Laboratoire d'Annecy de Physique des Particules

Introduction : Phoenix Shredder

Pierre Aubert

Introduction

How to test a program well ?

Introduction

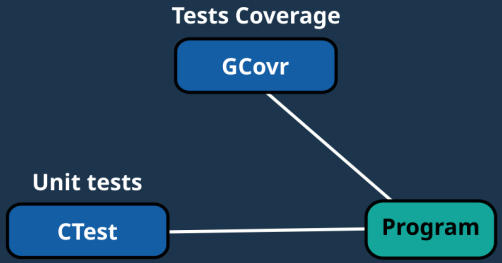
How to test a program well ?

Introduction

How to test a program well ?

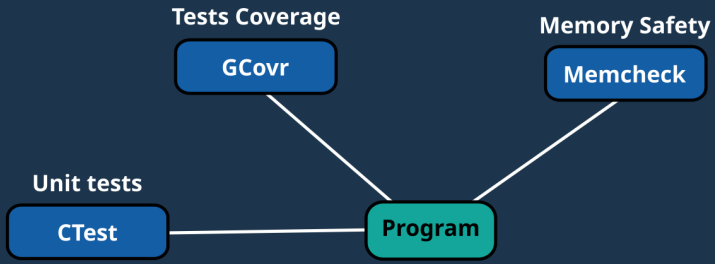
Introduction

How to test a program well ?



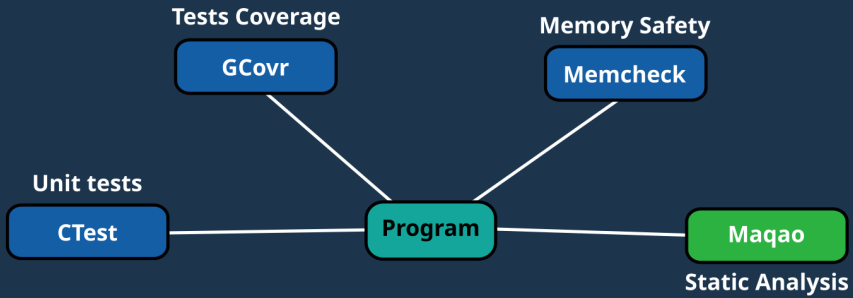
Introduction

How to test a program well ?



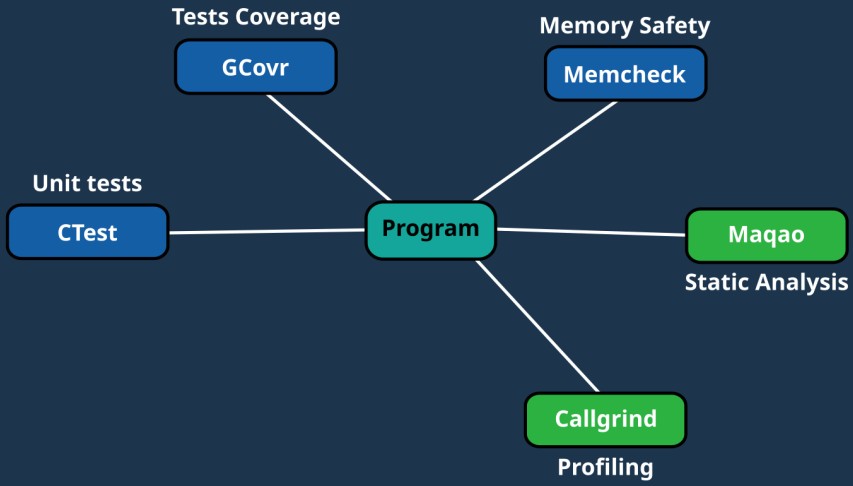
Introduction

How to test a program well ?



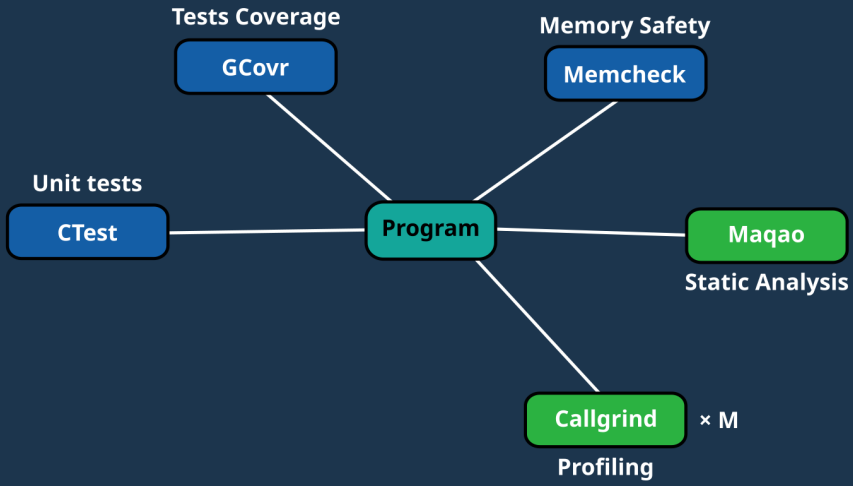
Introduction

How to test a program well ?



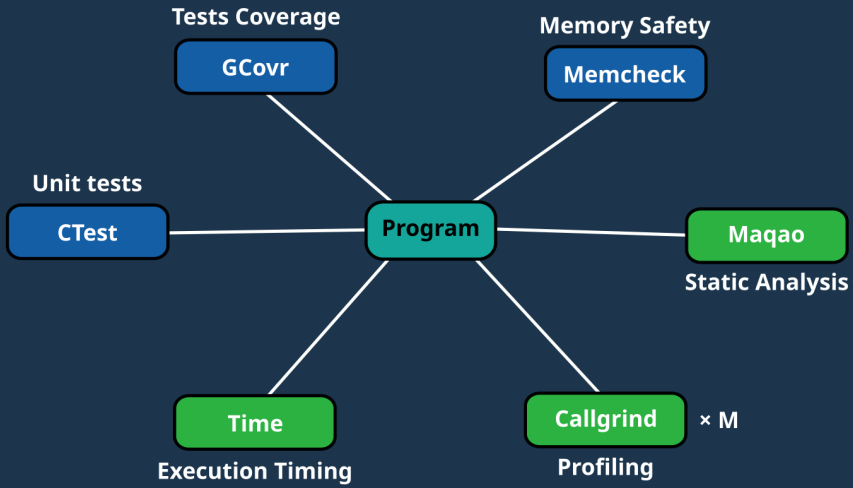
Introduction

How to test a program well ?



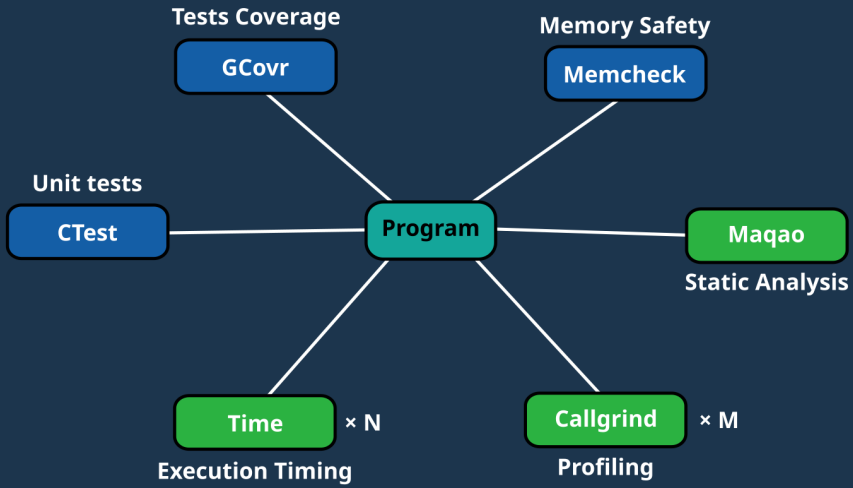
Introduction

How to test a program well ?



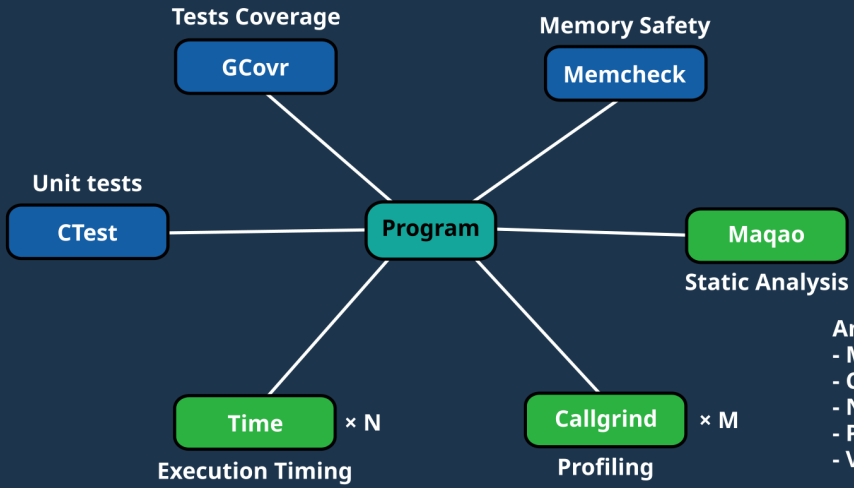
Introduction

How to test a program well ?



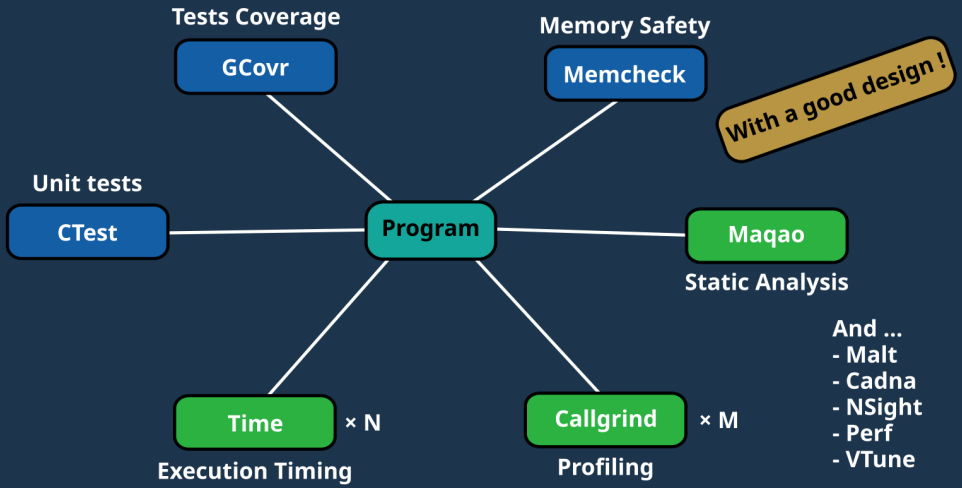
Introduction

How to test a program well ?



Introduction

How to test a program well ?



Unit Tests

CMake CTest / pytest : unit tests

GCovr : coverage report

GCC Code Coverage Report

Directory: ./
Date: 2025-10-16 17:30:07
Coverage: low: ≥ 0% medium: ≥ 75.0% high: ≥ 90.0%

	Exec	Total	Coverage
Lines:	734	781	94.0%
Functions:	126	136	92.6%
Branches:	808	902	89.6%

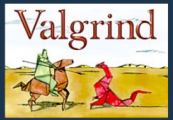
List of functions

File	Lines		Functions		Branches	
src/PAxisIdx.cpp	88.1%	96 / 109	85.2%	23 / 27	90.9%	80 / 88
src/PAxisIdx.h	100.0%	2 / 2	100.0%	1 / 1	-%	0 / 0
src/PMemoryManagerCpu.cpp	96.6%	28 / 29	100.0%	7 / 7	89.7%	26 / 29
src/PMemoryManagerCpu_impl.h	100.0%	13 / 13	100.0%	4 / 4	-%	0 / 0
src/PTensor.h	100.0%	1 / 1	-%	0 / 0	100.0%	13 / 13
src/PTensor_impl.h	95.9%	116 / 121	96.0%	24 / 25	78.2%	86 / 110
src/PTensorIdx.cpp	83.0%	137 / 165	84.4%	27 / 32	81.1%	120 / 148
src/PTensorIdx.h	100.0%	45 / 45	100.0%	11 / 11	70.8%	17 / 24
TESTS/TEST_PAXISIDX/main.cpp	100.0%	83 / 83	100.0%	10 / 10	75.3%	73 / 97
TESTS/TEST_PTENSOR/main.cpp	100.0%	78 / 78	100.0%	6 / 6	100.0%	137 / 137
TESTS/TEST_PTENSORIDX/main.cpp	100.0%	135 / 135	100.0%	13 / 13	100.0%	256 / 256

Generated by: GCOVR (Version 8.3)

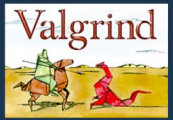
Valgrind Memcheck

Valgrind Memcheck



Memcheck : CPU emulation -> **Memory**
- **uninitialised** variables
- **invalid read/write** of an address
- **memory leaks**

Valgrind Memcheck



Memcheck : CPU emulation -> **Memory**
- **uninitialised** variables
- **invalid read/write** of an address
- **memory leaks**

```
==20774== Memcheck, a memory error detector
==20774== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==20774== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==20774== Command: /home/pierre/projects/PHOENIX2/PhoenixShredder/TESTS/TestSimpleProject//build_quality/./hadamard_product 512
==20774==
==20774==
==20774== HEAP SUMMARY:
==20774==     in use at exit: 0 bytes in 0 blocks
==20774==   total heap usage: 26 allocs, 26 frees, 92,756 bytes allocated
==20774==
==20774== All heap blocks were freed -- no leaks are possible
==20774==
==20774== For lists of detected and suppressed errors, rerun with: -s
==20774== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

/usr/bin/time

/usr/bin/time OPTIONS **my_program** ARGUMENTS

Simple profiler to get :
- **user** time
- **system** time
- **real** time
- **page fault**
- **cache miss**
- etc

/usr/bin/time

/usr/bin/time OPTIONS **my_program** ARGUMENTS

Simple profiler to get :
- **user** time
- **system** time
- **real** time
- **page fault**
- **cache miss**
- etc

/usr/bin/time -f "Real %e User %U System %s" -o perfOutput.txt **./hadamard_product 100000000**

perfOutput.txt -> Real 1.47 **User** 0.76 **System** 0

Maqao

cqa : Code Quality Analyzer
(check if the compiler did a good job)

Static binary analysis

Maqao

cqa : Code Quality Analyzer
(check if the compiler did a good job)

Static binary analysis

Usage : maqao cqa --fct-loops=functionName path/to/library.so

Maqao

MAQAO

cqa : Code Quality Analyzer
(check if the compiler did a good job)

Static binary analysis

Usage : maqao cqa --fct-loops=functionName path/to/library.so

No need to run the program

Analysis of :
- computation **vectorization**
- **bottlenecks**

Advices :
- new **compilation options**
- use **other compilers**

Maqao

cqa : Code Quality Analyzer
(check if the compiler did a good job)

Static binary analysis

Usage : maqao cqa --fct-loops=functionName path/to/library.so

No need to run the program

Analysis of :
- computation **vectorization**
- **bottlenecks**

Advices :
- new **compilation options**
- use **other compilers**

Packaged with **Pixi**

Maqao

cqa : Code Quality Analyzer
(check if the compiler did a good job)

Static binary analysis

Usage : maqao cqa --fct-loops=functionName path/to/library.so

No need to run the program

Analysis of :
- computation **vectorization**
- **bottlenecks**

Advices :
- new **compilation options**
- use **other compilers**

Available on **conda forge**

Packaged with **Pixi**

Maqao

cqa : Code Quality Analyzer
(check if the compiler did a good job)

Static binary analysis

Usage : maqao cqa --fct-loops=functionName path/to/library.so

MAQAO

Target processor is: 12th generation Intel Core processors and Intel Xeon processor product family based on Alder Lake microarchitecture (x86_64 architecture).

▼ **Source loop ending at line 72 in ...ut_autovec_laplacian_light.cpp**

It is composed of the loop 2

▼ **MAQAO binary loop id: 2**

The loop is defined in /home/pierre/projects/COURS/PerformanceWithStencil/Examples/6-Vectorization/64-AutoVecLaplacianLight/gray_scott_layout_autovec_laplacian_light.cpp:60-62,69-72.
The related source loop is not unrolled or unrolled with no peel/tail loop.
40% of peak computational performance is used (19.37 out of 48.00 FLOP per cycle (GFLOPS @ 1GHz))

gain | potential | hint | expert

Vectorization

Your loop is fully vectorized, using full register length.

Details

All SSE/AVX instructions are used in vector version (process two or more data elements in vector registers).

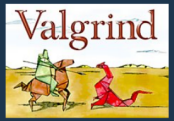
Execution units bottlenecks

Performance is limited by execution of FP multiply or FMA (fused multiply-add) operations (the FP multiply/FMA unit is a bottleneck). By removing all these bottlenecks, you can lower the cost of an iteration from 19.00 to 12.50 cycles (1.52x speedup).

Workaround

Reduce the number of FP multiply/FMA instructions

Valgrind Callgrind



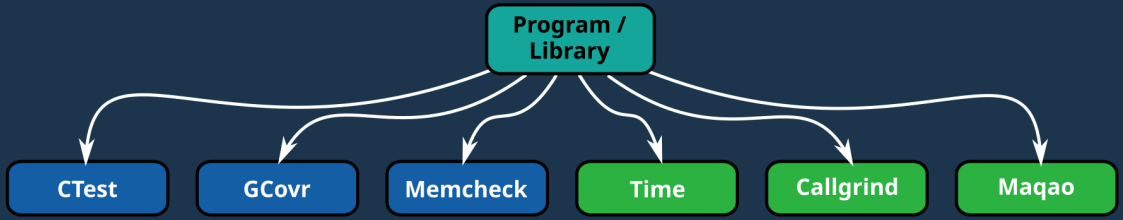
Callgrind : **CPU emulation** and **function profiling**
KCacheGrind : profile results visualisation

Shredder

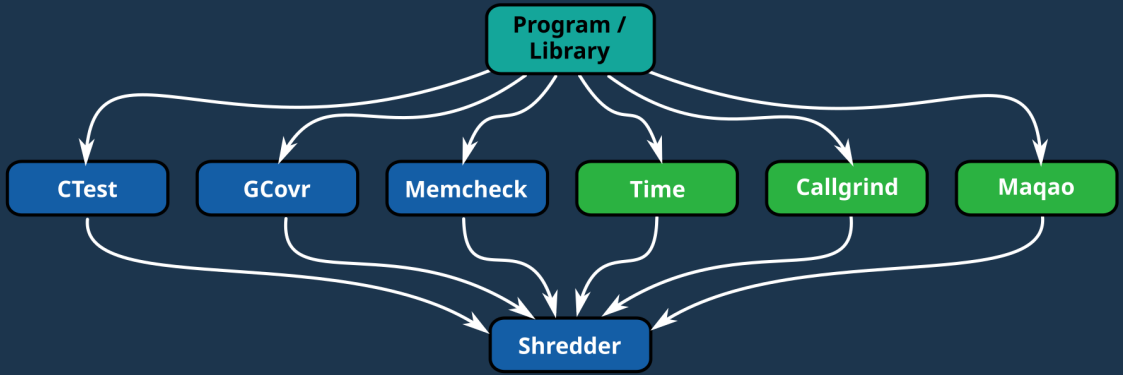
Shredder

Program / Library

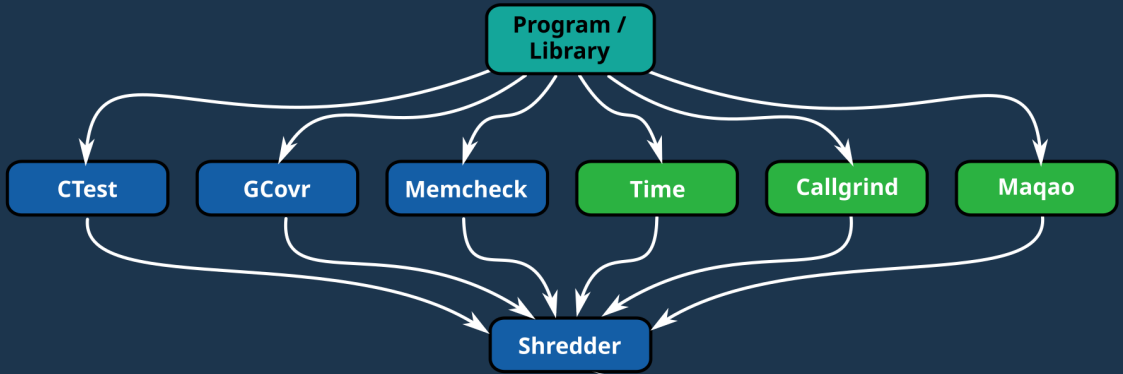
Shredder


Program / Library
CTest
GCovr
Memcheck
Time
Callgrind
Maqao

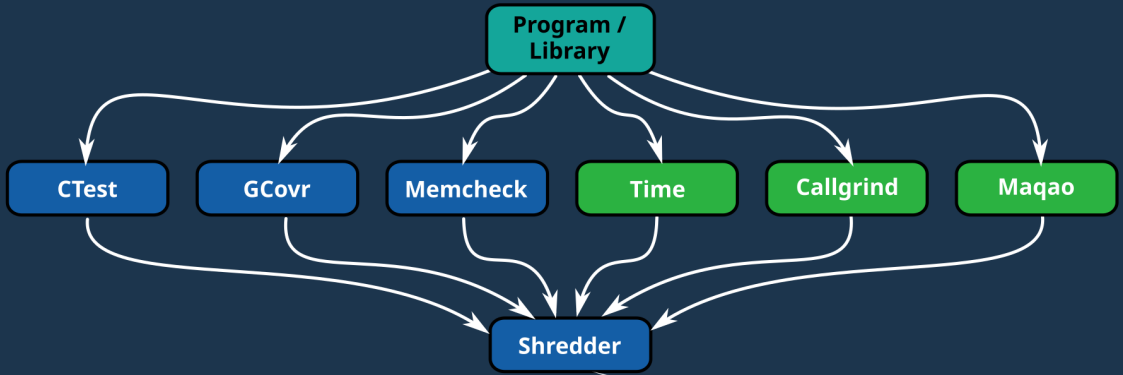
Shredder


Program / Library
CTest
GCovr
Memcheck
Time
Callgrind
Maqao
Shredder

Shredder


Program / Library
CTest
GCovr
Memcheck
Time
Callgrind
Maqao
Shredder
Gather **all tools report information** in a **single web site**

Shredder


Program / Library
CTest
GCovr
Memcheck
Time
Callgrind
Maqao
Shredder
Packaged with **Pixi**
Gather **all tools report information** in a **single web site**

Shredder as a service

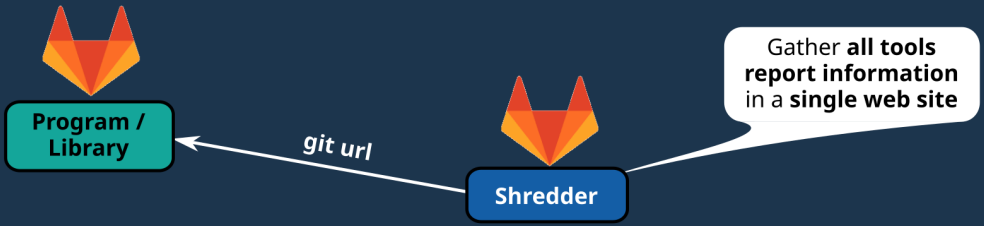
Shredder as a service

Program / Library

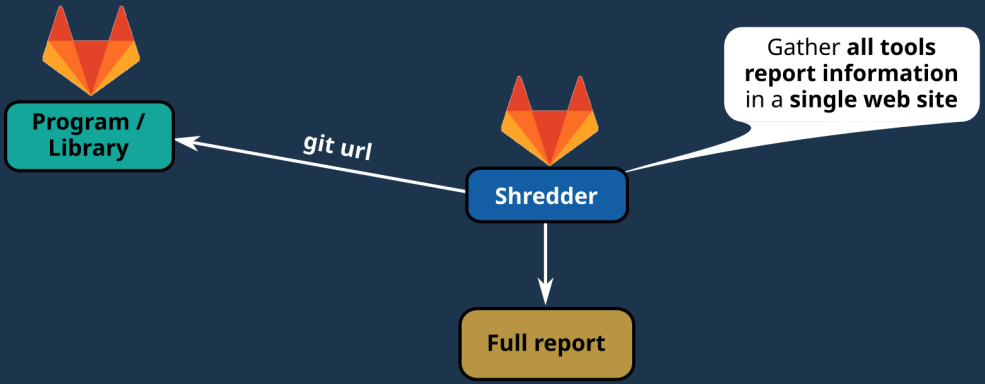
Shredder as a service

Shredder as a service

Shredder as a service



Shredder as a service



Conclusion

Shredder gathers all report information from :
- **ctest**
- **gcovr**
- **valgrind** : **memcheck** / **callgrind**
- **maqao**
- **time**

- Missing **Valgrind Cadna** to check precision
- Missing **Malt**