Runtime Function Instrumentation with EZTrace

Charles Aulagnon, **Damien Martin-Guillerez**, François Rué and François Trahay

5th Workshop on Productivity and Performance (PROPER 2012) 28 AUG. 2012

INTRODUCTION

Modern HPC applications are **complex**:

- **complex hardware:** NUMA architecture, hierarchical caches, accelerators;
- **hybrid programming models:** MPI/OpenMP/PThread.

Understand the performance of those applications is difficult.

➔ **Generating program behavior trace** with a **low overhead**

➔ **Code instrumentation**

INTRODUCTION
Code instrumentation – How?

Manual instrumentation:

- insert instrumentation code **manually** in the source code;
- **painful**: code modification, recompilation...

```
int f(int a, double b) {
 do_something();
}
```



```
int f(int a, double b) {
 record_event("f_entry");
 do_something();
 record_event("f_exit");
}
```

Automatic instrumentation:

- **Automatically** insert probes in the application;
- No code modification, no recompilation...

OUTLINE

1. EZTrace
2. Standard approaches
3. Function instrumentation in EZTrace
4. Evaluation
5. Conclusion

1
EZTrace

EZTrace

EZTrace is a trace generator:

- Look out for some events during execution and record them in a file

```
$ export EZTRACE_TRACE=”mpi pthread”
$ mpirun -np 16 eztrace mpiapp   or   $ eztrace pthreadapp
```

EZTrace

EZTrace is a trace generator:

- Look out for some events during execution and record them in a file

```
$ export EZTRACE_TRACE="mpi pthread"
$ mpirun -np 16 eztrace mpiapp   or   $ eztrace pthreadapp
```

Generic plugins
(MPI, PThread, OpenMP)
or user-defined

EZTrace

EZTrace is a trace generator:

- Look out for some events during execution and record them in a file

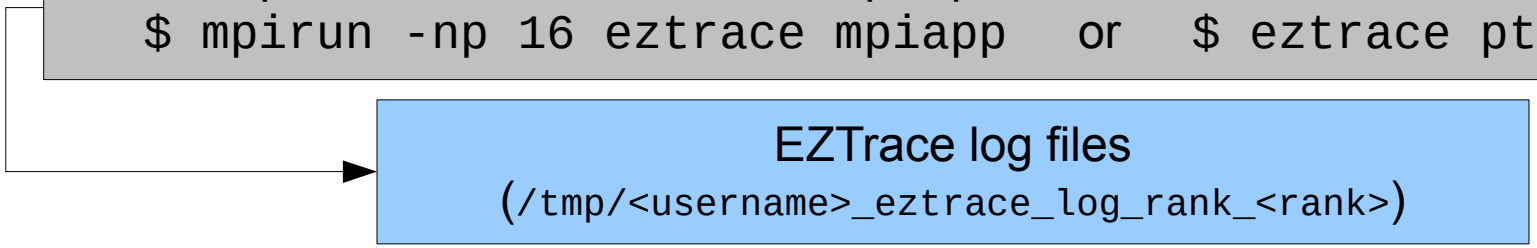
```
$ export EZTRACE_TRACE=”mpi pthread”
$ mpirun -np 16 eztrace mpiapp   or   $ eztrace pthreadapp
```

EZTrace

EZTrace is a trace generator:

- Look out for some events during execution and record them in a file

```
$ export EZTRACE_TRACE="mpi pthread"
$ mpirun -np 16 eztrace mpiapp   or   $ eztrace pthreadapp
```



EZTrace

EZTrace is a trace generator:

- Look out for some events during execution and record them in a file
- Convert them to a standard format

```
$ export EZTRACE_TRACE="mpi pthread"
$ mpirun -np 16 eztrace mpiapp   or   $ eztrace pthreadapp
```

EZTrace log files
(/tmp/<username>_eztrace_log_rank_<rank>)

```
$ eztrace_convert -t OTF -o f.otf /tmp/<username>_eztrace_log*
```

EZTrace

EZTrace is a trace generator:

- Look out for some events during execution and record them in a file
- Convert them to a standard format

```
$ export EZTRACE_TRACE="mpi pthread"
$ mpirun -np 16 eztrace mpiapp   or   $ eztrace pthreadapp
```

EZTrace log files
(/tmp/<username>_eztrace_log_rank_<rank>)

```
$ eztrace_convert -t OTF -o f.otf /tmp/<username>_eztrace_log*
```

OTF File (or Paje)

EZTrace

EZTrace is a trace generator:

- Look out for some events during execution and record them in a file
- Convert them to a standard format
- View the events

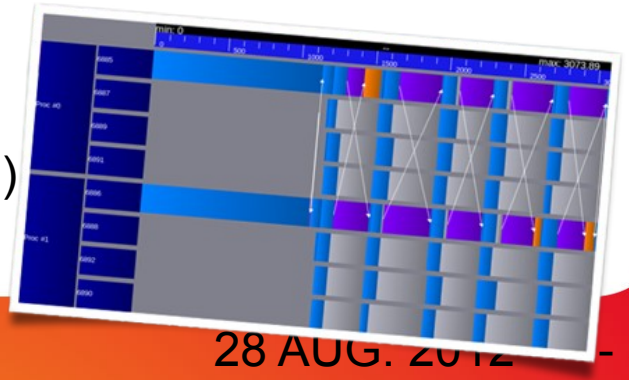
```
$ export EZTRACE_TRACE="mpi pthread"
$ mpirun -np 16 eztrace mpiapp   or   $ eztrace pthreadapp
```

EZTrace log files
(/tmp/<username>_eztrace_log_rank_<rank>)

```
$ eztrace_convert -t OTF -o f.otf /tmp/<username>_eztrace_log*
```

OTF File (or Paje)

Visualizer
(Vampir, ViTE)



2

Standard approaches for automatic instrumentation

Instrumentation at compile-time

- Binary is recompiled with a compiler stub
  - Each call to a function of interest is replaced by a call to a wrapper function
  - The wrapper function records events
- Pros:
  - Easy to conceive
- Cons:
  - Recompilation is painful:
    compilation time when tuning the instrumentation,
    cross-compilation time,
    source-code availability...

Instrumentation using binary modification

- Binary is edited on the hard drive or in memory
  - Binary is modified by inserting probes or trampolines

Instrumentation using binary modification

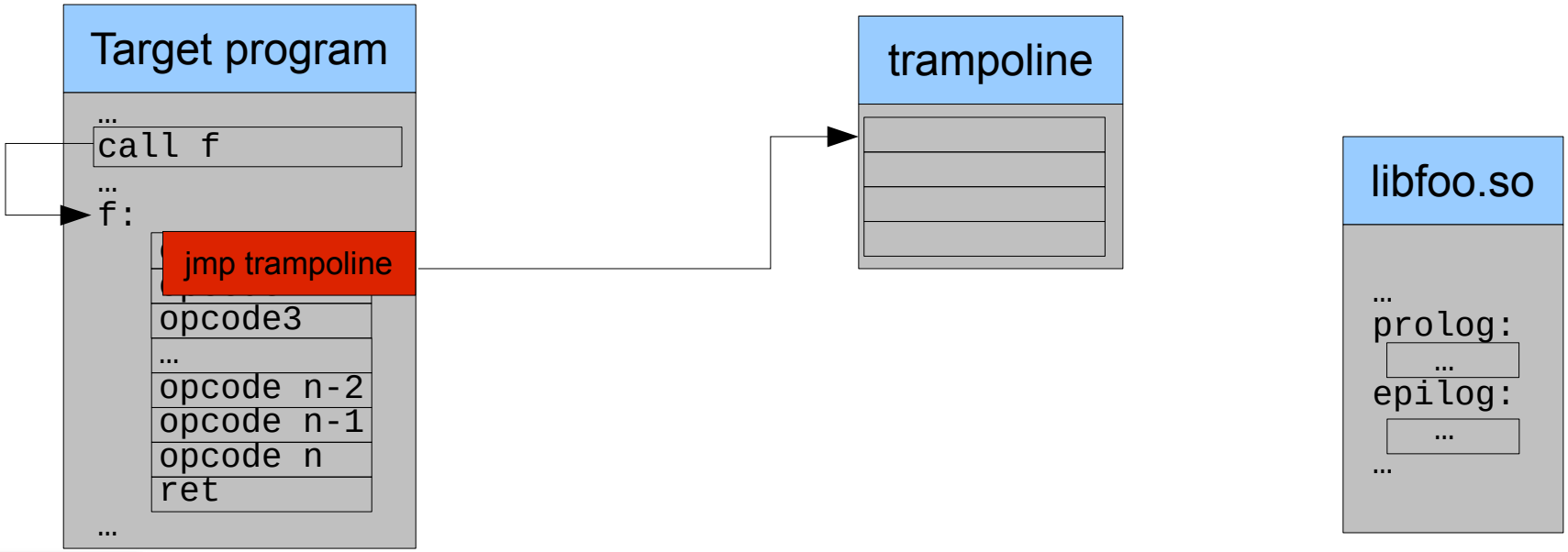
- Binary is edited on the hard drive or in memory
  - Binary is modified by inserting probes or trampolines

Example: the DynInst way

Instrumentation using binary modification

- Binary is edited on the hard drive or in memory
  - Binary is modified by inserting probes or trampolines

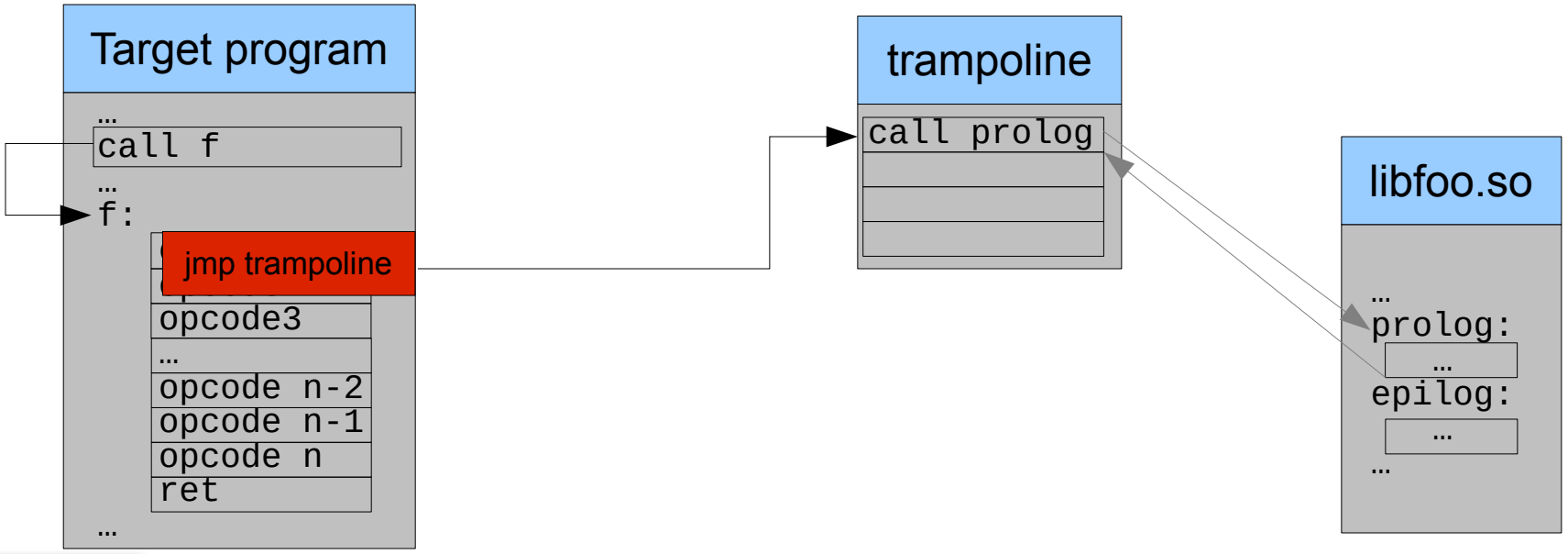
Example: the DynInst way



Instrumentation using binary modification

- Binary is edited on the hard drive or in memory
  - Binary is modified by inserting probes or trampolines

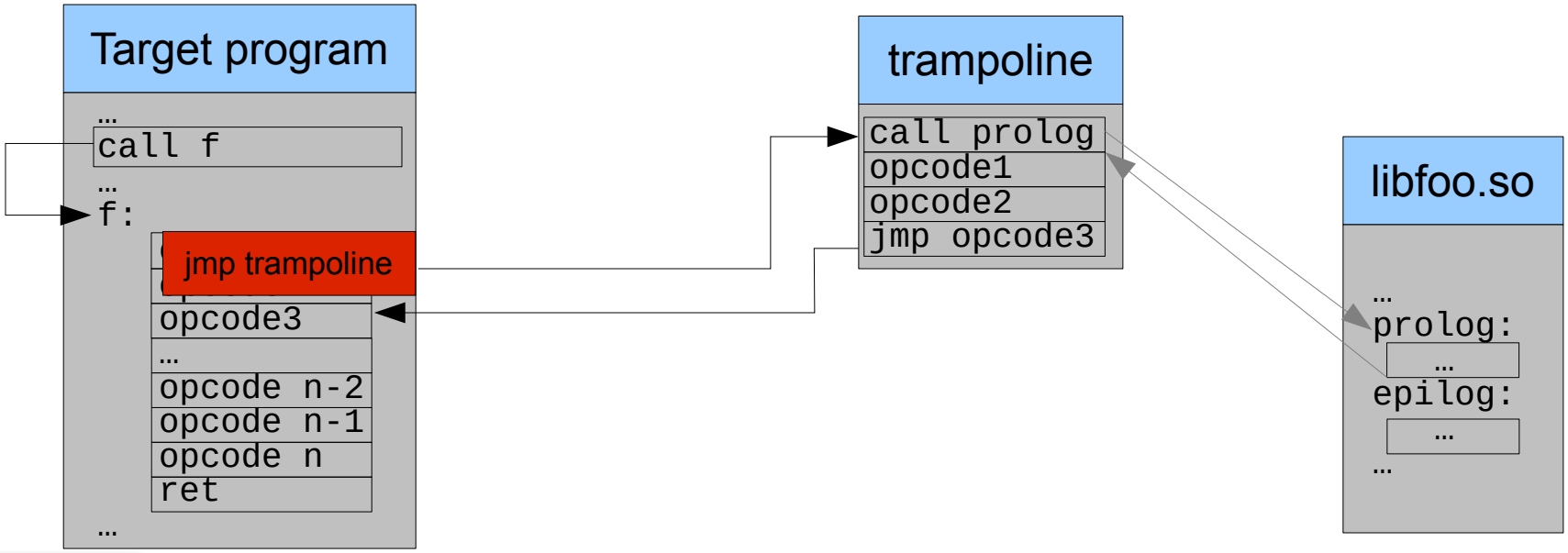
Example: the DynInst way



Instrumentation using binary modification

- Binary is edited on the hard drive or in memory
  - Binary is modified by inserting probes or trampolines

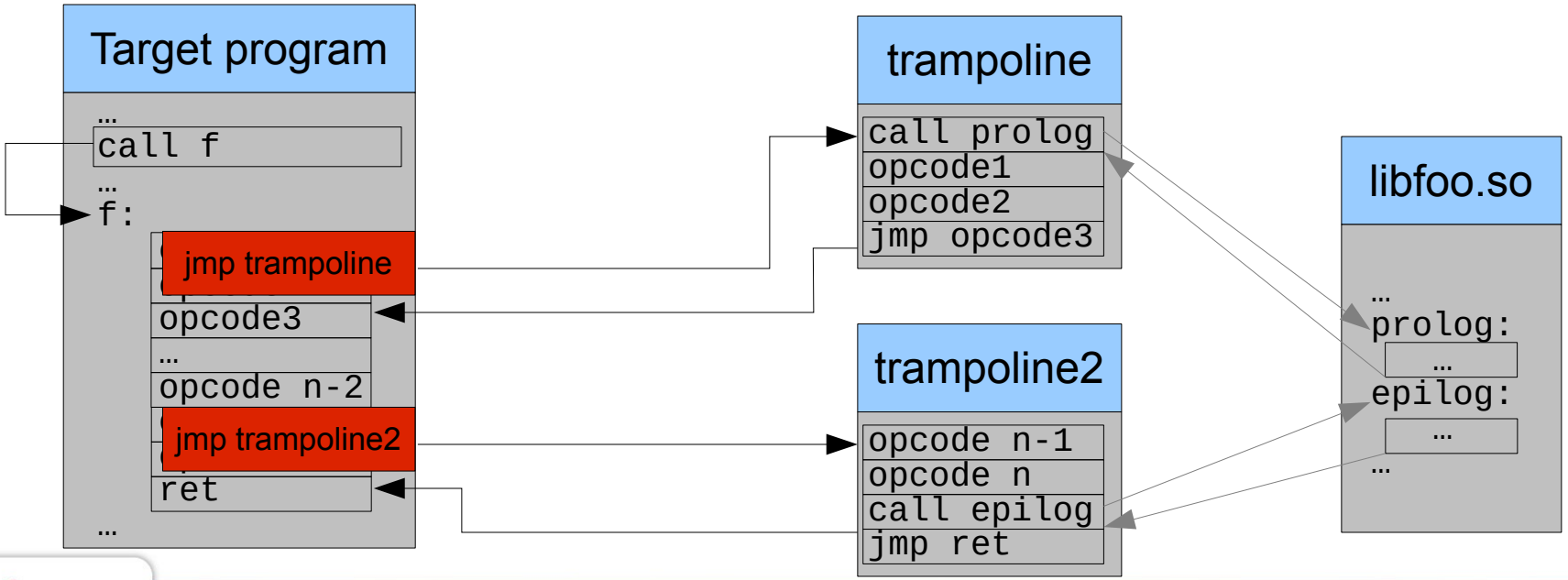
Example: the DynInst way



Instrumentation using binary modification

- Binary is edited on the hard drive or in memory
  - Binary is modified by inserting probes or trampolines

Example: the DynInst way

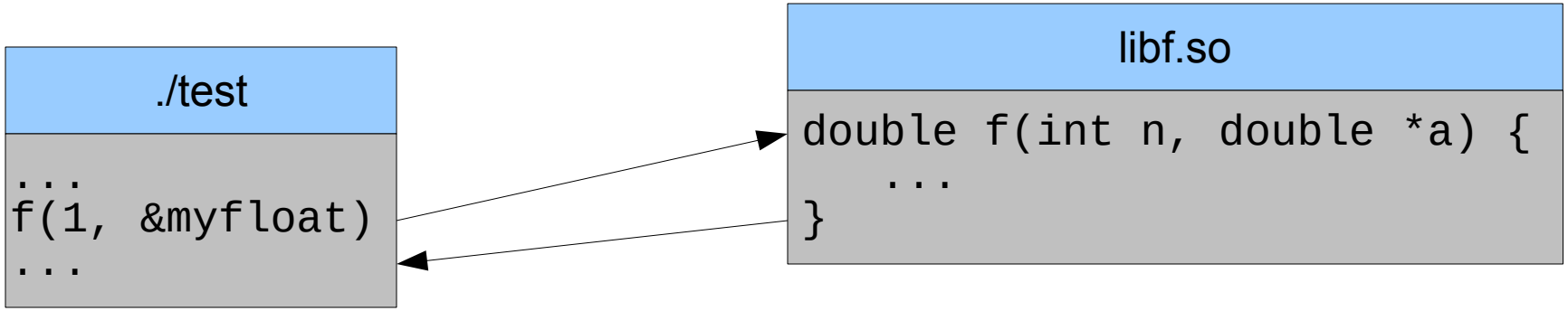


Instrumentation using binary modification

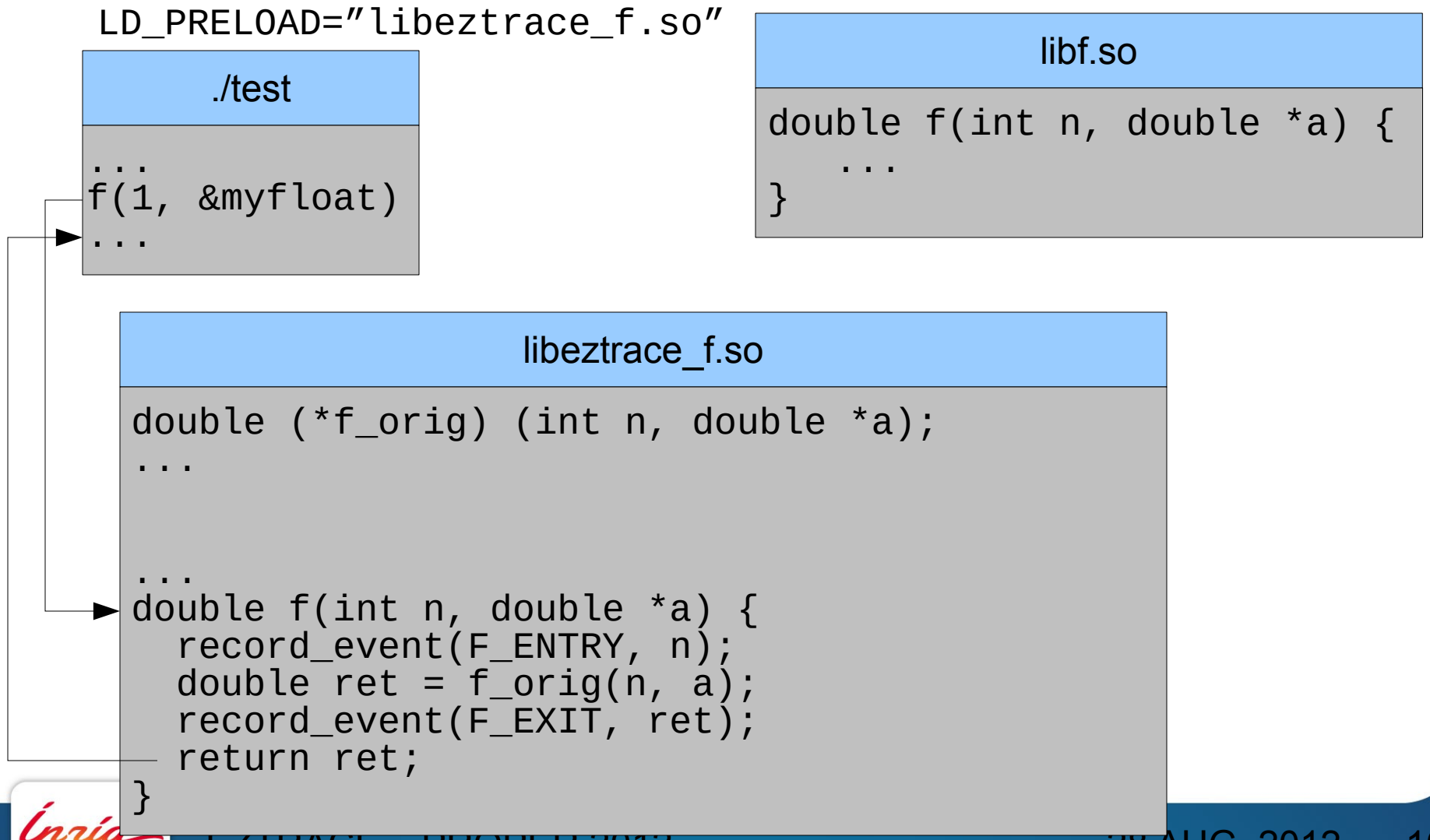
- Binary is edited on the hard drive or in memory
  - Binary is modified by inserting probes or trampolines

- Pros:
  - Efficient
  - Adaptive to all instrumentation needs
- Cons:
  - Relies on complex methods for locating exit points
    decompilation,
    emulation...
  - Libraries supporting this methods are hard to use

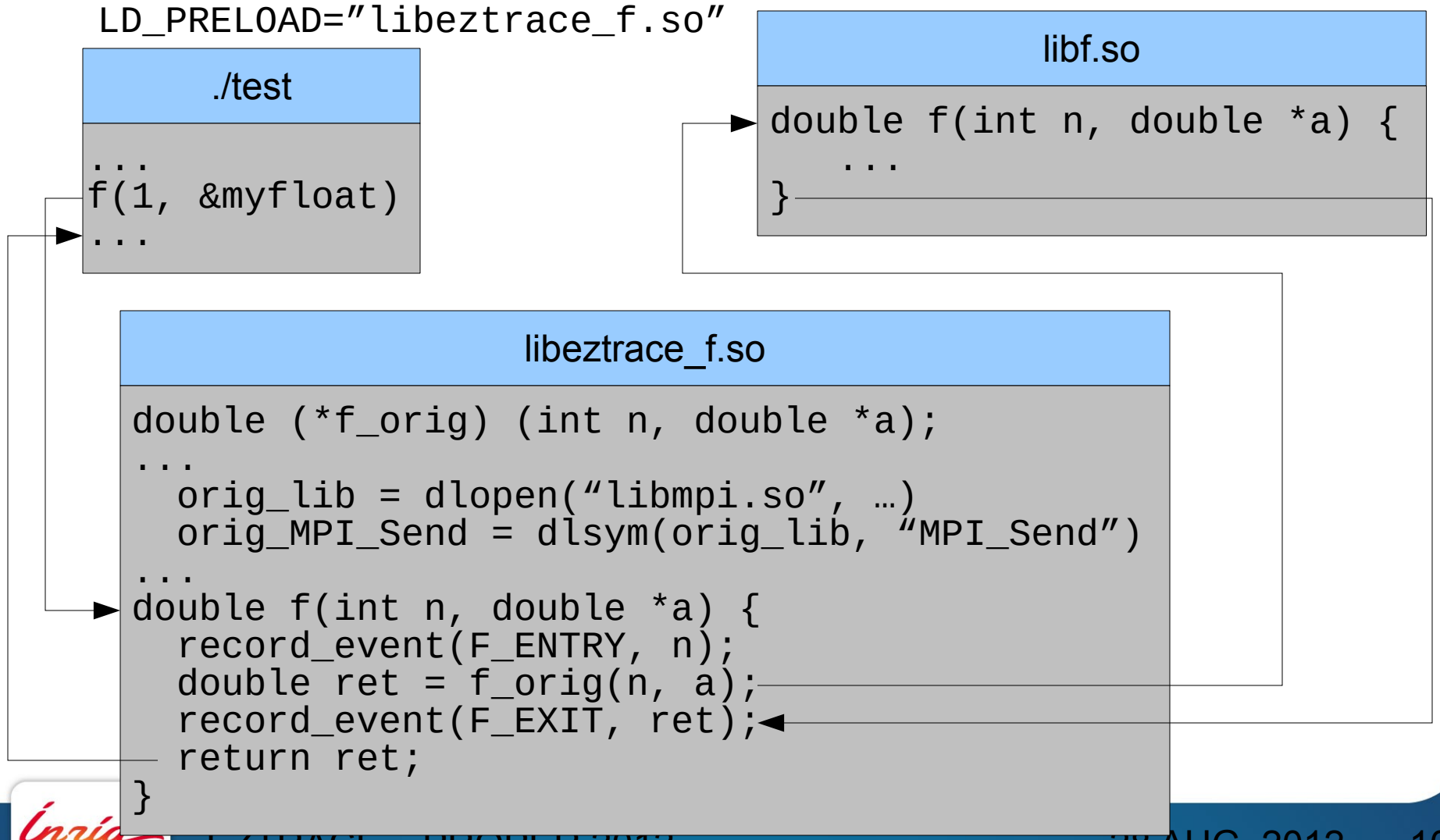
Instrumentation at runtime using LD_PRELOAD (EZTrace)



Instrumentation at runtime using LD_PRELOAD (EZTrace)



Instrumentation at runtime using LD_PRELOAD (EZTrace)



Instrumentation at runtime using LD_PRELOAD

- Pros:
  - Easy to use:
    C functions
    simple interface
  - Easy to run
  - Easy to do
- Cons:
  - Only functions
  - Only in dynamic libraries

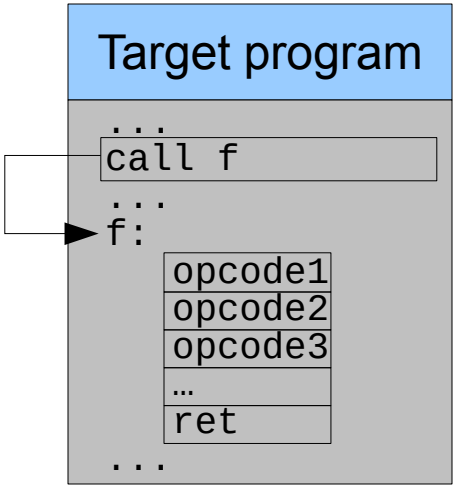
3

Function instrumentation in EZTrace

Function instrumentation in EZTrace

- Function are instrumented using plugins at the application startup
  - Function in dynamic library are intercepted using LD_PRELOAD
  - Statically linked functions are instrumented by modifying the target program in memory
  - EZTrace determines automatically the method to use

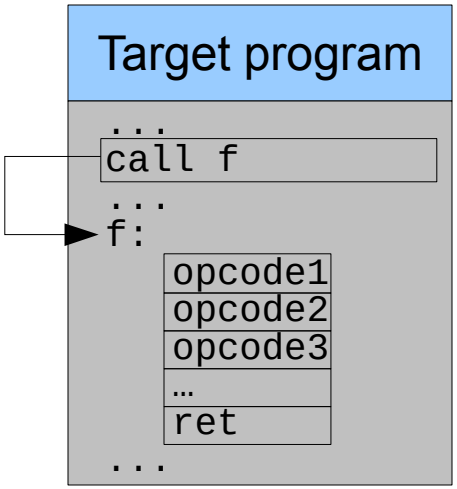
Statically-linked functions instrumentation



libeztrace_f.so

```
double (*f_orig) (int n, double *a);
double f(int n, double *a) {
  record_event(F_ENTRY, n);
  double ret = f_orig(n, a);
  record_event(F_EXIT, ret);
  return ret;
}
```

Statically-linked functions instrumentation

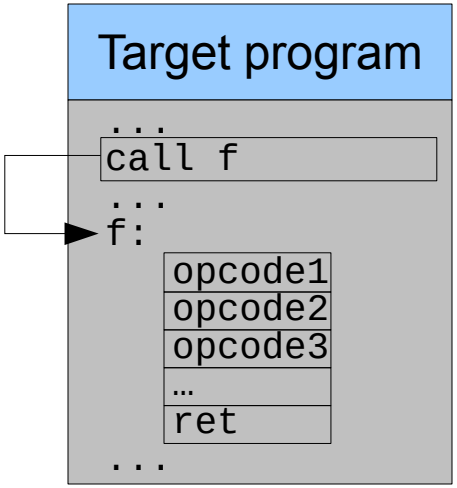


libeztrace_f.so

```
double (*f_orig) (int n, double *a);
double f(int n, double *a) {
  record_event(F_ENTRY, n);
  double ret = f_orig(n, a);
  record_event(F_EXIT, ret);
  return ret;
}
```

Insertion of the library in target memory using LD_PRELOAD

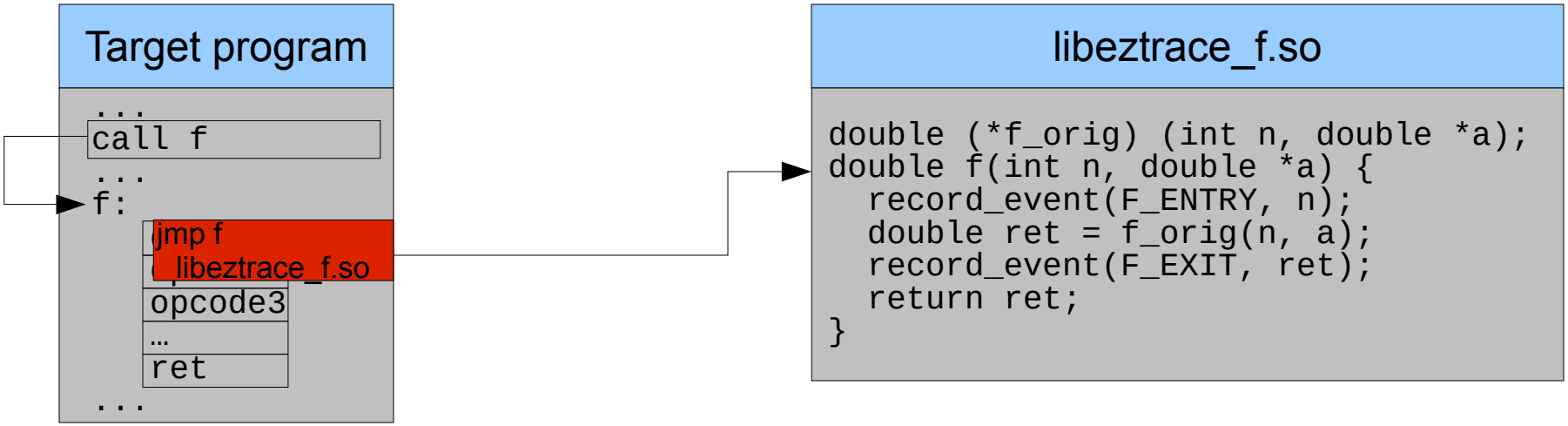
Statically-linked functions instrumentation



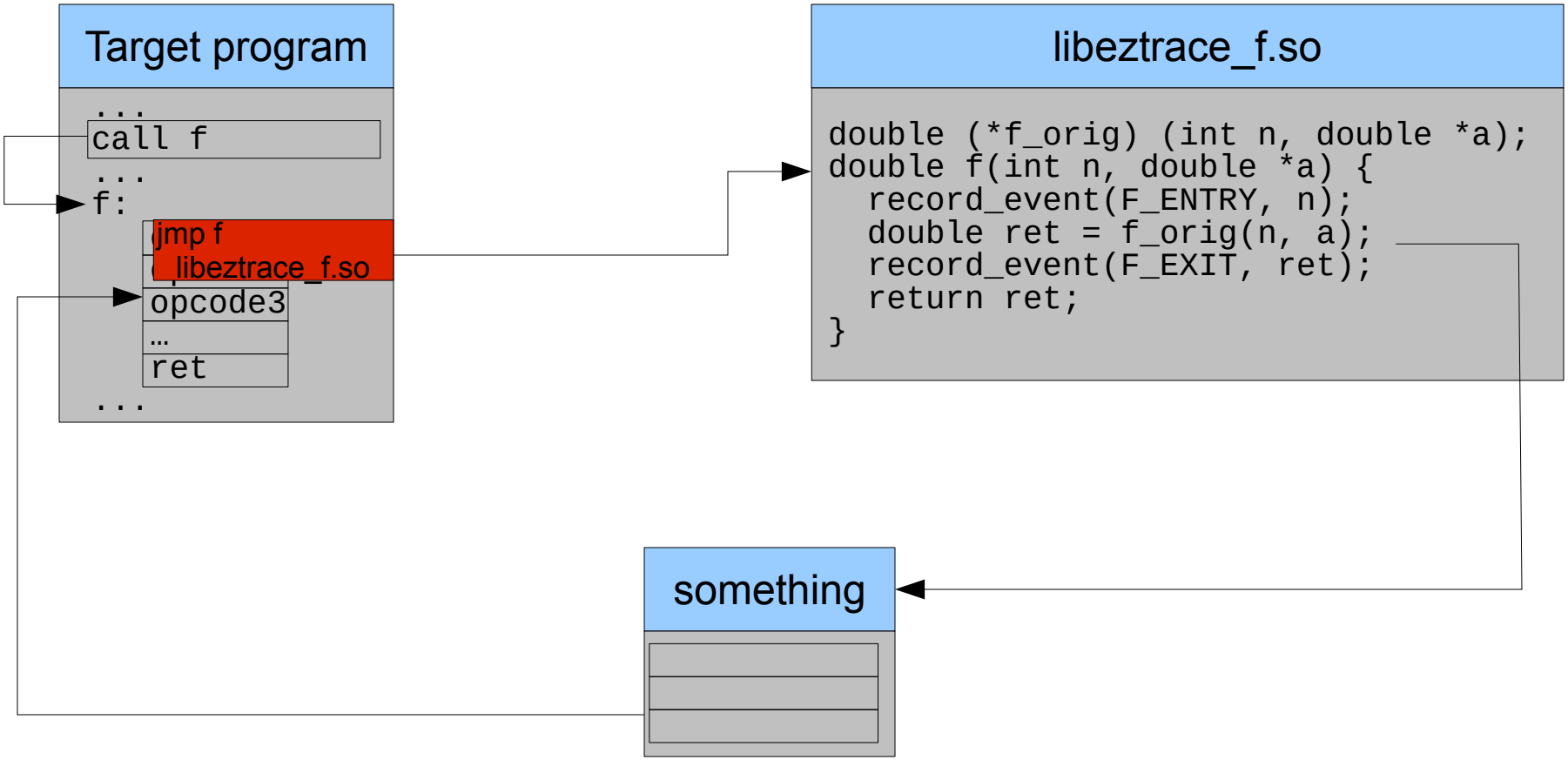
libeztrace_f.so

```
double (*f_orig) (int n, double *a);
double f(int n, double *a) {
  record_event(F_ENTRY, n);
  double ret = f_orig(n, a);
  record_event(F_EXIT, ret);
  return ret;
}
```

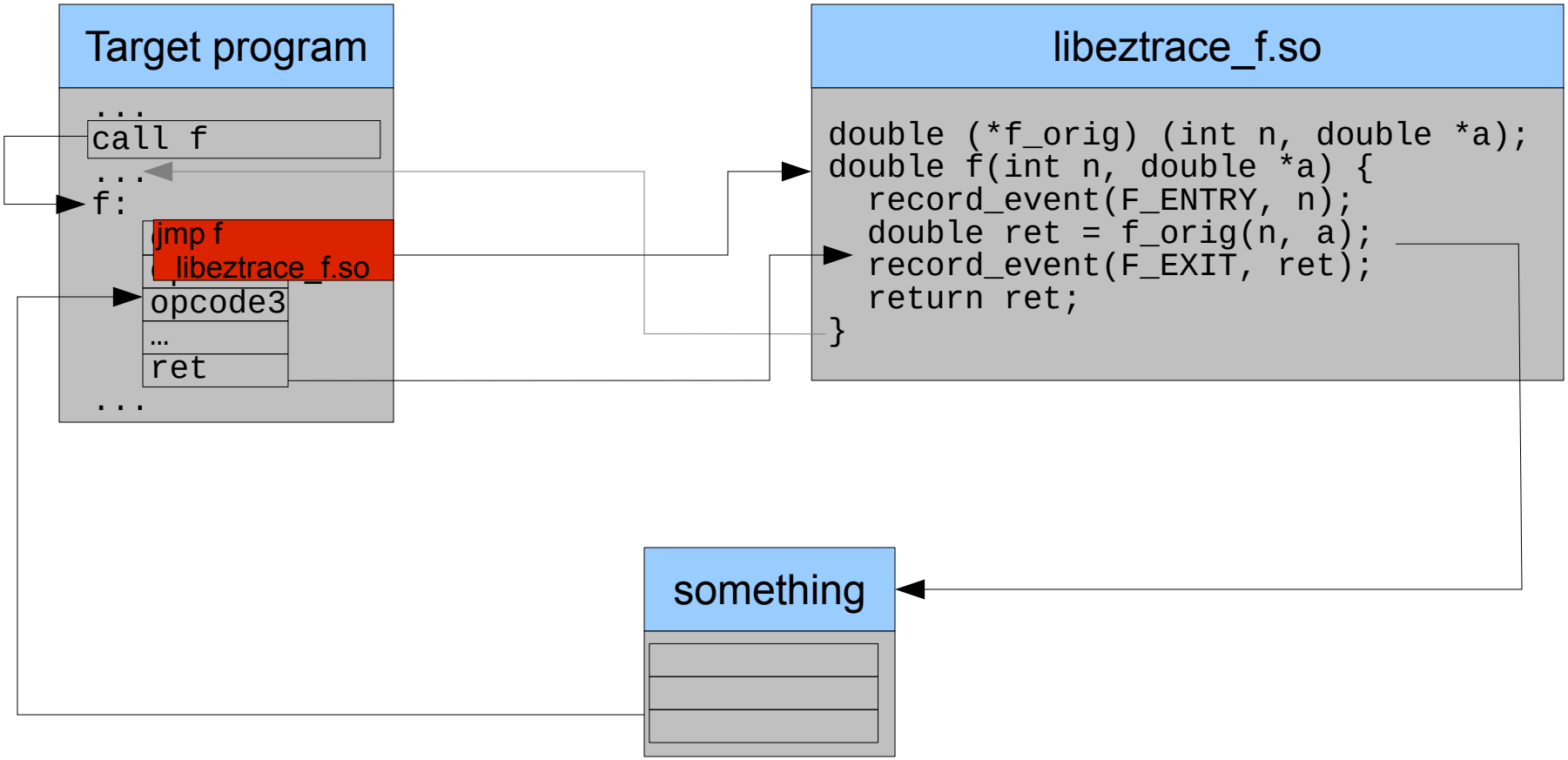
Statically-linked functions instrumentation



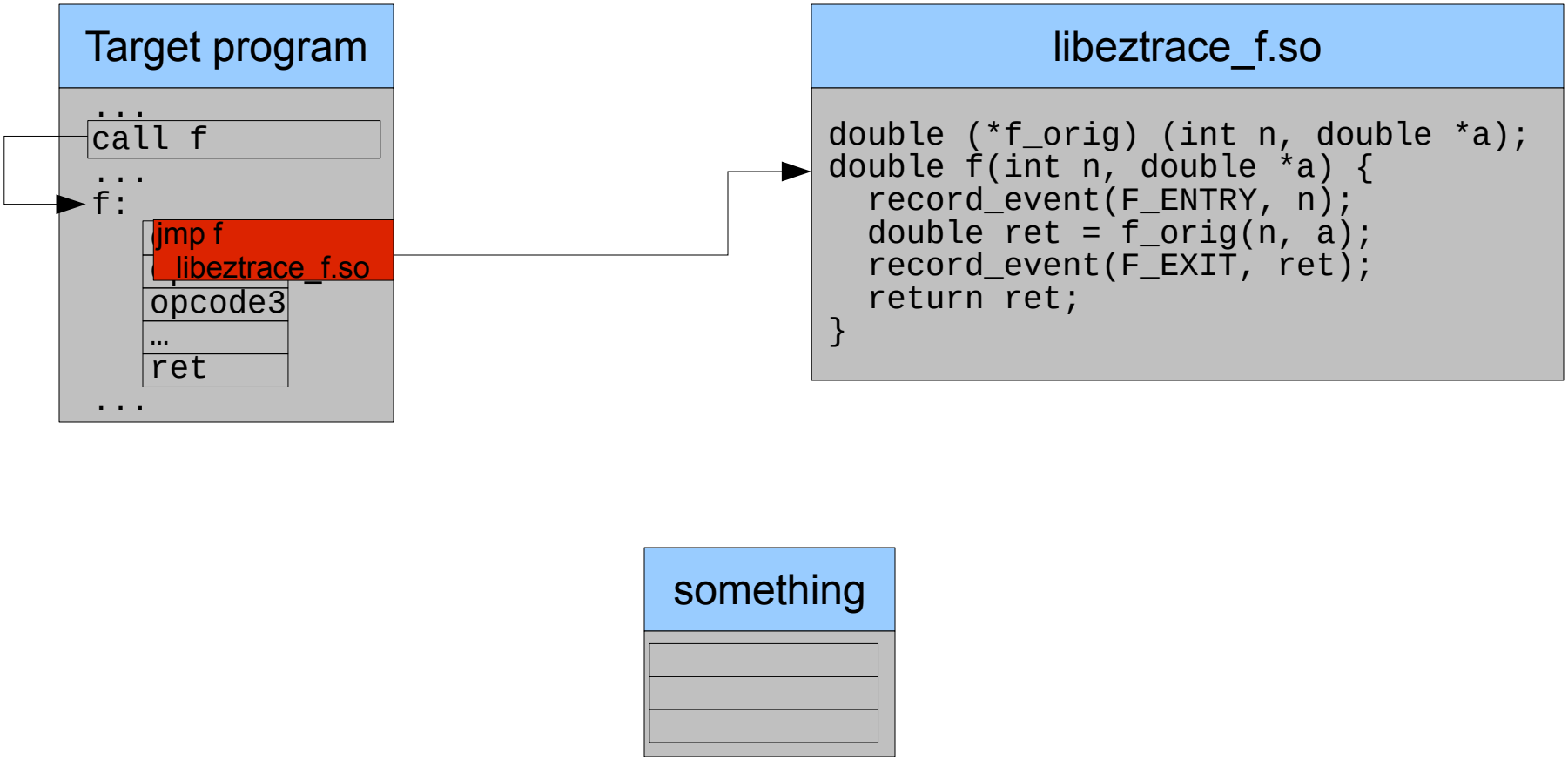
Statically-linked functions instrumentation



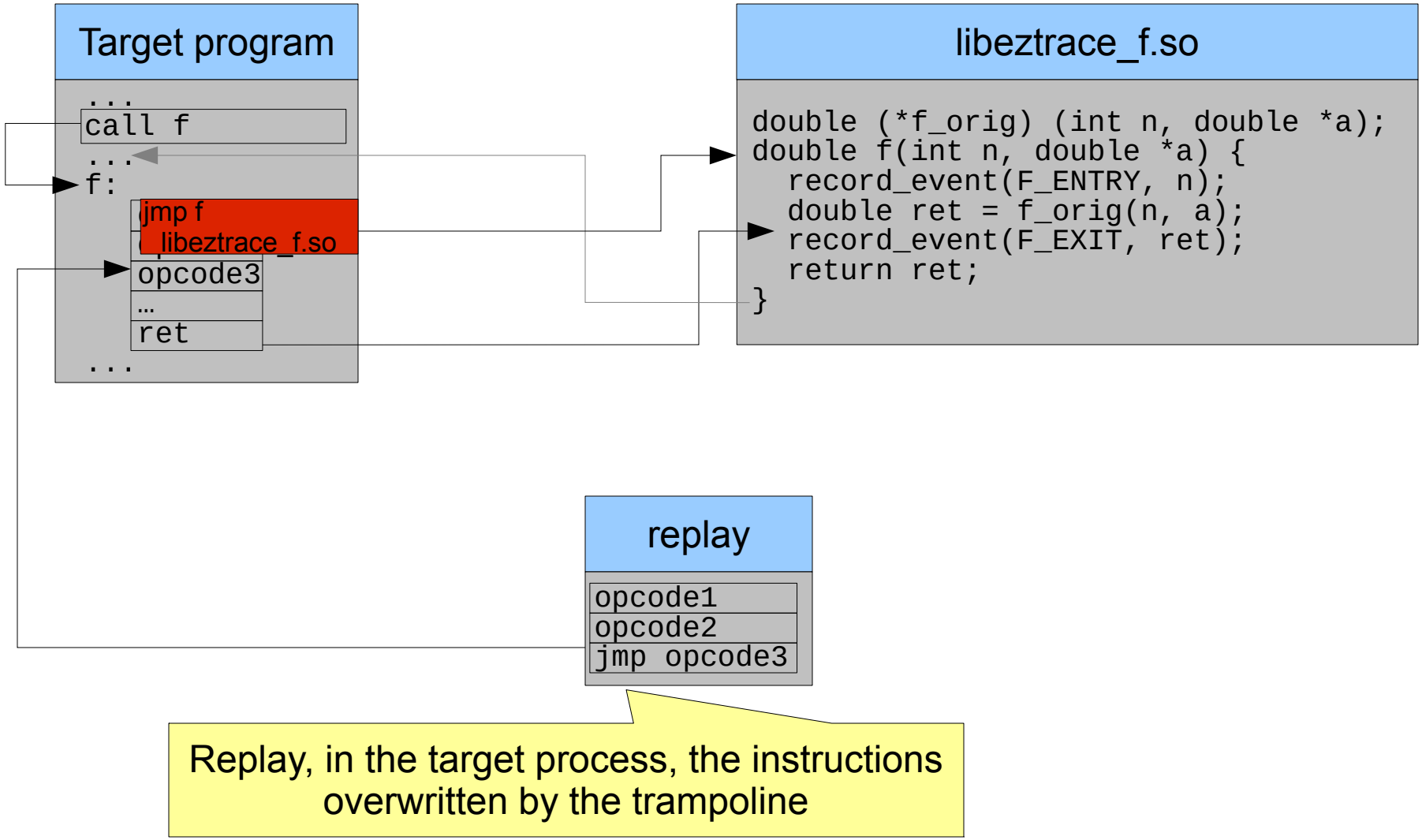
Statically-linked functions instrumentation



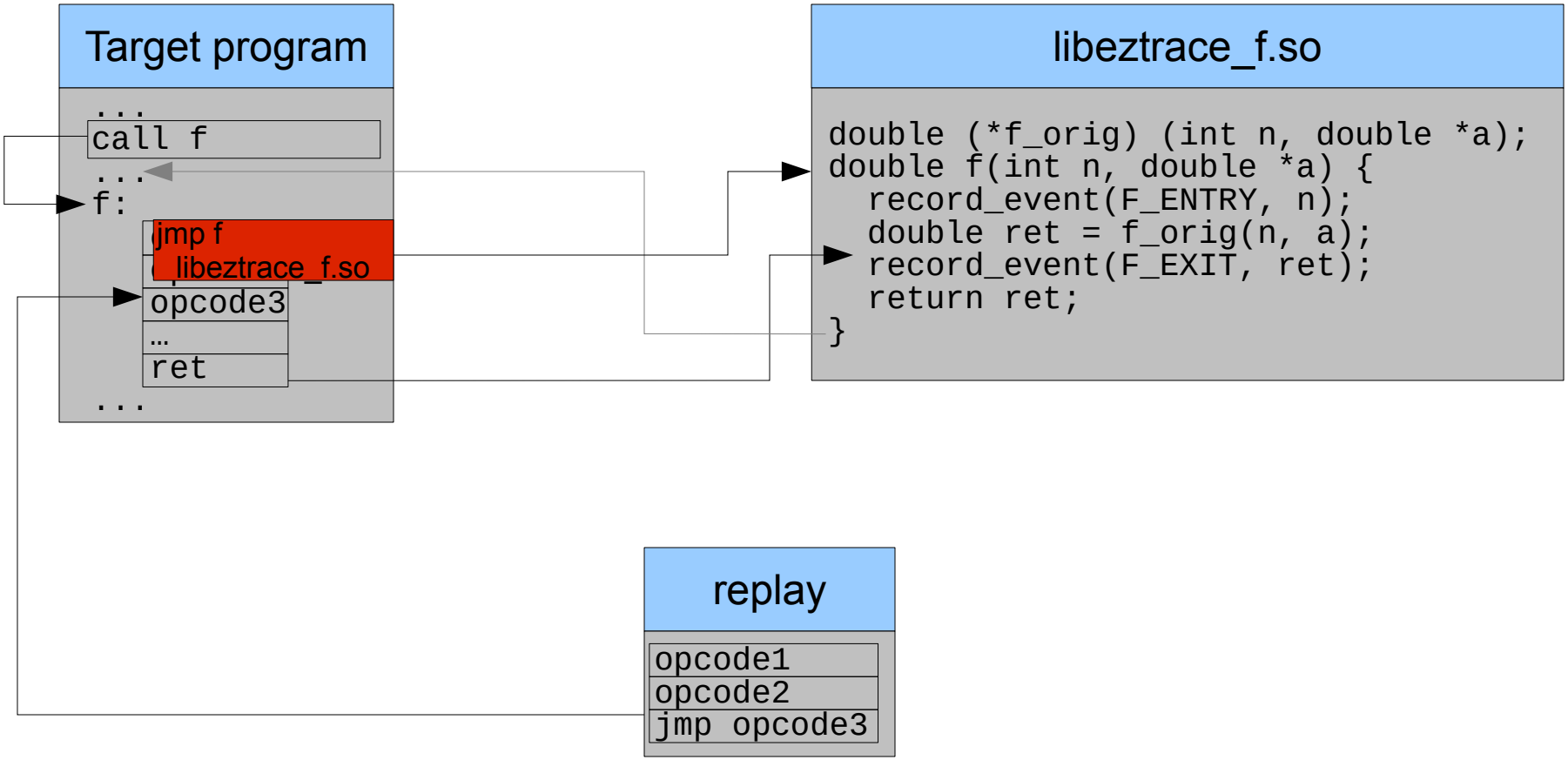
Statically-linked functions instrumentation



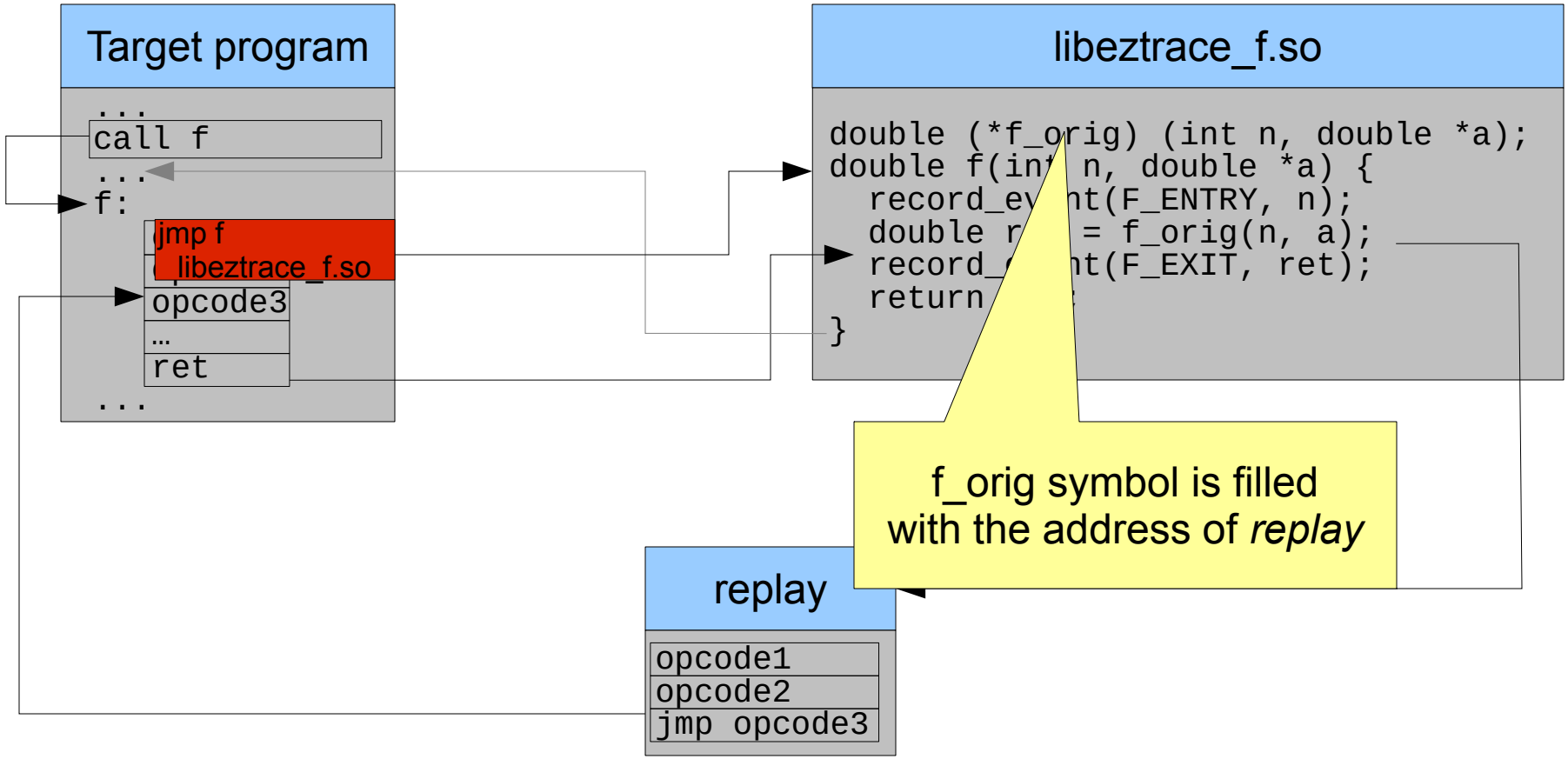
Statically-linked functions instrumentation



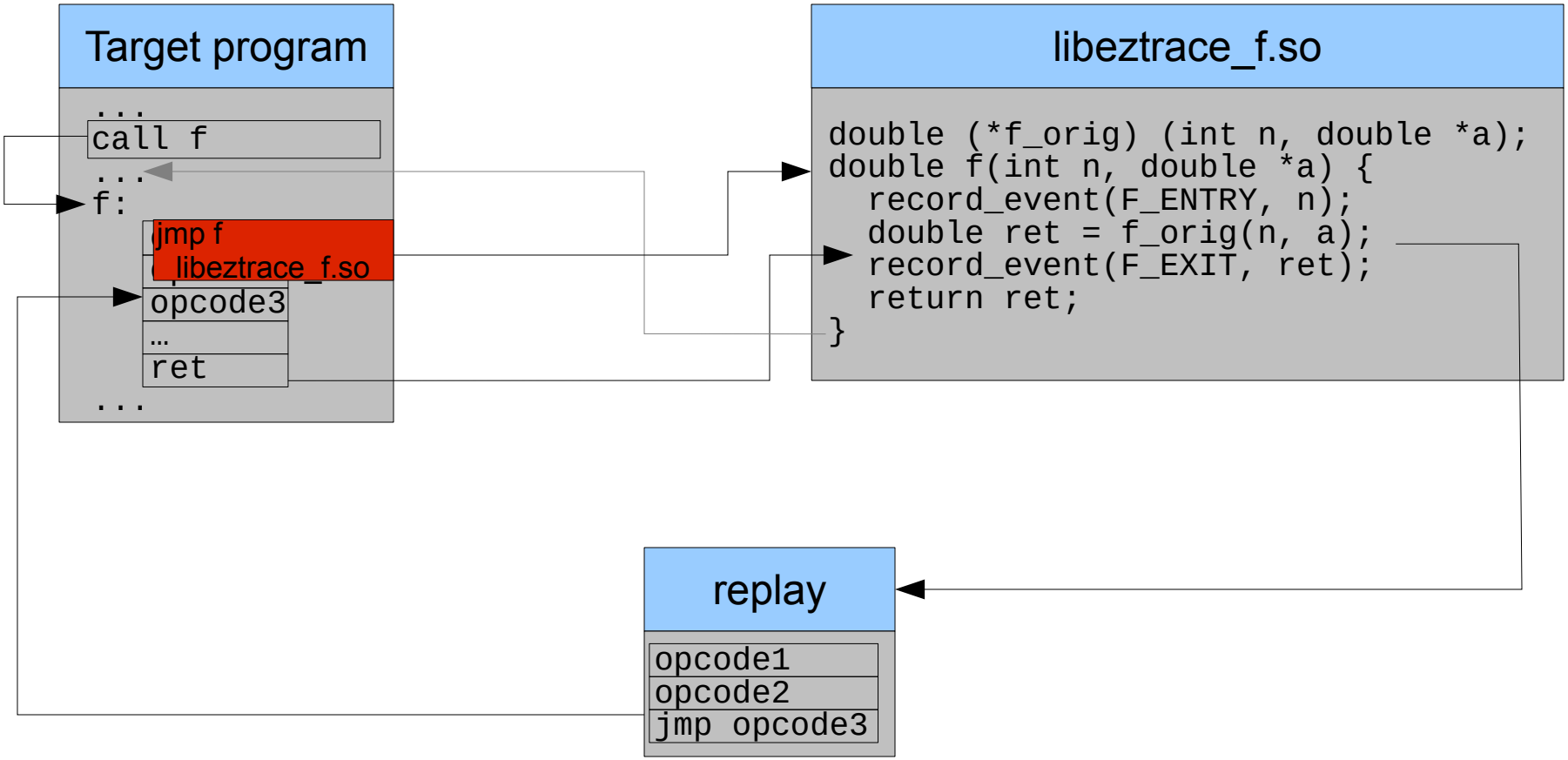
Statically-linked functions instrumentation



Statically-linked functions instrumentation



Statically-linked functions instrumentation



Function instrumentation in EZTrace

- Function are instrumented using plugins at the application startup
  - Function in dynamic library are intercepted using LD_PRELOAD
  - Statically linked functions are instrumented by modifying the target program in memory
  - EZTrace determines automatically the method to use

- Pros:
  - Efficient and simple
  - Easy to use: C function and EZTrace plugin generator
  - Easy to extends: little architecture-dependent code
- Cons:
  - Limited to functions entries and exits

4

Evaluation

Evaluation

- Comparison between
  - EZTrace 0.8
  - DynInst 7.1
  - PIN 2.11
- Platform: Intel Xeon X5550 at 2.67 Ghz

Evaluation: RAW overhead

```
t1 = get_time();
for (i = 0; i < NB_LOOPS; i++)
  compute();
t2 = get_time();
print((t2 - t1) / NB_LOOPS);
```

- Instrument the entry and exit of compute

Instrumentation method	No instrumentation	PIN	DynInst	EZTrace
Statically linked library	4.7 ns	20.2 ns	28.8 ns	12.3 ns
Shared library	5.2 ns	24.0 ns	–	11.3 ns

Evaluation: RAW overhead

```
t1 = get_time();
for (i = 0; i < NB_LOOPS; i++)
  compute();
t2 = get_time();
print((t2 - t1) / NB_LOOPS);
```

- Instrument the entry and exit of compute

Instrumentation method	No instrumentation	PIN	DynInst	EZTrace
Statically linked library	4.7 ns	20.2 ns	28.8 ns	12.3 ns
Shared library	5.2 ns	24.0 ns	–	11.3 ns

- Record an event at the entry/exit of compute

Instrumentation method	No instrumentation	PIN	DynInst	EZTrace
Statically linked library	4.7 ns	1287 ns	1294 ns	245.2 ns
Shared library	5.3 ns	1293 ns	–	227.4 ns

Evaluation: overhead on an application

- Molecular dynamics simulation
  - OpenMP
  - Instrumentation of statically-linked functions only
  - 7 941 671 events to record

Instrumentation method	No instrumentation	PIN	DynInst	EZTrace
Execution time	0.45 s	3.16 s	3.28 s	2.28 s
Overhead (ns / iteration)	+0	+341	+413	+227

5
Conclusion

Conclusion

- EZTrace: framework for performance analysis
  - How to instrument statically-linked applications?
- Coarse-grain instrumentation
  - Modify the process address space
  - Reroute the processing flow
- Benefits
  - Low overhead instrumentation
  - No need to modify existing modules

Thank you!

http://eztrace.gforge.inria.fr

Special thanks to Julien Pedron