$\mathcal{S}$ide-channel $\mathcal{M}$atters: a $\mathcal{U}$niversal po$\mathcal{R}$table $\mathcal{F}$ramework
The $\mathcal{SMURF}$ Library

Yan Yan

November 2, 2022

Abstract

$\mathcal{SMURF}$ is a framework/library for leakage simulator development.

Contents

Not finalised yet...

Chapter 1

Introduction

1.1 Leakage simulator

Leakage simulators are simulators enhanced with the functionality of producing leakage traces. They take as input cryptographic implementations cross compiled into binary images on the target platforms, and output leakage traces by applying artificial leakage models to the simulation.

Leakage simulators are very useful tools for early stage side channel evaluations as an alternative to real devices. Specifically, they have the advantages of:

- **Low cost** There is no need for real hardware and thus the manufacturing cost are saved. It also made it possible for massive leakage trace generation in parallel, as much as the computational power is affordable.
- **Flexibility** The users can customise the leakage simulator according to the needs, particularly when the leakage simulator is open sourced. For instance, it may allow any chosen fault to be injected at any point during the execution which, in contrast, is very difficult and costly to be achieve on real devices.
- **Better explanatory** Since the leakage are simulated its components can be broken down easily under a known leakage model. This particularly useful in analysing the causes when leakages are detected. This is rather a difficult task for real devices due to physical constrains including synchronisation and measurement errors, especially under a non invasive setting.

On the other hand, it must be noted that leakage simulator has its constrains. Above all, the quality of a leakage simulator is determined by the quality of the leakage model it employed, which is often judged by its representativeness towards physical devices. Hence certain degree of error is inevitable and it is an active research topic improving the modelling methods to mitigate such error. In reality, leakage simulators also must face the problem black box modelling,

i.e. the leakage models are modelled by third parties that does not necessarily have access to the exact hardware design. This is typically seen on commercial products since their hardware designs are often propitiatory.

Examples of leakage simulator A simplest example is the Hamming weight simulators which are frequently seen in side channel literatures. Its leakage at each time point is simulated simply by summing the Hamming weights of all operands. The popularity of Hamming weight could be traced back to the earliest side channel literatures such as [6] due to its empirical successful application on real devices and it has lasted even until today. Alternatively, Hamming distance has also been used in a similar manner when people trying to capture transitional leakage between two intermediates [6].

However, as the semiconductor technology evolves over the decades, Hamming weight can now hardly be seen as a well captured leakage model for the latest devices. Particularly due to the fact that none of the microarchitecture factors has been taken into account in the Hamming weight model whereas in reality they have been shown to be significantly contributing to the leakage. The ELMO [9] project is a remarkable example that addressed this issue by introducing an abstracted ELMO architecture that consists an ALU and pipeline. The ELMO architecture enables the leakage to be depicted with a better granularity since the leakage model is taking more detailed intermediates rather than simply the operands. ELMO also proposed the idea of depicting the leakage model as a linear function of the intermediates with coefficients derived from regressing the real traces. As a result, ELMO traces are more realistic and hence better leakage coverage than simply Hamming weight and Hamming distances. ELMO has later spawned the GILES[1] which additionally supports fault injection.

A further step down the line is the μ-ELMOproject [7, 8]. Comparing with its predecessor, μ-ELMOis built on a more realistic architecture rather than an abstraction as did by ELMO which allows leakage model to be defined with further realisticness. Since more intermediates are captured during the simulation, μ-ELMOalso creates a better potential of portability for the proving tools used by provable side channel analysis (more details in Section 1.2). However, there are certain practicability flaws in μ-ELMO. First of all, having access to the architectural information is generally a difficult task, especially for commercial products where the designs are proprietary. Secondly, it is also practically difficult to include every component in a processor into the leakage model due to its complexity. In case of μ-ELMO, the architecture was derived from reverse engineering the device [8] and then a selection process (details in [7]) is needed to produce a set of architectural components that are significant to leakage.

1.2 Provable side channel and the proving tools

[I need input from Ines on this topic...]

1.3 The $\mathcal{SMURF}$ solution

1.3.1 History: from ELMO to μ-ELMO

The initiative of $\mathcal{SMURF}$ stems from the ELMO [9] project. The goal was to provide a proof of concept for building a more sophisticated leakage simulator that captures architectural leakage context. The code was developed based on the open sourced Thumbulator [2]. As a prototype of its kind, ELMO nicely simulated the leakage of its targeted device which is an ARM Cortex-M0 on a STM32F0 board.

The ELMO leakage simulator has later been restructured into GILES [1], which additionally supported fault injection simulation. The decision to restructure ELMO was made due to the fact that the code was highly entangled with Thumbulator and lack comprehensible extensibility. The restructured GILES takes a configuration file in JSON format for coefficients in the leakage model which was hardcoded in the original ELMO. Also as a side project, GILES additionally provides the functionality of fault injection simulation.

Although GILES was intended to have a generic framework that could be reused for further development of other leakage simulators, it was later found unsatisfactorily meeting this expectation in the μ-ELMOproject. Firstly, GILES is written in C++ and its interface to other emulators has been designed at the code level, that is, to reuse the GILES framework the developer is required to integrate the source code of the emulator into GILES code base and then recompile the integrated code to generate the executable. As a generic solution, such design could be problematic in certain scenarios, e.g. developers might need to solve problems caused by cross compiling when the emulator is not written in C/C++, and/or part of the emulator must be adapted to be compatible to GILES' code interface. Secondly, GIELS' interface are defined based on the abstracted ELMO architecture. This directly caused the GILES' framework being abandoned during the development of μ-ELMO, since it is based on a realistic architecture that is not compatible with ELMO's.

1.3.2 The $\mathcal{SMURF}$ framework

Reviewing the predecessors, it does not take long to conclude that there is a great engineering difficulty in implementing a leakage simulator platform that can invoke a third party emulator and apply a customisable leakage model to generate leakage traces in a comprehensive manner. Therefore comes the idea of $\mathcal{SMURF}$: rather than plugging different components into a generic leakage simulator platform as attempted in GILES, $\mathcal{SMURF}$ inverts the approach by being a "middleware". That is, it is designated to be a library that is serves as a plugin to different components providing a data exchanging mechanism.

$\mathcal{SMURF}$ focuses on two major tasks during the life cycle of a leakage simulator:

- $\mathcal{SMURF}$ provides an universal interface to the emulator for exporting execution traces, i.e. execution details such as states of registers, flags and

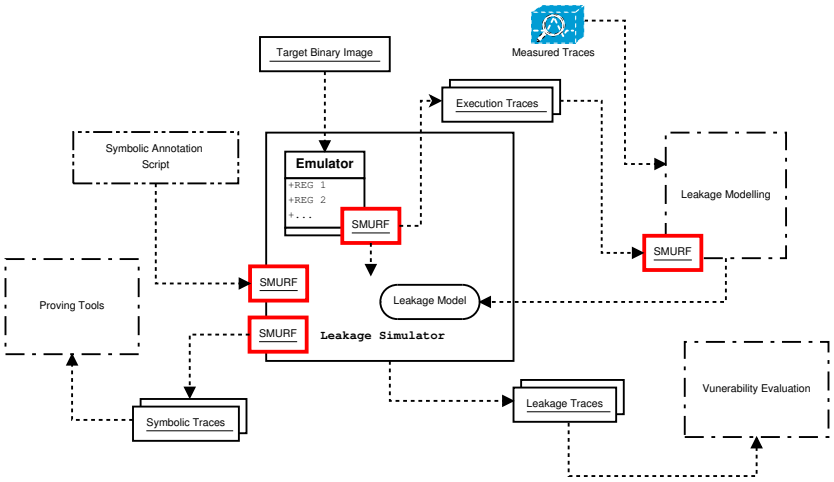

Figure 1.1: An Overview of $\mathcal{SMURF}$

buses during the emulation. The Execution Traces are store in a portable format. The execution traces are combined with the measured traces from real devices to deduce the leakage model, which then to be integrated into the leakage simulator. Finally the simulated leakage traces are generated for vulnerability evaluation where $\mathcal{SMURF}$ can help porting information from the emulator to the leakage simulator.

- $\mathcal{SMURF}$ also provides a set of universal interfaces for the leakage simulator to interact with the proving tools. Firstly, it provides an interface of instructing the leakage simulator to annotate the simulated traces with symbolic information. Secondly, it provides the functionality to export the symbolic traces, i.e. the simulated traces with annotated symbolic information, that can later be imported to proving tools in the back end with $\mathcal{SMURF}$.

In the technical aspect, the $\mathcal{SMURF}$ design prioritises portability and modularity. Figure 1.1 summaries the framework.

Chapter 2

User's Manual

2.1 Introduction

The design of $\mathcal{SMURF}$ follows the concept of Object Oriented and is highly modularised. C/C++ and Python3 are chosen as the supported languages for their generic usage in emulator development and data analysis. The library is written fully compatible with POSIX.1-2001 [3] standard for best portability among various UNIX-style operating systems.

2.2 $\mathcal{SMURF}$ objects

We first introduce the major objects within the $\mathcal{SMURF}$ framework.

2.2.1 Core

The Core object is the abstraction of the target device to be simulated. A Core is constituted of multiple Components (see below) representing either physical components in the hardware or abstracted attributes of the device.

2.2.2 Component

The Component objects are the objects that describe the Core. For comprehensiveness, they can be categorised into two categories:

- Physical Components. These represents physically existed components within the device such as registers, flags or buses.
- Virtual Components. They are used to represent arributes of the Core which does not have a correspondence physically. Some examples in this category are time in the simulation, or human readable descriptions in

Component Type	Reference C99 data type	Note
BOOL	bool	
OCTET	uint8_t	
STRING	char *	Only ASCII strings. No array support. Length: max # of characters (including \0).
INT16	int16_t	
UINT16	uint16_t	
INT32	int32_t	
UINT32	uint32_t	

Table 2.1: Smurf Component Types

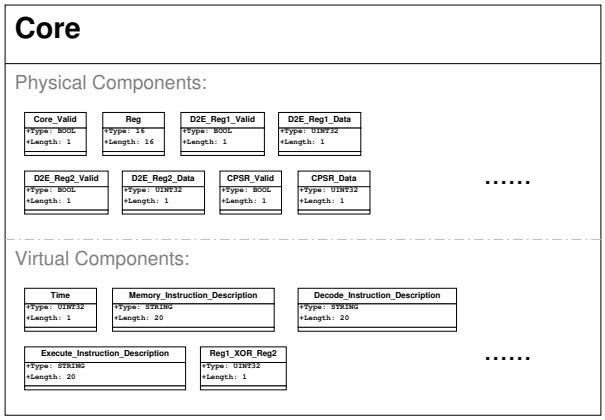

Figure 2.1: Example of Core and Components

fetching, decoding and execution cycles. They may also be used to represent tuples of other Components, such as "(Reg1, Reg2)" which could be used in scenarios like multivariate leakage analysis.

The state of a Component is described by four attributes, namely name, type, length and value. Most of the Component types have their equivalence in C99. Static[1] array declaration is supported except for the STRING type, where the lenght of array[2] is indicated by the Component length. For STRING type Component, only strings of ASCII characters are accepted as the value and the length indicates the maximum number of characters including the NULL terminator '\0'. A reference table is given in Table 2.1.

Figure 2.1 shows an example of a Core containing several Components.

[1] a.k.a fixed length

[2] a.k.a. number of elements

2.2.3 Trace

The Trace object is a sequence of Frames (see below) that records the simulation. In the $\mathcal{SMURF}$ framework it is the major outcome of emulation and simulation and may adopt different forms depending on the usage (e.g. to have execution details in the Execution Trace, symbolic information in the Symbolic Trace, or be applied a leakage model in the Leakage Trace).

2.2.4 Frame

A Frame is a record of the state of Core at a specific moment during the simulation. A Frame can be considered an instantiated Core where values are assigned to its Components at the moment of being record. We refer to this valued Components in a Frame as Component Instances.

An important note is that the $\mathcal{SMURF}$ framework does not specify at which moments during the simulation that a Frame should be recorded and thus it is completely up to the choice of users (i.e. developers of the emulator and the leakage simulator).

2.2.5 Dictionary (optional)

Not finalised yet...

The Dictionary is the collection of all valid Symbols (see below). Note that in contrary to the Core, the Dictionary is dynamic in that it is possible to expand it with new Symbols at the runtime (i.e. during the leakage simulation).

2.2.6 Symbol (optional)

The Symbol objects represent the symbolic representations of intermediate variables during the leakage simulation that are used to generate the Symbolic Traces which are then to be imported by the proving tools. During the simulation, Components in Frames are annotated by Symbols as instructed by a symbolic annotation script specified by the user.

A Symbol needs to be first declared within Dictionary before it can be annotated by. This can be done at the runtime. $\mathcal{SMURF}$ also supports user defined operators over the Symbols, for instance, the user may define an XOR operator:

$$\text{“}s_1 \oplus s_2\text{”} := \text{XOR}(\text{“}s_1\text{”}, \text{“}s_2\text{”})$$

such that it returns a new Symbol (“$s_1 \oplus s_2$”) from two Symbols “s_1” and “s_2”.

Note that Dictionary and Symbols are optional that are only required if the leakage simulator intends to support the symbolic features. Figure 2.2 gives an example of a Trace of multiple Frames with optional symbolic features.

A tree view summary of the relationships of $\mathcal{SMURF}$ objects is provided in Figure 2.3.

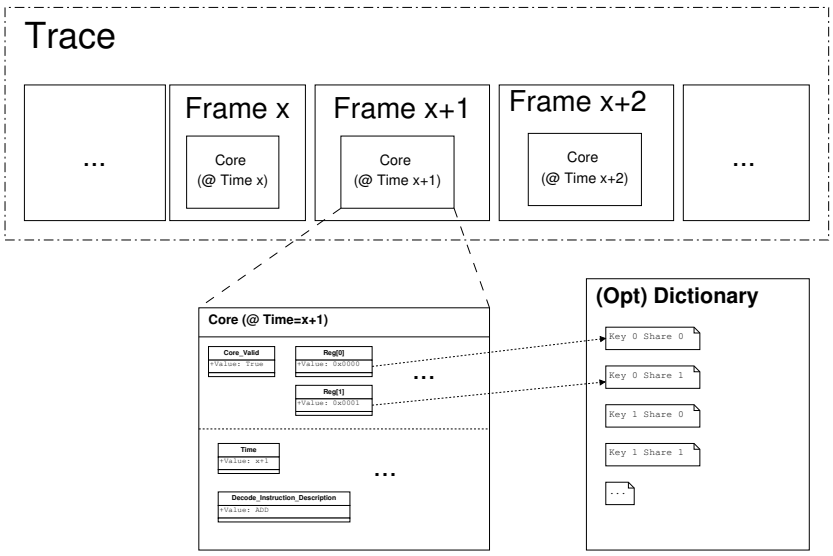

Figure 2.2: Example of Trace and Frames with Symbolic information

```
Core
    [Component](s)
Trace
    [Frame](s)
        Core
            [Component](s) - (Opt)[Symbol](s)
(Opt) Dictionary
    (Opt) [Symbol](s)
```

Figure 2.3: Tree view of $\mathcal{SMURF}$ objects

2.3 $\mathcal{SMURF}$ Modules

Depending on the theme of functionality, from a high level $\mathcal{SMURF}$ can be divided into the following major modules:

- The $\mathcal{SMURF}$ **Basic** module provides basic functionalities for manipulating various $\mathcal{SMURF}$ objects. Its typical usage include initialising and cleaning of $\mathcal{SMURF}$ objects, as well as exporting and importing the $\mathcal{SMURF}$ objects into various user programs. APIs within this module are supported both in C/C++ and Python3.
- The $\mathcal{SMURF}$ **Emulator** module focuses on tasks related to the emulator, particularly about constructing a $\mathcal{SMURF}$ Frame from the internet data structures of the emulator and exporting them as a $\mathcal{SMURF}$ Trace to be read by other parties in the $\mathcal{SMURF}$ framework. There is also a submodule within this module called SmurfIO that provides an universal interface to simulate a virtual serial port on POSIX compatible systems. This module is supported in C/C++ only as it is designed for the usage of the emulator.
- **(Optional)** The $\mathcal{SMURF}$ **Symbolic** module provides the functionality related to handling symbolic operations. Note that it is implemented as an optional extension that it is up to the decision of leakage simulator to support related features or not. This module is supported both in C/C++ and in Python3.

2.4 The Smurf Basic module

For C/C++ programs, the corresponding APIs are defined in `smurf/smurf.h`. For Python3 programs, the corresponding module is `smurf`.

2.4.1 Smurf debugging output

The debugging output of Smurf library are disabled by default. User's can turn it on with `SmurfSetVerbose()`:

```
void SmurfSetVerbose(bool dbginfo);

//Turn on debug info.
SmurfSetVerbose(true);

//Turn on debug info.
SmurfSetVerbose(flase);
```

Listing 2.1: SmurfSetVerbose()

Set `dbginfo` to `true` turns on the debugging output and vice versa.

2.4.2 Core specification

To begin with the $\mathcal{SMURF}$ framework, the first step is to define your Core and generate the Core specification with a Python script. To do so, we first initiate a `smurf.Core` object:

```
import smurf

# Class:
# smurf.Core()

# Example: generate a Core object.
core = smurf.Core()
```

Listing 2.2: smurf.Core()

Use `Smurf.Core.NewComponent()` to add Components into the Core :

```
# Prototype:
# smurf.Core.NewComponent(compname, comptype='OCTET', complen=1):

# Example: Adding 3 Components "RegA", "RegB" and "RegC".
core.NewComponent("RegA")
core.NewComponent("RegB", 'UINT32')
core.NewComponent("RegC", 'OCTET', 4)
```

Listing 2.3: smurf.Core.NewComponent()

`compname` is the name of the Component given as a string. `comptype` is of the types given in Table 2.1 as a string, and `complen` is the size of the Component (in its own type). In the example given in Listing 2.3, three different Components are added, which are namely:

- `RegA` that has an "OCTET" (8 bit) value,
- `RegB` that has an "UINT32" (unsigned 32 bit) value, and
- `RegC` that has 4 "OCTET" value which in total has $8 * 4 = 32$ bits.

Once all the Components are added, the Core specification can be dumped into a file using `Smurf.Core.Save()`:

```
# Prototype:
# smurf.Core.Save(filepath)

# Example: saving the Core into a Core specification
core.Save("./test.core")
```

Listing 2.4: smurf.Core.Save()

`filepath` is the file path of the output Core specification file.

This generates a Core specification file which actually is implemented as a JSON file that can be opened with other viewers such as a browser:

```
{
  "version": "1",
  "components": {
    "RegA": {
      "type": "OCTET",
      "len": 1
    },
    "RegB": {
      "type": "UINT32",
      "len": 1
    },
    "RegC": {
      "type": "OCTET",
      "len": 4
    }
  }
}
```

Listing 2.5: test.core

2.4.3 $\mathcal{SMURF}$ objects at code level

Initialisation and free

To use the Smurf library in C/C++ programs, a Smurf session must be established first. The APIs are defined in the header "smurf/smurf.h". A Core specification (details in Section 2.4.2) is required for the initialisation. It is also required that a path of the Trace to be provided during this procedure alongside its mode, which is either read or write. For a reading session, the Core specification is needed to interpret the Trace and for a writing session it determines how data are structured in the Trace. As of a general practice in C/C++ programs, the Smurf session needs to be freed by the end of its lifespam to recycle its resource.

To initialise a Smurf session, use `InitSmurf()`:

```
Smurf* InitSmurf(
const char *corepath, //Path of Core specificatoin.
const char *tracepath, //Path of Trace file.
int tracemode); //Mode of Trace file.
```

Listing 2.6: InitSmurf()

The `Smurf*` returned by InitSmurf() is the handle of the newly initialised Smurf session. `corepath` and `tracepath` are the paths of the Core specification and the Trace respectively. There are currently 4 modes available for the Trace mode argument `tracemode`:

- To read a Trace, `tracemode` should be set to `SMURF_TRACE_MODE_READ`.
- To write a Trace, `tracemode` should be chosen from one of:

  `SMURF_TRACE_MODE_CREATE` Creates a new Trace file. The file will be truncated if it already exists.

`SMURF_TRACE_MODE_APPEND` Opens an existing Trace file and appending new Frames from its end.

`SMURF_TRACE_MODE_FIFO` Creates a temporary First-In-First-Out (FIFO [4]) at `tracepath`. The FIFO will be removed upon calling `FreeSmurf()`.

Notes about the `tracemode`:

- Trace generated always has been set the permissions of all read and write (i.e. 0666 file mode).
- `SMURF_TRACE_MODE_READ` is used universaly to read Traces of a regular file (as created with `SMURF_TRACE_MODE_CREATE`) or a FIFO (as created with `SMURF_TRACE_MODE_FIFO`).
- As indicated in POSIX statndard, when calling with `SMURF_TRACE_MODE_FIFO`, the process will be blocked until the same FIFO Trace has been opened for read by another process. Behaviours of writing in or reading out the FIFO Trace are also consistent with the FIFO specifications in POSIX.
- It is recommend to use `SMURF_TRACE_MODE_FIFO` when dealing with exceptionally large Traces, as data written into the FIFO Trace will directly passed to the reader's end by the kernel without going through the harddisk. In constrast, when storage of the Trace is desired, `SMURF_TRACE_MODE_CREATE` should be used instead. However, users should be minded that simutaneously reading a regular file Trace as it is being written may create a competition and thus synchronous issues.

By the end of the Smurf session, use `FreeSmurf()` to free the session:

```
void FreeSmurf(Smurf * smurf);
```

Listing 2.7: FreeSmurf()

`FreeSmurf()` has no return value and `smurf` is the handle of the Smurf session to be freed, including allocated memory and other system resources such as file descriptors. If argssmurf is initialised with `SMURF_TRACE_MODE_FIFO`, then the associated FIFO Trace will also be deleted.

Once a Smurf session has been initialised, the static $\mathcal{SMURF}$ objects described in Section 2.2 can be directly accessed from the user program via data structures within the session `smurf`. Note that the structure definitions demonstrated in this section are simplified from the source code such that certain internal data structures are omitted.

For Python3 programs, simply import the `smurf` module:

```
import smurf
```

Listing 2.8: Importing Smurf module

Core

The Core is constructed from the Core specification `corepath` provided to `InitSmurf()`. Core is defined and accessed as:

```
typedef struct {
  int ncomponents;  //Number of Core Components.
  SmurfCoreComponent **components;  //Core Components.
} SmurfCore;

//Access.
smurf->core; //Type: SmurfCore*
```

Listing 2.9: struct SmurfCore

Note that `smurf->core` is a pointer to the Core object. Within the Core, `ncomponents` records the number of Components constitutes the Core and their entries are given as an array of their points.

In Python3, `smurf.Core.Load()` loads a Core specification into a Core object:

```
# Prototype:
# smurf.Core.Load(filepath)

# Example: load "test.core".
core = Smurf.Core.Load("test.core")
```

Listing 2.10: smurf.Core.Load()

`filepath` is the path of the Core specification file to be loaded.

Component

The Components are then defined and accessed as:

```
typedef struct {
  char *name; //Component name.
  SmurfCoreComponentType type; //Component type.
  size_t len; //Size of Component, measured by its type.
} SmurfCoreComponent;

//Access the i-th Component.
smurf->core->components[i]; //Type: SmurfCoreComponent*
```

Listing 2.11: struct SmurfCoreComponent

In particular, the `smurf->core->components[i]->type` is one of the following as in Table 2.1:

```
typedef enum {
  UNDEFINED,  //Error case.
  BOOL, OCTET, STRING,
  INT16, UINT16, INT32, UINT32
} SmurfCoreComponentType;
```

Listing 2.12: enum SmurfCoreComponentType

`smurf->core->components[i]->len` is the length of the Component measured by its type. That is, its value is 1 if the Component has been declared as univariate, or n if the Component has been declared as an array of size n. Only for the type `STRING`, `len` is the maximum number of characters in this Component.

It is important to note that values of Components are not presented within the session `smurf` as the data strctures inside `smurf` records only the static information. Details of accessing the values of Components are given in Section 2.4.4.

In Python3, the Components are implemented as a dictionary member of the Core. It further has three members namely `name`, `type` and `len`:

```
# class:
# smurf.Core.Component

# Example: print Components in a Core "core"
comps = core.components
for c in comps:
  print("Name:{:s}".format(comps[c].name))
  print("Type:{:s}".format(comps[c].type))
  print("Length:{:d}".format(comps[c].len))
```

Listing 2.13: smurf.Core.components

`name` and `type` are the name and type of the Component in strings as described in Section 2.2. `len` is the length of the Component in its type(details in Section 2.2).

Trace

Once a Smurf session is initialised, its Trace is opened as a file descriptor in the user's process. Although the relevant resources will be released when the session is freed, the Trace object provides informatoin for management purposes.

```
typedef struct {
  char *path; //Path of the Trace file.
  int fd;   //File descriptor of Trace file.
  int mode; //Mode of Trace file as being initialised.
  size_t offset;  //Current Frame offset.
} SmurfTrace;

//Accessing the Trace.
smurf->trace; //Type: SmurfTrace*
```

Listing 2.14: struct SmurfTrace

Note that the Trace object should be read only and users should not modify its contents. `mode` records the value of `tracemode` as being initialised in **??**. `offset` records the current position of Frame. When the Trace is initialised by `SMURF_TRACE_MODE_FIFO`, `offset` is effectively the number of Frames read or written.

In Python3, a Trace, `smurf.Trace`, needs first be initialised from a Core and then loaded from the file system using `smurf.Trace.Open()`. Note that currently in Python3 only Trace reading is supported.

```
# Class:
# smurf.Trace

# Example: initialise from "core" and load a Trace at "filepath"
trace = smurf.Trace(core)
trace.Open(filepath)
```

Listing 2.15: smurf.Trace

Frame

The Smurf library implements two types of Frame.

Buffer Frame A buffer Frame is allocated alongside session initialisation for quick access:

```
typedef struct {
  int len;  //Size of the Frame in bytes.
  unsigned char *buf; //Contents of Frame.
} SmurfFrame;

//Accessing the Frame buffer.
smurf->frame; //Type: SmurfFrame*
```

Listing 2.16: struct SmurfFrame

Further details of Frame APIs are given in Section 2.4.4.

Self managed Frame Alternatively, users may use self managed Frames for more flexibility. To create a self manged Frame within the Smurf session `smurf`, use `SmurfNewFrame()`:

```
SmurfFrame *SmurfNewFrame(Smurf * smurf);

//The new self manged Frame newframe:
newframe = SmurfNewFrame(smurf);
```

Listing 2.17: SmurfNewFrame()

NULL is returned if `SmurfNewFrame()` failed.

It lies in the user's responsibility to free the self managed Frames at the end of its lifecycle by calling `FreeFrame()`:

```
void *SmurfFreeFrame(SmurfFrame * frame);

//Free the self manged Frame frame:
SmurfFreeFrame(frame);
```

Listing 2.18: SmurfFreeFrame()

`CopyFrame()` copies a Frame from one to another:

```
int *CopyFrame(SmurfFrame * dstframe, SmurfFrame * srcframe);

//Example: copy the buffer Frame into a self manged Frame
SmurfFrame *newframe;
newframe = SmurfNewFrame(smurf);
CopyFrame(newframe, smurf->frame); //newframe is a copy of the
    buffer Frame.
```

Listing 2.19: CopyFrame()

On success, `SMURF_SUCCESS` is returned. Otherwise `SMURF_ERROR` is returned.

In Python3 Frames are defined as the class `smurf.Frame`:

```
# Class:
# smurf.Frame
```

Listing 2.20: smurf.Frame

It is returned by calling `smurf.Trace.NextFrame()`. Further details are provided in Section 2.4.6.

2.4.4 Component Instances within a Frame

The Smurf library provides the `FrameComp` structure as a handle to access the Component Instances:

```
typedef struct{
  const char *name; //Name of the Component.
  SmurfCoreComponentType type;  //Type of the Component.
  size_t typesize;  //Unit size of the Component.
  const char *typestr;  //Type of Component in string.
  size_t size;  //Size of the Component in its type.
  CompVal val;  //Pointer to its value.
} FrameComp;
```

Listing 2.21: struct FrameComp

The members in a `FrameComp` object should be read only since they are derived from the Core specification, except for `val`.

`name` is the name of the Component. `type` is as defined in Listing 2.12 and `typestr` are their correponding strings, e.g. `"BOOL"`, `"OCTET"` and `"UINT16"`, etc.

`typesize` is the unit size of this Component in bytes. This is equivalent to `sizeof(comp_c99type)` in C/C++ where `comp_c99type` is the referenced C99 data type of the Component in Table 2.1, for example, it is 1 for `BOOL`, `OCTET`, 2 for `INT16` and `UINT16` and 4 for `INT32` and `UINT32`. `size` gives the size of the Component in its type. That is, for univariate Components it is 1, and the number of elements for Components defined as arrays. For `STRING` typed Components `typesize` and `size` are constantly 1.

The value(s) of a Component Instance is given by the member `val`, which type is defined as a union of pointers:

```
typedef union {
  uint8_t *BOOL;
  uint8_t *OCTET;
  char *STRING;
  int16_t *INT16;
  uint16_t *UINT16;
  int32_t *INT32;
  uint32_t *UINT32;
} CompVal;
```

Listing 2.22: union CompVal

Then `FetchComp()` can be used to fetch a Component Instance from a Frame by its name:

```
int FetchComp(FrameComp * comp, SmurfFrame * frame, const char *
    compname);

//Example: fetch "RegA" from the buffered Frame
FrameComp comp = {0}; //The Component Instance.
FetchComp(&comp, smurf->frame, "RegA");
```

Listing 2.23: FetchComp()

Users can then read or alter the Component Instances using the corresponding members of `comp.val`. The total size of `comp.val` in bytes can be computed by `(comp.typesize * comp.size)`.

```
//Examples of accessing Component Instances.
//e.g. for univariate Component Instances:
*comp.val.OCTET;  //Type: uint8_t, or
comp.val.OCTET[0]; //Type: uint8_t

//For arrayed Compoent Instances, the i-th value is:
comp.val.UINT16[i]; //Type: uint_16
```

Listing 2.24: Accessing value(s) of a Components Instance in C/C++

For Python3, the Component Instances are derived from the class `smurf.Core.Component` and are implemented as a member of Frame `smurf.Frame.ComponentInstance`:

```
# Class:
# smurf.Frame.ComponentInstance(smurf.Core.Component)

# Example: print the value of "RegA" in a Frame "frame"
rega = frame.components["RegA"]
print("Value: {}".format(rega.val)) # In its own type.
print("Value(raw): {}".format(rega.raw)) # In bytes.
```

Listing 2.25: smurf.Trace

`smurf.Frame.ComponentInstance` has two additional member comparing to its parent, namely:

- `val` is the value as in the type indicated by `type`.
- `raw` is the value in bytes.

2.4.5 Generate Traces

Trace generation are normally done by the Emulator in the $\mathcal{SMURF}$ framework and thus the relevant functionalities are provided only in C/C++ since they are the general languages for the Emulator. The lower level APIs for Trace generation are provided in the Basic module whilst the more sophisticated ones are introduced in Section 2.5.

To generate a Trace, the Smurf session must be initialised with `tracemode` specified as either `SMURF_TRACE_MODE_CREATE` or `SMURF_TRACE_MODE_APPEND`. A user may choose to write the buffer Frame into the Trace or a self managed one.

`SmurfWrite()` writes the buffer Frame into the Trace within the same Smurf session:

```
int SmurfWrite(Smurf * smurf);

//Example:
SmurfWrite(smurf);
smurf->frame; //The next Frame.
```

Listing 2.26: SmurfWrite()

On success, `SMURF_SUCCESS` is returned. Otherwise `SMURF_ERROR` is returned.

Alternatively, users can use `SmurfWriteFrame()` to write a self managed Frame into a Trace of a Smurf session:

```
int SmurfWriteFrame(Smurf * smurf, SmurfFrame * frame);

//Example: fetch the i-th Frame and add it back to Trace.
SmurfFrame *newframe;
newframe = FetchFrame(newframe, smurf, i);
//You may also modify newframe between.
SmurfWriteFrame(smurf, newframe); //Add newframe back into the
    Trace.
```

Listing 2.27: SmurfWriteFrame()

On success, `SMURF_SUCCESS` is returned. Otherwise `SMURF_ERROR` is returned.

The Python3 `smurf` module does not support Trace generation currently.

2.4.6 Read Traces

The Smurf library provides two styles of Trace reading:

- Sequential reading. The Frames are read out one by one from the beginning towards the end.
- Indexed reading. The user can pick a Frame to read by its index in the Trace.

Sequential reading

Sequential reading is implemented using the buffer Frame in the Smurf session by `SmuefNextFrame()`:

```
int SmurfNextFrame(Smurf * smurf);

//Example: get next Frame
SmurfNextFrame(smurf);
smurf->frame; //The next Frame.
```

Listing 2.28: SmurfNextFrame()

Each call to `SmurfNextFrame()` updates the buffer Frame `smurf.frame` to the new Frame read. On success the index of the newly read Frame is returned (counting from 1). A return value of 0 indicates the End of File (EOF) is reached and the buffer Frame is unchanged. `SMURF_ERROR` is returned in case of error.

Sequential reading is also supported in Python3 by `smurf.Trace.NextFrame()`:

```
# Prototype:
# smurf.Trace.NextFrame()

# Example: extract frames from the Trace "trace"
frames = list()
while True:
  frame = trace.NextFrame() # Returns the next Frame.
  if None == frame: # EOF
    break
  frames.append(frame)
```

Listing 2.29: smurf.Trace.NextFrame()

Indexed reading

Indexed reading is implemented by `SmurfFetchFrame()` which fetches the Frame within the Trace by its index:

```
int SmurfFetchFrame(SmurfFrame * frame, Smurf * smurf, unsigned
    long frameno);

//Example: fetching the i-th Frame into a self managed Frame.
SmurfFrame *newframe;
newframe = SmurfNewFrame(smurf);
SmurfFetchFrame(newframe, smurf, i); //The i-th Frame is fetched
    into newframe.
```

Listing 2.30: SmurfFetchFrame()

On success `SmurfFetchFrame()` returns the total size of the Frame values in bytes. A return value of 0 indicates the Frame could not be fetched, commonly caused by the specified *frameno* exceeding the total number of Frames in the Trace. In cases of error, `SMURF_ERROR` is returned.

`SmurfFetchFrame()` only works on regular Trace files (see `S_ISREG` section in [5]). This also means it is not compatible for Trace files created with mode `SMURF_TRACE_MODE_FIFO`. Also fetching a Frame does not change the Trace offset `smurf->trace->offset`. A noticable feature of `SmurfFetchFrame()` is that it

		Python3	C/C++
Generate Core specification		✓	×
Read Trace	Sequential reading	✓	✓
	Indexed reading	×	✓
Write Trace	Buffered writing	×	✓
	Non-buffer writing	×	✓
Interpret Frame		✓	✓

Table 2.2: Summary of Basic module

can also be used on Traces opened for write (`SMURF_TRACE_MODE_CREATE`) or append (`SMURF_TRACE_MODE_APPEND`).

2.4.7 Summary

Table 2.2 summarises different functionalities in the Basic module both in Python3 and C/C++.

2.5 The Smurf Emulator module

The corresponding APIs are defined in `smurf/emulator.h`. Since Emulators are normally written in C/C++ languages, there is no corresponding Python3 module for this Smurf module.

2.5.1 Synchronising Component Instances

Although the Smurf Basic module (Section 2.4) provides a method to construct Frames that can be written into the Trace, it also requires the user to manually update the Frame whenever it is to be written. Such procedures are normally highly repetitive; thus the Smurf Emulator module provides a synchronisation mechanism that synchronises Smurf Component Instances with internal variables of the Emulator into the within the buffer Frame. Together with the buffer Frame related APIs (Listing 2.26 particularly), this greatly reduces the effort for constructing new Frames.

SmurfBind() binds the internal variables of Emulator to the Component Instances in the buffer Frame of the Smurf session `smurf`:

```
int SmurfBind(Smurf * smurf, const char *comp, const void *var);

//Example: binding variables
uint32_t cycle = 0;
uint32_t regs[16] = {0};

//Bind univariate Component "Cycle"
SmurfBind(smurf, "Cycle", &cycle);
//Bind array Component "Regs"
```

```
SmurfBind(smurf, "Regs", regs);
```

Listing 2.31: SmurfBind()

`comp` is the name string of the Component that is to be bound and `val` is a pointer to the source internal variable of the Emulator, or the address of the first element if it is an array. `SmurfBind` relies on user's responsibility to ensure that the types of `comp` and `var` are matched as in Table 2.1. On success `SmurfBind()` returns `SMURF_SUCCESS`, or `SMURF_ERROR` otherwise.

Once a variable is bound, `SmurfSync()` synchronises the bound Component Instances within the buffer Frame of Smurf session `smurf` from the source variables of the Emulator:

```
void SmurfSync(Smurf * smurf);

//Example: Synchronise and write the buffer Frame into Trace.
//Synchronise bound Component Instances.
SmurfSync(smurf);
//Write the buffer Frame into the Trace.
SmurfWrite(smurf);
```

Listing 2.32: SmurfSync()

The example in Listing 2.32 shows a typical usage of `SmurfSync()` that is called before writing the updated buffer Frame into the Trace.

2.5.2 Smurf IO

The submodule Smurf IO provides a mechanism for Emulators to create a synchronised virtual serial port interface that can interact with the hosting system as if being a real device.

Note that Smurf IO can be independently used without any Smurf session.

Host: configure virtual interface in Emulator

Like a general device in any UNIX systems, the virtual interface appears as a file in the system. The first step that needs to be done by the Emulator is to create such a Smurf interface (Smurf IF) using `SioOpen()`:

```
SmurfIO* SioOpen(const char* ifpath);

//Example: create a Smurf IF at "/tmp/smurfif"
SmurfIO sif = NULL;
sif = SioOpen("/tmp/smurfif");
```

Listing 2.33: SioOpen()

`ifpath` is the file path in the system. On success `SioOpen()` returns the handle of the newly created Smurf IF, or a `NULL` pointer is returned if failed. A file specified by `ifpath` is created on the successful return of `SioOpen()`.

The Smurf IF must be connected to another end before any data can be transmitted over it. `SioWaitConn()` waits for a connection to be established:

```
int SioWaitConn(SmurfIO * sif);

//Example: wait for "sif" to be connected
SioWaitConn(sif);
```

Listing 2.34: SioWaitConn()

`sif` is the handle of the Smurf IF waiting to be connected. On success `SioWaitConn()` returns `SMURF_SUCCESS`indicated `sif` is now connected, or `SMURF_ERROR`if it failed.

Note that `SioWaitConn()` is blockage, i.e. it blocks the procedure until a connection is received or an error is occurred.

Within the `SmurfIO` structure, users can check the Smurf IF using its members with macros in `smurf/emulator.h`:

```
typedef struct
{
  int type; //Type of the Smurf IF
  int stat; //Status of the Smurf IF
} SmurfIO;
```

Listing 2.35: struct SmurfIO

`type` indicates the type of the Smurf IF, it is either:

- `SMURF_IO_SERVER` indicating it is the server side that usually is the Emulator that waits for a connection, or
- `SMURF_IO_CLIENT` indicating this is the client side that is usually the user program that handles the IO of simulated device.

`stat` is the status of the Smurf IF, its possible values are:

- `SMURF_IO_WAITCONN` indicates that it is a newly initialised Smurf IF where the connection is to be established.
- `SMURF_IO_READY` indicates that the Smurf IF is connected and ready to be used.
- `SMURF_IO_EOF` indicates that the other end of connection has been closed.
- `SMURF_IO_ERROR` is the erroneous state of the Smurf IO.

`SioGetchar()` and `SioGet()` are provided for basic IO with the Smurf IF:

```
int SioGetchar(SmurfIO *);

//Example: read a byte from Smurf IF "sif".
int c;
c = SioGetchar(sif);
```

Listing 2.36: SioGetchar()

```
int SioPutchar(SmurfIO *, const int charval);

//Example: write a byte "c" into Smurf IF "sif".
char c = 0;
SioGetchar(sif, c);
```

Listing 2.37: SioPutchar()

`SioGetchar()` and `SioPutchar()` are synonyms to `getchar()` and `putchar()` in standard C library with `sif` being the Smurf IF to be used. Note that similar to most read and write APIs in UNIX systems, `SioGetchar()` blocks until a byte is received or an error has occurred, whilst `SioPutchar()` returns immediately after the sending request is issued to the Smurf IF.

Note that Smurf IO covers only data transmission between the outside interactive program and the Emulator. It is then the responsibility of the Emulator to provide a communication mechanism between the Emulation and the binary image to be emulated. A general practice of implementing such communication channel is through reading or writing a reserved memory address, resembling to the UART interfaces of many commercial devices.

By the end of the life cycle of the Smurf IF, use `SioClose()` to release it:

```
void SioClose(SmurfIO *sif);

//Example: close a Smurf IF "sif"
SioClose(sif);
```

Listing 2.38: SioClose()

Client: connect to Smurf IF from interactive program

For C/C++ programs, use `SioConnect()` to connect to a Smurf IF:

```
SmurfIO *SioConnect(const char *sifpath);

//Example: connect to a Smurf IF at "/tmp/sioif"
SmurfIO* sio;
sio = SioConnect("tmp/sioif");
```

Listing 2.39: SioConnect()

`sifpath` is the file path of the Smurf IF as created by the host (i.e. normally the Emulator). On success return, the handle of the newly established Smurf IF (a pointer to a `SmurfIO` object) is returned. Otherwise a `NULL` pointer is returned indicating the establishment of a connection is failed.

Note that `SioConnect()` is also blockage , i.e. it blocks the process until the connection is established or an error occurs.

Once a connection is established, the user process can use `SioGetchar()`(Listing 2.36) and `SioPutchar`(Listing 2.37) to receive and send data through the channel. The established Smurf IF needs to be later freed by `SioClose()`(Listing 2.38).

Smurf IF can also be used in Python3 clients. The first step is to initialise a `smurf.IO` object with the file path of the Smurf IF:

```
# Class:
# smurf.IO(sifpath)

# Example: connecting to Smurf IF at "/tmp/sioif"
sif = smurf.IO("/tmp/sioif")
```

Listing 2.40: smurf.IO

Initiating a `smurf.IO` object, which is also blockage, automatically connects to the Smurf IF specified by `sifpath`. It then can send and receive data using `smurf.IO.Getchar()` and `smurf.IO.Putchar()`:

```
  # Prototype:
  # smurf.IO.Gettchar()

  # Example: send a byte 0x00 through Smurf IF "sif"
  recv_b = sif.Getchar()
```

Listing 2.41: smurf.IO.Getchar()

```
# Prototype:
# smurf.IO.Putchar(a_byte)

# Example: send a byte 0x00 through Smurf IF "sif"
sif.Putchar(bytes([0]))
```

Listing 2.42: smurf.IO.Putchar()

The input `a_byte` and the returned variable `recv_b` are both of type `bytes`. Like their C correspondences `SioGetchar()`(Listing 2.37) and `SioPutchar()`(Listing 2.36,) `smurf.IO.Getchar()` and `smurf.IO.Putchar()` are blockage and non blockage respectively.

2.6 The $\mathcal{SMURF}$ Symbolic engine

Chapter 3

Developer's Guide

Bibliography

[1] URL: https://github.com/sca-research/GILES.

[2] URL: https://github.com/dwelch67/thumbulator.

[3] URL: https://pubs.opengroup.org/onlinepubs/9699919799.2008edition/.

[4] URL: https://man7.org/linux/man-pages/man7/fifo.7.html.

[5] URL: https://www.gnu.org/software/libc/manual/html_node/Testing-File-Type.html.

[6] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2. URL: https://doi.org/10.1007/978-3-540-28632-5%5C_2.

[7] Si Gao and Elisabeth Oswald. "A Novel Completeness Test for Leakage Models and Its Application to Side Channel Attacks and Responsibly Engineered Simulators". In: *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part III*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13277. Lecture Notes in Computer Science. Springer, 2022, pp. 254–283. DOI: 10.1007/978-3-031-07082-2_10. URL: https://doi.org/10.1007/978-3-031-07082-2%5C_10.

[8] Si Gao, Elisabeth Oswald, and Dan Page. "Towards Micro-architectural Leakage Simulators: Reverse Engineering Micro-architectural Leakage Features Is Practical". In: *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part III*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13277. Lecture Notes in Computer Science. Springer, 2022, pp. 284–311. DOI: 10.1007/978-3-031-07082-2_11. URL: https://doi.org/10.1007/978-3-031-07082-2%5C_11.

[9] David McCann, Elisabeth Oswald, and Carolyn Whitnall. "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, 2017, pp. 199–216. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann.