform the decode path processing with the same technique as the encoding (lines 26-27). Ingesting the data requires providing run-time configuration, stored

```
1  configuration = toml.load("./My\ config.toml")
2  experiment =
    ↪ PerformanceExperiment(config=configuration)
3  experiment.generate()
4  for run in
    ↪ datapro.util.misc.leaves(experiment.phases):
5      experiment.digest()
6  experiment.write()
7  experiment.execute_in_environment("My Environment")
8  ingest_arguments = dict(
9      lne={
10         "decision_device":
            ↪ sklearn.pipeline.make_pipeline(
11             sklearn.preprocessing.StandardScaler(),
12             sklearn.svm.LinearSVC(),
13         )
14     },
15     io={
16         "env": "My Environment",
17         "rep": 0,
18         "matcher": datapro.util.wrangle.Matcher(
19             "medium",
20             "type",
21             ["variable"],
22             list(range(0, 8)),
23         ),
24     },
25 )
26 for run in
    ↪ datapro.util.misc.leaves(experiment.phases):
27     experiment.ingest(**ingest_arguments)
28 experiment.write()
29 experiment.backup()
```

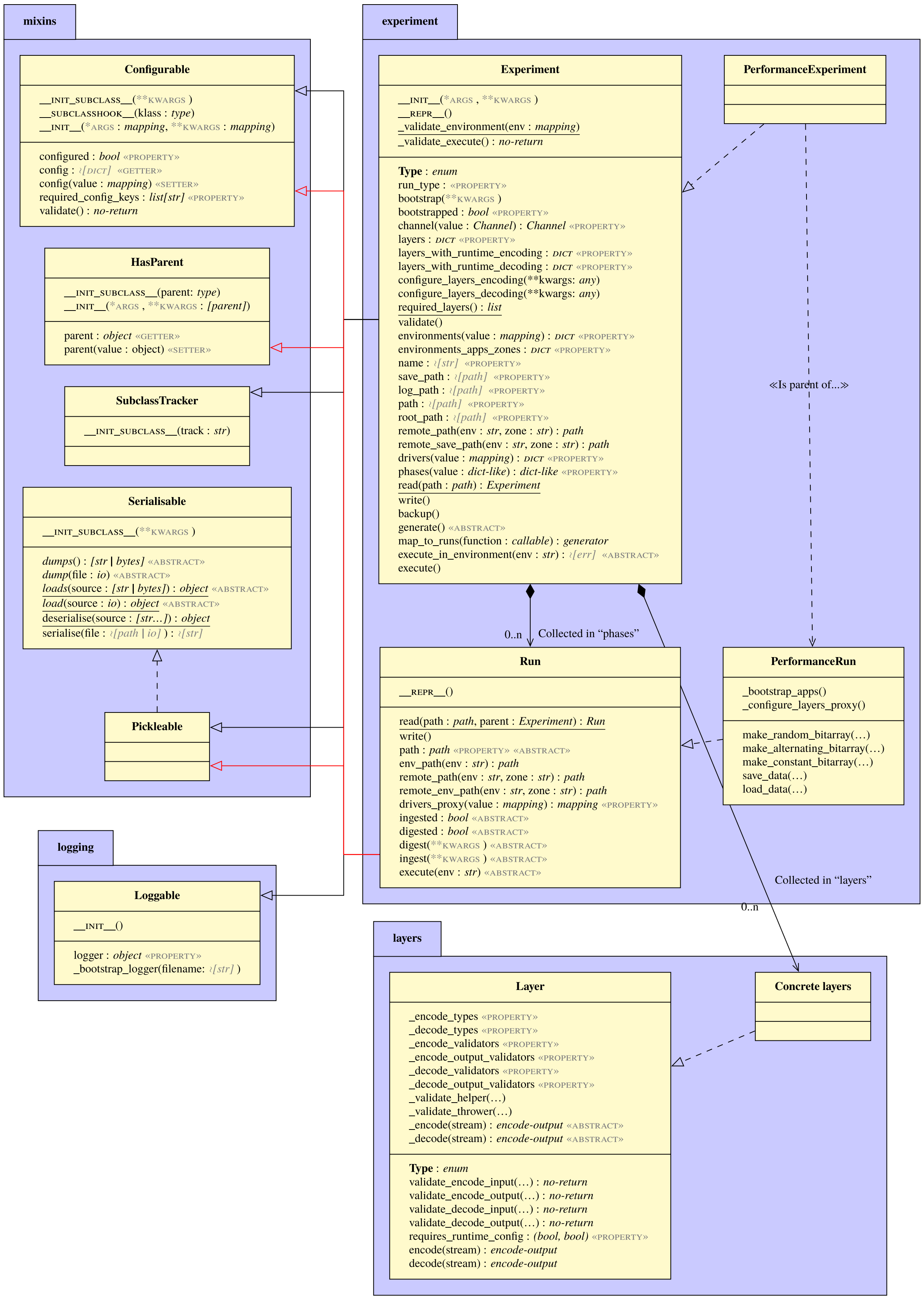
Listing 10: Typical workflow with the experimental experiment engine.

in the `ingest_arguments` variable in this example. Finally, we can serialise the results of our analysis with another call to `write` and upload an archived copy to a backup server.

An instance of the experiment class might also contain ready to use analysis procedures, which are implemented in the channel class. Besides simple local data processing, such analysis procedures might include data processing with machine learning libraries on computing clusters.

### 5.2. Configuration and interoperability

Configuration files for the experiment engine are written in TOML, as it has good support for the required key–value structure, understands a number of well-defined data types (such as arrays, which before required complex structures or fault-prone string splitting), and has good readability. Listing 11 shows how the TOML-based configuration looks like. The structure can be effortlessly converted into other mapping or dictionary-like types, such as JSON or Python's built-in `dict`.



mixins
Configurable
__INIT_SUBCLASS__(**KWARGS )
__SUBCLASSHOOK__(klass : type)
__INIT__(*ARGS : mapping, **KWARGS : mapping)
configured : bool «PROPERTY»
config : ≀[DICT] «GETTER»
config(value : mapping) «SETTER»
required_config_keys : list[str] «PROPERTY»
validate() : no-return
HasParent
__INIT_SUBCLASS__(parent: type)
__INIT__(*ARGS , **KWARGS : [parent])
parent : object «GETTER»
parent(value : object) «SETTER»
SubclassTracker
__INIT_SUBCLASS__(track : str)
Serialisable
__INIT_SUBCLASS__(**KWARGS )
dumps() : [str | bytes] «ABSTRACT»
dump(file : io) «ABSTRACT»
loads(source : [str | bytes]) : object «ABSTRACT»
load(source : io) : object «ABSTRACT»
deserialise(source : [str...]) : object
serialise(file : ≀[path | io] ) : ≀[str]
Pickleable
logging
Loggable
__INIT__()
logger : object «PROPERTY»
_bootstrap_logger(filename: ≀[str] )
experiment
Experiment
__INIT__(*ARGS , **KWARGS )
__REPR__()
_validate_environment(env : mapping)
_validate_execute() : no-return
Type : enum
run_type : «PROPERTY»
bootstrap(**KWARGS )
bootstrapped : bool «PROPERTY»
channel(value : Channel) : Channel «PROPERTY»
layers : DICT «PROPERTY»
layers_with_runtime_encoding : DICT «PROPERTY»
layers_with_runtime_decoding : DICT «PROPERTY»
configure_layers_encoding(**kwargs: any)
configure_layers_decoding(**kwargs: any)
required_layers() : list
validate()
environments(value : mapping) : DICT «PROPERTY»
environments_apps_zones : DICT «PROPERTY»
name : ≀[str] «PROPERTY»
save_path : ≀[path] «PROPERTY»
log_path : ≀[path] «PROPERTY»
path : ≀[path] «PROPERTY»
root_path : ≀[path] «PROPERTY»
remote_path(env : str, zone : str) : path
remote_save_path(env : str, zone : str) : path
drivers(value : mapping) : DICT «PROPERTY»
phases(value : dict-like) : dict-like «PROPERTY»
read(path : path) : Experiment
write()
backup()
generate() «ABSTRACT»
map_to_runs(function : callable) : generator
execute_in_environment(env : str) : ≀[err] «ABSTRACT»
execute()
PerformanceExperiment
«Is parent of...»
0..n
Collected in "phases"
Run
__REPR__()
read(path : path, parent : Experiment) : Run
write()
path : path «PROPERTY» «ABSTRACT»
env_path(env : str) : path
remote_path(env : str, zone : str) : path
remote_env_path(env : str, zone : str) : path
drivers_proxy(value : mapping) : mapping «PROPERTY»
ingested : bool «ABSTRACT»
digested : bool «ABSTRACT»
digest(**KWARGS ) «ABSTRACT»
ingest(**KWARGS ) «ABSTRACT»
execute(env : str) «ABSTRACT»
PerformanceRun
_bootstrap_apps()
_configure_layers_proxy()
make_random_bitarray(...)
make_alternating_bitarray(...)
make_constant_bitarray(...)
save_data(...)
load_data(...)
Collected in "layers"
0..n
layers
Layer
_encode_types «PROPERTY»
_decode_types «PROPERTY»
_encode_validators «PROPERTY»
_encode_output_validators «PROPERTY»
_decode_validators «PROPERTY»
_decode_output_validators «PROPERTY»
_validate_helper(...)
_validate_thrower(...)
_encode(stream) : encode-output «ABSTRACT»
_decode(stream) : encode-output «ABSTRACT»
Type : enum
validate_encode_input(...) : no-return
validate_encode_output(...) : no-return
validate_decode_input(...) : no-return
validate_decode_output(...) : no-return
requires_runtime_config : (bool, bool) «PROPERTY»
encode(stream) : encode-output
decode(stream) : encode-output
Concrete layers

Figure 8: UML diagram of some of the relationships between software components of the data processing and experimental orchestration framework.

```
name = "Report demo"
save_path = "data"
backup_path = "data/_backup"
experiment_exists_action = "move"

[EXPERIMENT]
type = "PerformanceExperiment"
channel = "Cache"

[EXPERIMENT.PHASES]
train = {bit_count = 10000, symbol_rates = [1000, 10000], repetitions = 5}
eval = {bit_count = 10000, symbol_rates = [1000, 10000], repetitions = 5}

[EXPERIMENT.LAYERS]
src = {name = "Huffman", params = { length = 4 }}
lne = {name = "GenericLineCoding", params = { saturated = false, demean = false }}
rdp = {name = "DirectActivation", params = { sync_pulse_duration = 0.5, sync_pulse_detection = "falling" }}
io = {name = "TimeValue", params = {output_timing_multiplier = 1e9, input_timing_multiplier = 1e9}}

[EXPERIMENT.GENERAL]
latency = 10
fan = true
governors = "userspace"
frequencies = "max"
sampling_period = 3e-6

[APPS]
snk = {executable = "meter_cache_fr", zone = "insecure"}
src = {executable = "generator_cache_read_st", zone = "insecure"}
```

Listing 11: An excerpt of a the experiment engine configuration file.

### 5.2.1. *Interface to applications*

Listing 12 shows an example of an actual source application configuration. We can clearly see that passing this number of parameters via command line arguments would be inconvenient and prone to errors. Each configurable component accesses its options in the JSON object specified by its name (as defined on line 4 in Listing 7). In the experiment engine these JSON configuration objects are generated directly from values specified in the experiment configuration file, as shown in Listings 13 and 14. This allows the master configuration file to encompass all necessary settings in a single file.

```
{
  "generator": {
    "cores": [0], "worker_policy": "round_robin", "self_policy": "round_robin",
    "worker_priority": 98, "self_priority": 97, "use_busy_sleep": true,
    "busy_sleep_yield": true, "shm_file": "/dev/hugepages/8",
    "set_count": 2, "set_increment": 64
  },
  "schedule_reader": {
    "input_file": null, "read_as_hex": false, "reading_from_file": false
  },
  "logging": {
    "append_governor_to_files": false, "async": false, "async_size": 8192,
    "log_level": "trace", "provide_platform_identification": false,
    "debug_log_filename": null, "app_log_filename": null,
    "timestamp_files": true
  }
}
```

Listing 12: New JSON-based configuration format

```
[ENVIRONMENTS."Environment name".src]
generator.host_pinning = 3
generator.should_pin_host = true
generator.cores = [ 0 ]
generator.should_pin_workers = true
generator.host_policy = "round_robin"
generator.host_priority = 97
generator.worker_policy = "round_robin"
generator.worker_priority = 98
generator.use_busy_sleep = true
generator.busy_sleep_yield = false
generator.use_huge_pages = true
generator.shm_file = "/dev/hugepages/8"
generator.set_count = 64
generator.set_increment = 64

logging.debug_log_filename = ""
logging.app_log_filename = ""
logging.log_level = "info"
logging.provide_platform_identification = false
logging.async = false

schedule_reader.input_file = ""
schedule_reader.reading_from_file = true
schedule_reader.cpu_to_pin = 1
```

Listing 13: Source application config

## 5.3. Experiment engine implementation

In this subsection, we briefly explain the implementation strategies applied to the experiment engine of ExOT.

### 5.3.1. *Python modularity, documentation and typing support*

This allows the experiment engine to be used in a more consistent manner and makes it easier to import and use

```
[ENVIRONMENTS."Environment name".snk]
logging.append_governor_to_files        = false
logging.async                           = true
logging.async_size                      = 4096
logging.log_level                       = "debug"
logging.provide_platform_identification = true
logging.timestamp_files                 = false
logging.rotating_logs       = false
logging.rotating_logs_count = 10
logging.rotating_logs_size  = 104857600

meter.host_policy       = "round_robin"
meter.host_pinning      = 7
meter.should_pin_host   = true
meter.host_priority     = 95
meter.log_header        = true
meter.start_immediately = false
meter.use_busy_sleep    = true
meter.busy_sleep_yield  = false

cache.use_huge_pages = true
cache.shm_file       = "/dev/hugepages/8"
cache.set_count      = 64
cache.set_increment  = 64
```

Listing 14: Sink application config

in an IDE or an interactive environment. **??** and shows an example of the suggestions displayed by a text editor or an IDE when one tries to import a namespace or module from the top-level `datapro` package.

Modern Python adds the support for type annotations, which provides well-defined function signatures and allows static type checkers, such as MYPY[8], to be used. Thanks to the type hints the intent of each function is much clearer to the user.

Supplements typing with documentation in the standard “docstring” format conforming to Python Enhancement Proposal (PEP) 257[9]. This style of documentation is much easier to parse by tools such as text editors, as shown in Figure 9 and Figure 10.

### 5.3.2. *Defensive programming*

Multiple value and type checks are used throughout the experiment engine, preventing the misuse of the provided software and helps to ensure the correctness of the execution. If wrong values or types are supplied, the user is notified in a quick and informative manner.

### 5.3.3. *Mix-in classes*

These classes are are lightweight classes that are meant to be inherited from, but which provide limited and generic functionality. These improve universality, contain certain generic functionality in a single unit, and help reduce code duplication.

[8]http://mypy-lang.org
[9]https://www.python.org/dev/peps/pep-0257/

### 5.3.4. *Config-driven class instantiation*

The experiment engine takes advantage of abstract base classes and provides a subclass tracker mix-in and a generic object factory. The mix-ins are enabled by the simpler customisation of class creation introduced in PEP 487[10], and define custom `__init_subclass__` class methods. The subclass tracker traverses the method resolution order and allows a base class to keep track of derived classes. The generic object factory uses the information provided by the subclass tracker and provides an interface for creation of non-abstract subclass instances. The code example below shows how little effor is required to provide a factory for a user-defined class hierarchy:

```python
class Base(SubclassTracker,
    track="customisation_point",
    metaclass=abc.ABCMeta): pass
class Derived(Base,
    customisation_point=SomeEnum.SomeValue): pass
class Factory(GenericFactory, klass=Base): pass

instance = Factory()("Derived")
```

In this example, a base class inherits from `SubclassTracker`. The `GenericFactory` is provided the base class in its `klass` parameter. The derived class is readily available in the factory, and can be created using its name. These factories are used in conjunction with the experiment config file.

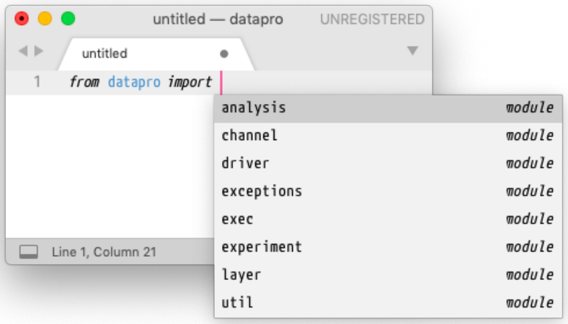
### 5.3.5. *Dependency management*

The experiment engine uses the *pyproject.toml* file that has been proposed in PEP 518[11], which is increasingly used to specify build system requirements for Python projects. The control over the dependencies is achieved using the POETRY packaging and dependency manager[12]. Using a simple syntax (e.g. `numpy = "^1.16"`) one can declare which version of a dependency needs to be installed. A lock file, which contains all dependencies and their versions, is created and can be easily versioned. The control over the Python version is exercised with the popular PYENV project[13].

### 5.3.6. *Enhances the inspection and control of platform parameters*

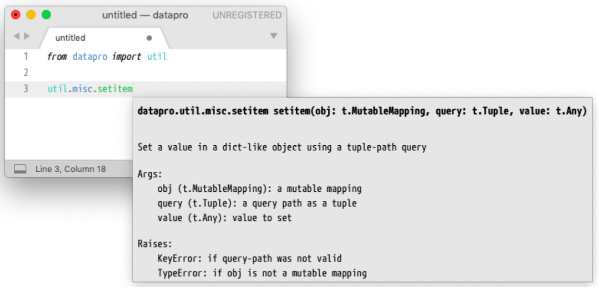
The experiment engine makes it possible to both inspect and set the state of the target platform. Thanks to the ability to read platform settings, the experiment engine also validates whether the provided values are available or suitable. Moreover, the experiment engine has better knowledge of the original state and can more easily restore it after the experiment is completed.

[10]https://www.python.org/dev/peps/pep-0487/
[11]https://www.python.org/dev/peps/pep-0518/
[12]https://poetry.eustace.io
[13]https://github.com/pyenv/pyenv



```
from datapro import
```

Figure 9: Module importing suggestions



```
from datapro import util

util.misc.setitem
```

```
datapro.util.misc.setitem setitem(obj: t.MutableMapping, query: t.Tuple, value: t.Any)


Set a value in a dict-like object using a tuple-path query

Args:
    obj (t.MutableMapping): a mutable mapping
    query (t.Tuple): a query path as a tuple
    value (t.Any): value to set

Raises:
    KeyError: if query-path was not valid
    TypeError: if obj is not a mutable mapping
```

Figure 10: Function signature and documentation hints

## 6. Showcase

In this section, we present small examples of the usage of different ExOT components, that illustrate the capabilities of the toolkit. For more extensive usage see Miedl [1].

### 6.1. Application library

Before presenting an example for using the application library, let us look at how an actual application is composed using the library. The applications are written in quite a declarative style, as the example in Listing 15 shows. The typical order is as follows:

(2) Declare type aliases for the sake of convenience, with the help of the `using` statement. Aliases are indispensable for class templates, variadic in particular. In the shown example, one meter modules is provided to the meter host component. Using a subtype contained in the alias `meter_type`, we then provide the right template parameter (time) to the logger.

(4) Instantiate the command line wrapper which will in itself instantiate the executor and spawn component processes.

A number of concrete applications were created using the application library. They all share the same structure as the one shown in Listing 15, and include a

```
#include <chrono>
#include <exot/components/meter_host_logger.h>
#include <exot/meters/thermal_msr.h>
#include <exot/utilities/main.h>

using namespace exot;
using meter_t = components::meter_host_logger<          (2)
                  std::chrono::nanoseconds,
                ↪     modules::thermal_msr>;          (2.2)

int main(int argc, char** argv) {
  return utilities::cli_wrapper<meter_t>(argc, argv);
  ↪   (4.0)
}
```

Listing 15: A sample application

source application, and various meter configurations[14]. An experimental data set was produced using the multithreaded utilisation generator, which was pinned to cores 1, 3, 5, and 7 of a Lenovo T440p Laptop, based on an Intel i7-4700MQ core.

Two examples of logged data are visualised in Figures 11 and 12. The former presents an excerpt produced with a meter that combines thermal and power information accessed via model specific registers, and the latter shows a combination of multiple available meter modules in a single log output. The red dashed vertical lines are produced from the load generator's application log, and indicate the onset and end of high utilisation states, respectively. The presented graphs have been post-processed with a 5-sample moving average filter, to improve the clarity of the output[15]. As the graphs show, the change in measured states coincides with the changes in the input state trace. Moreover, Figure 12 shows how the meter host (Section 2.7.6) could be used for a more exploratory analysis, where individual modules can be cross-referenced.

### 6.2. Experimental flow

As an example for the use of the experimental flow, we show a simple experiment with the thermal covert channel shown by Bartolini, Miedl and Thiele [24].

Figure 13 shows examples of temperature traces gathered from "Haswell" (Lenovo T440*p*) and "ARM" (Raspberry Pi 3) platforms, produced by execution traces to transmit data with bit rates of 50 and 5 bits per second, respectively. The dotted lines show boundaries between the line-coded symbols. Each contains a transition from either lower to higher or higher to lower temperature (for message bits 0 and 1, respectively). Depending on the symbol rate, each line-coded symbol can be represented

[14]In the future, a meter factory class may be provided to set up meter modules at runtime.

[15]In particular, the averaging was performed to alleviate the issue with MSR access as described by Jón Thoroddsen [21].

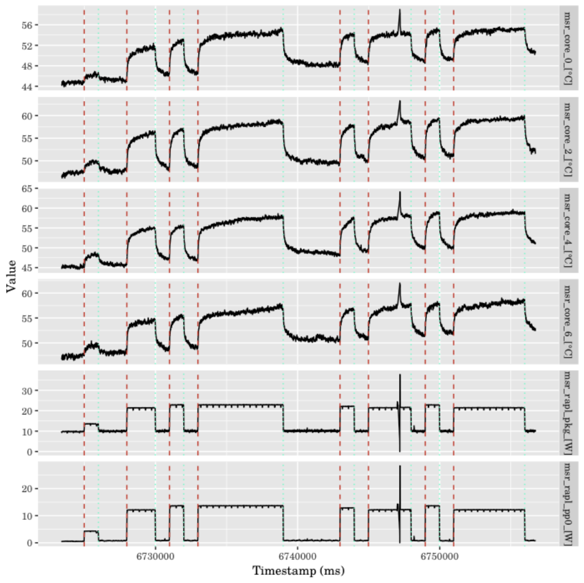


Figure 11: Showcase of a typical experimental run

by a number of samples. The decoding of the input trace follows the strategy described by the authors [24, sec. 7.1]. A Gaussian naive-Bayes classifier operates on slices of the input trace transformed into complex plane. The complex plane representation is produced with the help of a 0°and 90°phase-shifted carrier signals. The slices are de-meaned to remove long-term temperature variations. The corresponding symbol spaces with annotated classifier's decision boundaries are shown in Figure 14 for the two platforms.

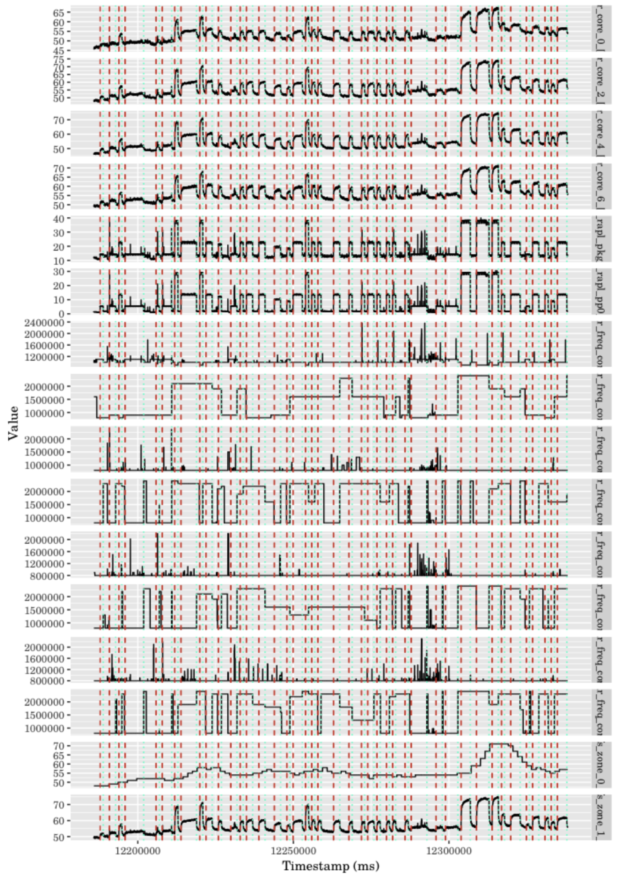


Figure 12: A meter using multiple meter modules From top to bottom, the displayed variables are: core temperatures for cores 0, 2, 4, and 6; power in RAPL domains PKG and PP0; scaling frequencies of cores 0 to 7; thermal information from `sysfs` thermal zones 0 and 1.

## 7. Future work

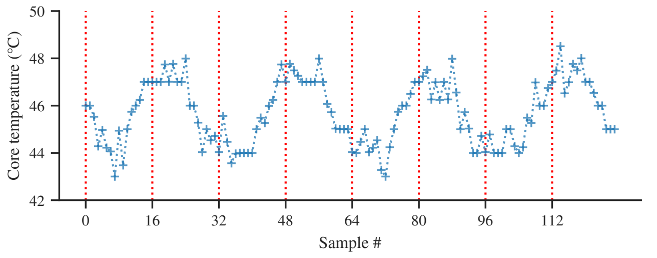
The following sections describe the possible future work classified into functional enhancements, extensions and improvements, and software engineering refinements. Future directions for the application library development include:

**Exploration** Since a large part of the research work involves informal exploration of the side-effects of execution of programs, it might be helpful to provide a more direct, perceptible indication of acquired meter measurements. A pipeline component or a script could be provided to visualise the accumulating readings, using graphical or terminal-based output. Moreover, exploratory visualisation could be provided for the Android platform, possibly taking advantage of existing example code which plots sensor data using OpenGL primitives.

**New paradigms** Implementing the *task-based model of execution* would allow for a broader variety of programming styles and allow for greater sharing of resources among asynchronous components, which could be executed by a thread pool and avoid expensive thread context switches.

**Executors** The executors in the current implementation are quite basic, and provide only a thin layer of abstraction over lower-level system or user-space threads. Further possibilities exist, including loop executors and thread pool executors [see 29].

**Communication between components** At the moment the queues used for communication are concurrent, but there is likely room for improvement in terms of performance. Additional focus could be given to *zero-copy mechanisms* in order to verify that the implemented move operations (or copy elision/return value optimisation) actually prevent copies being created. It also might be the case that there are leaner synchronisation mechanisms than the currently used combination of locks and condition variables. Moreover, using a more hand-crafted storage method (e.g., using a circular buffer) rather than



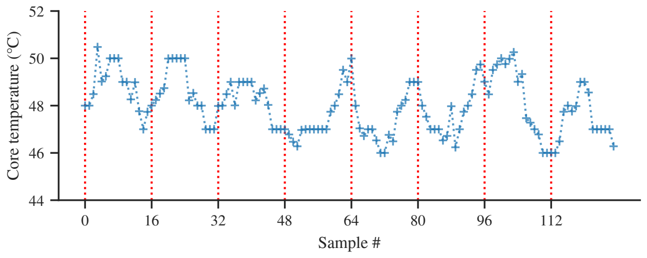
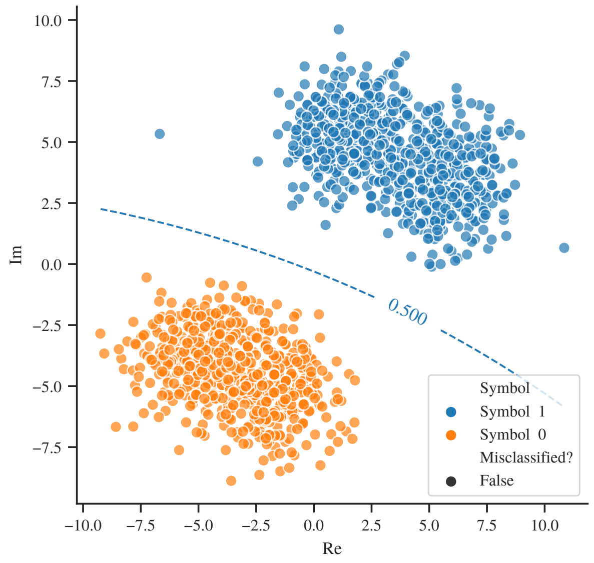


Figure 13: Example transmissions with the thermal channel for Haswell at 50 bps (left) and ARM at 5 bps (right).



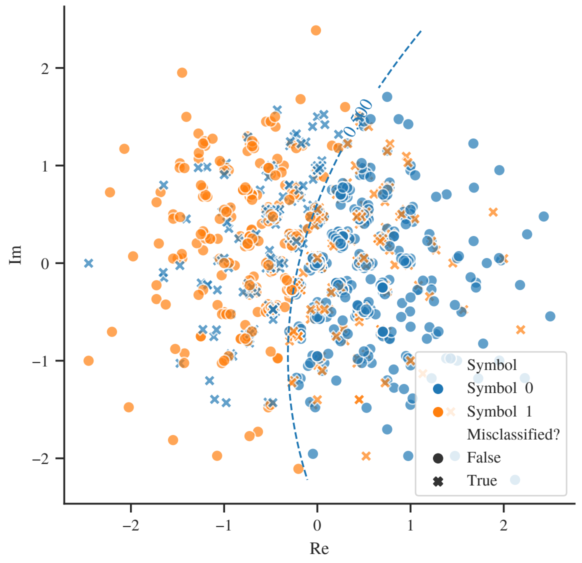


Figure 14: Example symbol spaces of the decoded thermal channel transmissions for Haswell at 50 bps (left) and ARM at 5 bps (right).

wrapping a queue class from the STL could deliver better performance, or provide bulk/stride access to queue elements.

Implementing a control flow in addition to data flow might also allow for better collaboration between components; for example, in the GNU Radio project the ability to exchange control messages between nodes allows the use of push/pull message passing semantics (instead of the push-only model of classical process networks).

**Deadlocks** Currently deadlocks may only arise from improper ordering of functional code and state changes inside the process network nodes. In the case of a pipeline arrangement of nodes there is little chance for deadlocks, both global and local (due to insufficient queue capacities). If the library was to be extended with multiple inputs and outputs, a proper deadlock detection and resolution mechanism would need to be provided. An observer could monitor and resize the queues, but the process nodes might also need to be instrumented with an execution state and thread-safe access to it. Successful mechanisms have been developed by Allen, Zucknick and Evans [36] and Geilen and Basten [38]. At the moment, the extended semantics of `try_{read,write}_{for,until}` allow for handling of local deadlocks without any impact to token ordering, and may even prove sufficient for more complex use cases.

**Extended read/write interface** The interface to queues/channels that allows the "try" semantics should ideally exist as an interface class, such that a user can extend the core framework by deriving from it and implementing the member functions. However, the functions are template member functions, which cannot be made virtual.[16] Using a different mechanism than inheritance might be necessary.

**Duplicated MSR access objects** Each meter that needs

[16]"Member template functions cannot be declared virtual. Current compiler technology expects to be able to determine the size of a class's virtual function table when the class is parsed."[37, p. 242]

Figure 15: Left: Experiment Orchestration Toolkit (ExOT) logo; Right: QR-Code linking to the Experiment Orchestration Toolkit (ExOT) website

access to model specific registers has its own instance of the `MSR` class, resulting in duplication of functionality and increased count of file descriptors opened by the process. It might be a good idea to make the `MSR` a singleton class, and allow users to limit which registers they want to access.

**State semantics** Improving the state management semantics, and holding the state in an enumeration instead of atomic boolean variables.

Future directions for the extension of the experiment engine might include:

**Plotting and data visualisation** The plotting and data visualisation module of the experiment engine is very rudimentary. Therefore, an extension of this functionality using modern python plotting libraries would be recommendable to improve the usability of the experiment engine.

**Automated data backup and restore** While an automated backup and restore mechanism for experimental data was already foreseen in the initial design, it was never implemented. This mechanism should allow to backup and restore experimental data to a (remote) location defined in the experiment configuration file.

## 8. Concluding Remarks

In this paper we presented the implementation details and underlying design decision for the Experiment Orchestration Toolkit (ExOT). We gave a detailed overview of the design patterns of the application library, the Android integration and the experiment engine of ExOT.

For further information, consult the ExOT website (exot.ethz.ch) or the ExOT wiki (https://gitlab.ethz.ch/tec/public/exot/wiki/-/wikis/home). ExOT was developed in the Computer Engineering Group, which is part of the Computer Engineering and Networks Laboratory at ETH Zürich. All components of the ExOT project are publicly available under the 3-clause BSD license.

## Acknowledgements

Thanks to all students and colleagues at the Computer Engineering Laboratory, which have used Experiment Orchestration Toolkit (ExOT) in their projects and helped to improve it by providing feedback. Moreover, thanks to Lukas Sigrist and the RocketLogger Team for providing the website template and giving advice during the process of open sourcing.

Last but not least, we also want to thank Azra Gradinčić the design of the ExOT logo shown in Figure 15.

## References

[1] P. Miedl, 'Threat potential assessment of power management related data leaks', PhD thesis, ETH Zurich, 2020.

[2] P. Miedl, R. Ahmed and L. Thiele, 'We know what you're doing! Application detection using thermal data', *Leibniz Transactions on Embedded Systems*, vol. Special Issue on Embedded System Security, no. 1, 2020, Under review.

[3] P. Miedl, B. Klopott and L. Thiele, 'Increased reproducibility and comparability of data leak evaluations using ExOT', in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2020. DOI: 10.3929/ethz-b-000377986. [Online]. Available: https://doi.org/10.3929/ethz-b-000377986.

[4] A. Fitsios, 'Towards Task Inference on Mobile Systems based on Thermal Traces', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, Mar. 2019, Semester Thesis; Supervisors: Philipp Miedl, Rehan Ahmed and Lothar Thiele.

[5] B. Klopott, 'How bad are data leaks really?', Supervisors: Philipp Miedl and Lothar Thiele, Master's thesis, ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Jun. 2019.

[6] N. Lohmann. (2019). nlohmann/json, [Online]. Available: https://github.com/nlohmann/json.

[7] C. Barth, 'Come again? Towards repeatable security experiments', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, Aug. 2018, Semester Thesis; Supervisors: Philipp Miedl and Lothar Thiele.

[8] V. Kirilov, 'onqtam/doctest', 2018. [Online]. Available: https://github.com/onqtam/doctest.

[9] B. Klopott, 'You also want to explore other security leaks? Building an easily extendable application library for security leak research', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, Aug. 2018, Semester Thesis; Supervisors: Philipp Miedl and Lothar Thiele.

[10] A. Mackintosh, *C++ Latches and Barriers*, ISO/IEC JTC1 SC22 WG21, May 2018.

[11] M. Meier, 'Feature Extraction from Thermal Traces for the Thermal Fingerprinting Attack', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, May 2018, Semester Thesis; Supervisors: Philipp Miedl, Rehan Ahmed and Lothar Thiele.

[12] G. Melman, 'spdlog', 2018. [Online]. Available: https://github.com/gabime/spdlog.

[13] P. Miedl, X. He, M. Meyer, D. B. Bartolini and L. Thiele, 'Frequency Scaling as a Security Threat on Multicore Systems', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2497–2508, Nov. 2018, ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2857038. [Online]. Available: https://doi.org/10.1109/TCAD.2018.2857038.

[14] P. Miedl and L. Thiele, 'The Security Risks of Power Measurements in Multicores', in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18, Pau, France: Association for Computing Machinery, 2018, pp. 1585–1592, ISBN: 978-1-45035-191-1. DOI: 10.1145/3167132.3167301. [Online]. Available: https://doi.org/10.1145/3167132.3167301.

[15] M. Millen, 'Analysis and Optimization of Frequency Governors', Supervisors: Rehan Ahmed, Philipp Miedl and Lothar Thiele, Master's thesis, ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Apr. 2018.

[16] 'muellan/clipp', 2018. [Online]. Available: https://github.com/muellan/clipp.

[17] P. Smith, 'How to cross compile with LLVM based tools', in *FOSDEM'18*, Linaro, Feb. 2018, pp. 1–28. [Online]. Available: https://fosdem.org/2018/schedule/event/crosscompile/.

[18] B. Bzeznik, O. Henriot, V. Reis, O. Richard and L. Tavard, 'Nix as HPC package management system', in *HUST 2017*, Nov. 2017, pp. 1–24.

[19] C++ Technical Committee, *ISO/IEC 14882:2017*, 2017. [Online]. Available: http://www.eel.is/c++draft/.

[20] X. He, 'A Smart Attack using the Frequency Covert Channel', Supervisors: Philipp Miedl, Matthias Meyer and Lothar Thiele, Master's thesis, ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Oct. 2017.

[21] Ólafur Jón Thoroddsen, 'UnCovert 4: The Power Covert Channel', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, Jun. 2017, Semester Thesis; Supervisors: Philipp Miedl and Lothar Thiele.

[22] M. Selber, 'UnCovert3: Covert Channel Attacks on Commerical Multicore Systems', Supervisors: Philipp Miedl and Lothar Thiele, Master's thesis, ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Apr. 2017.

[23] R. Strebel, 'What is my Thermal Fingerprint?', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, Jul. 2017, Semester Thesis; Supervisors: Philipp Miedl, Rehan Ahmed and Lothar Thiele.

[24] D. B. Bartolini, P. Miedl and L. Thiele, 'On the Capacity of Thermal Covert Channels in Multicores', in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, London, United Kingdom: ACM, 2016, 24:1–24:16, ISBN: 978-1-45034-240-7. DOI: 10.1145/2901318.2901322. [Online]. Available: http://doi.acm.org/10.1145/2901318.2901322.

[25] M. Selber, 'UnCovert: Operating Frequency, a Security Leak?', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, Feb. 2016, Semester Thesis; Supervisors: Philipp Miedl and Lothar Thiele.

[26] P. Wild, 'UnCovert: Evaluating thermal covert channels on Android systems', ETH Zürich, Gloriastrasse 35, 8092 Zürich, Switzerland, Tech. Rep. 1, Aug. 2016, Semester Thesis; Supervisors: Philipp Miedl and Lothar Thiele.

[27] J. C. Beard, P. Li and R. D. Chamberlain, 'RaftLib', in *the Sixth International Workshop*, New York, New York, USA: ACM Press, 2015, pp. 96–105, ISBN: 978-1-45033-404-4. DOI: 10.1145/2712386.2712400. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2712386.2712400.

[28] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen and A. Francillon, 'Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters.', English, *RAID*, vol. 9404, no. Chapter 3, pp. 48–65, 2015. DOI: 10.1007/978-3-319-26362-5_3. [Online]. Available: http://link.springer.com/10.1007/978-3-319-26362-5_3.

[29] C. Mysen, *Executors and schedulers*, ISO/IEC JTC1 SC22 WG21, Apr. 2015.

[30] M. Torquati, *Parallel Programming Using FastFlow*, September 2015, University of Pisa, Sep. 2015.

[31] A. Kukanov, V. Polin and M. J. Voss, 'Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks', Intel Corporation, Tech. Rep., Jun. 2014.

[32] G. E. Allen, 'Computational process networks', English, PhD thesis, University of Texas at Austin, Austin, May 2011. [Online]. Available: https://repositories.lib.utexas.edu/handle/2152/ETD-UT-2011-05-2987.

[33] P. Smith, *Software Build Systems*, ser. Principles and Experience. Addison-Wesley, 2011.

[34] A. Podobas, M. Brorsson and K.-F. Faxén, 'A Comparison of some recent Task-based Parallel Programming Models', English, in *3rd Workshop on Programmability Issues for Multi-Core Computers*, 2010. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-23671.

[35] Z. Vrba, 'Implementation and performance aspects of Kahn process networks', PhD thesis, Jul. 2009. [Online]. Available: https://dblp.org/rec/phd/basesearch/Vrba09.

[36] G. E. Allen, P. E. Zucknick and B. L. Evans, 'A Distributed Deadlock Detection and Resolution Algorithm for Process Networks', in *2007 IEEE International Conference on Acoustics, Speech, and Signal Processing*, IEEE, 2007, pp. II–33–II–36, ISBN: XXX-1-4244-0727-3. DOI: 10.1109/ICASSP.2007.366165. [Online]. Available: http://ieeexplore.ieee.org/document/4217338/.

[37] B. Eckel and C. D. Allison, *Thinking in C++*, English, ser. Practical Programming. Prentice Hall, Dec. 2003, vol. 2, ISBN: XXX-0-13-035313-2. [Online]. Available: http://www.cs.ust.hk/~dekai/library/ECKEL_Bruce/.

[38] M. Geilen and T. Basten, 'Requirements on the Execution of Kahn Process Networks.', *ESOP*, vol. 2618, 2003. DOI: 10.1007/3-540-36575-3_22. [Online]. Available: http://link.springer.com/10.1007/3-540-36575-3_22.

[39] H. Van Der Linden, 'Scheduling distributed Kahn process networks in Yapi', PhD thesis, Technische Universiteit Eindhoven, 2003.

[40] A. Alexandrescu, *Modern C++ Design*, ser. Generic Programming and Design Patterns Applied. Addison Wesley, Feb. 2001, ISBN: 978-0-20170-431-5. [Online]. Available: http://www.worldcat.org/title/c-in-depth/oclc/316330731.

[41] M. Goel, 'Process Networks in Ptolemy II', PhD thesis, Berkeley, CA, Dec. 1998.

[42] T. M. Parks, 'Bounded Scheduling of Process Networks', PhD thesis, Dec. 1995. [Online]. Available: http://ptolemy.eecs.berkeley.edu/papers/parksThesis.

[43] J. Roelofs, 'Which targets does Clang support?', in *EuroLLVM 2014*, pp. 1–15. [Online]. Available: https://llvm.org/devmtg/2014-04/.