Playing with process tracing to instrument static function at runtime in EZTrace

Damien Martin-Guillerez
SED
CENTRE Inria
BORDEAUX SUD-OUEST

14 FEB. 2012

INTRODUCTION

EZTrace is a performance trace generator for parallel programs

INTRODUCTION

EZTrace is a performance trace generator for parallel programs

It relies on the **dynamic library preloading** mechanism to **intercept function calls** and insert event logs

INTRODUCTION

EZTrace is a performance trace generator for parallel programs

It relies on the **dynamic library preloading** mechanism to **intercept function calls** and insert event logs

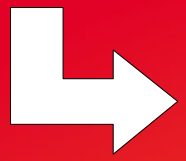
As a consequence, **no instrumentation of static function**, i.e. no trace for them

INTRODUCTION

EZTrace is a performance trace generator for parallel programs

It relies on the **dynamic library preloading** mechanism to **intercept function calls** and insert event logs

As a consequence, **no instrumentation of static function**, i.e. no trace for them



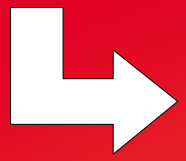
PROCESS TRACING

INTRODUCTION

EZTrace is a performance trace generator for parallel programs

It relies on the **dynamic library preloading** mechanism to **intercept function calls** and insert event logs

As a consequence, **no instrumentation of static function**, i.e. no trace for them



PROCESS TRACING

OUTLINE

1. EZTrace insights
2. The ptrace() system call
3. Instrumentation of static functions in EZTrace
4. Technical details for the geek
5. Other architectures...

1
EZTrace Insights

EZTrace basics

EZTrace is a trace generator:

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

EZTrace basics

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 -e mpiapp`   or   $ eztrace pthreadapp
```

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

EZTrace basics

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 -e mpiapp`   or   $ eztrace pthreadapp
```

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

```
$ eztrace_convert -o myfile.paje /tmp/<username>_eztrace_log*
```

EZTrace basics

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 -e mpiapp`   or   $ eztrace pthreadapp
```

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

```
$ eztrace_convert -o myfile.paje /tmp/<username>_eztrace_log*
```

Paje File (or OTF)

EZTrace basics

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

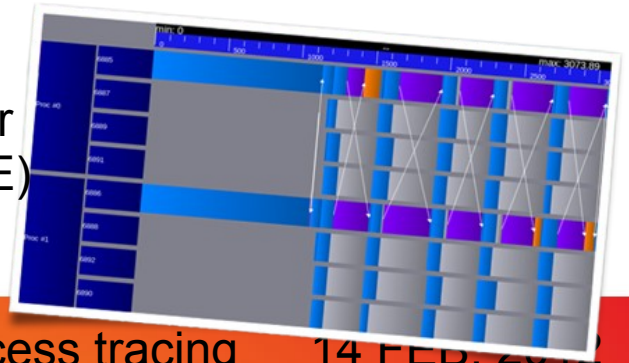
```
$ export EZTRACE_TRACE="mpi pthread"
$ mpirun -np 16 `eztrace -e mpiapp`   or   $ eztrace pthreadapp
```

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

```
$ eztrace_convert -o myfile.paje /tmp/<username>_eztrace_log*
```

Paje File (or OTF)

Visualizer
(ex.: ViTE)



EZTrace basics

EZTrace is a trace generator:

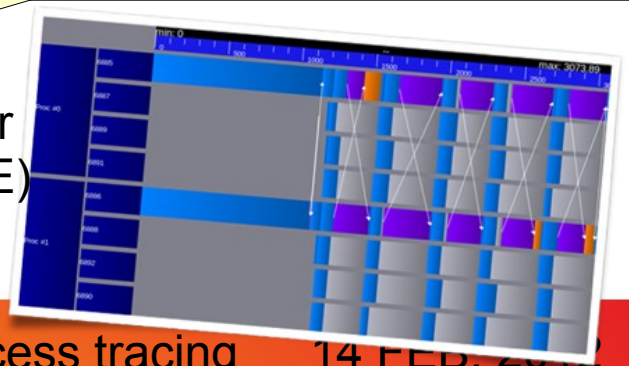
- Look out for some events during execution and rec… e
- Convert them to a standard format
- View the events

```
$ export EZTRACE_TRA…
$ mpirun -np 1 …                                    eadapp
```

ON DYNAMIC LIBRARIES!

```
$ eztrace_…                                    …ame>_eztrace_log*
```

Paje File … → Visualizer (ex.: ViTE)



EZTrace is good for your health

You can use EZTrace to trace parallel programs in order to:

- Detect problems in message exchanges and in parallel algorithm
- Improve communication patterns
- Show beautiful traces of your software to the world

EZTrace is good for your health

You can use EZTrace to trace parallel programs in order to:

- Detect problems in message exchanges and in parallel algorithm
- Improve communication patterns
- Show beautiful traces of your software to the world

You probably should not use EZTrace to:

- Detect general bugs like memory faults
- Trace non-parallel programs
- Show that you program has ugly traces

EZTrace is good for your health

You can use EZTrace to trace parallel programs in order to:

- Detect problems in message exchanges and in parallel algorithm
- Improve communication patterns
- Show beautiful traces of your software to the world

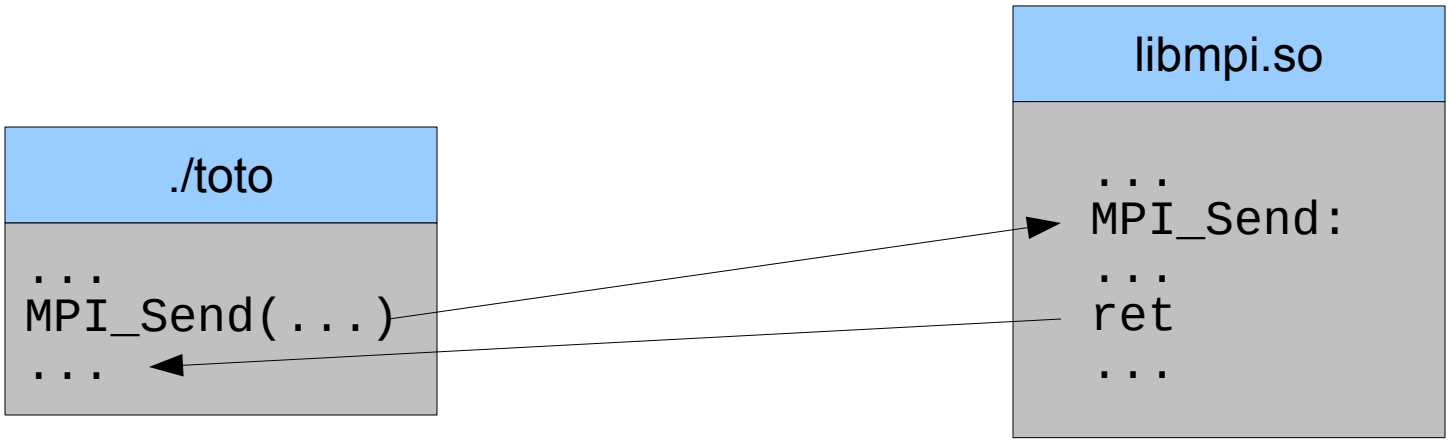
You probably should not use EZTrace to:

- Detect general bugs like memory faults
- Trace non-parallel programs
- Show that you program has ugly traces

You cannot (currently) use EZTrace to:

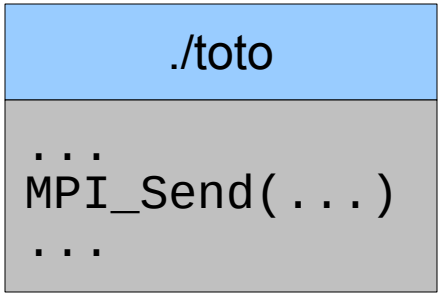
- Stop when a memory error occurs
- Trace your own library or non-packaged library
- Trace static libraries
- Make the coffee

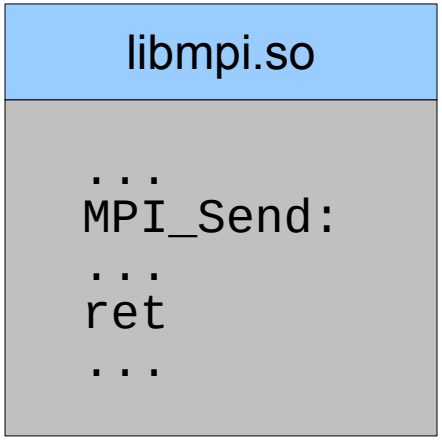
Dynamic library loading



Dynamic library preloading
With an EZTrace module

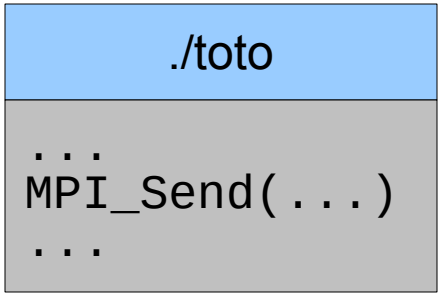
```
LD_PRELOAD=”libeztrace_mpi.so”
```

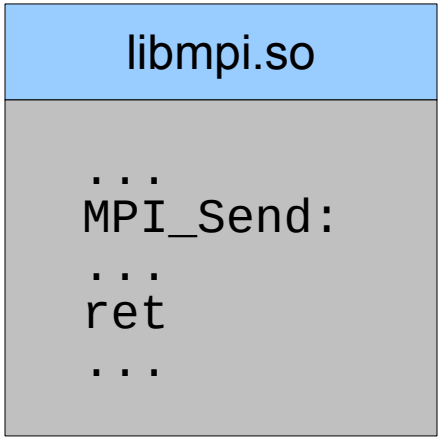




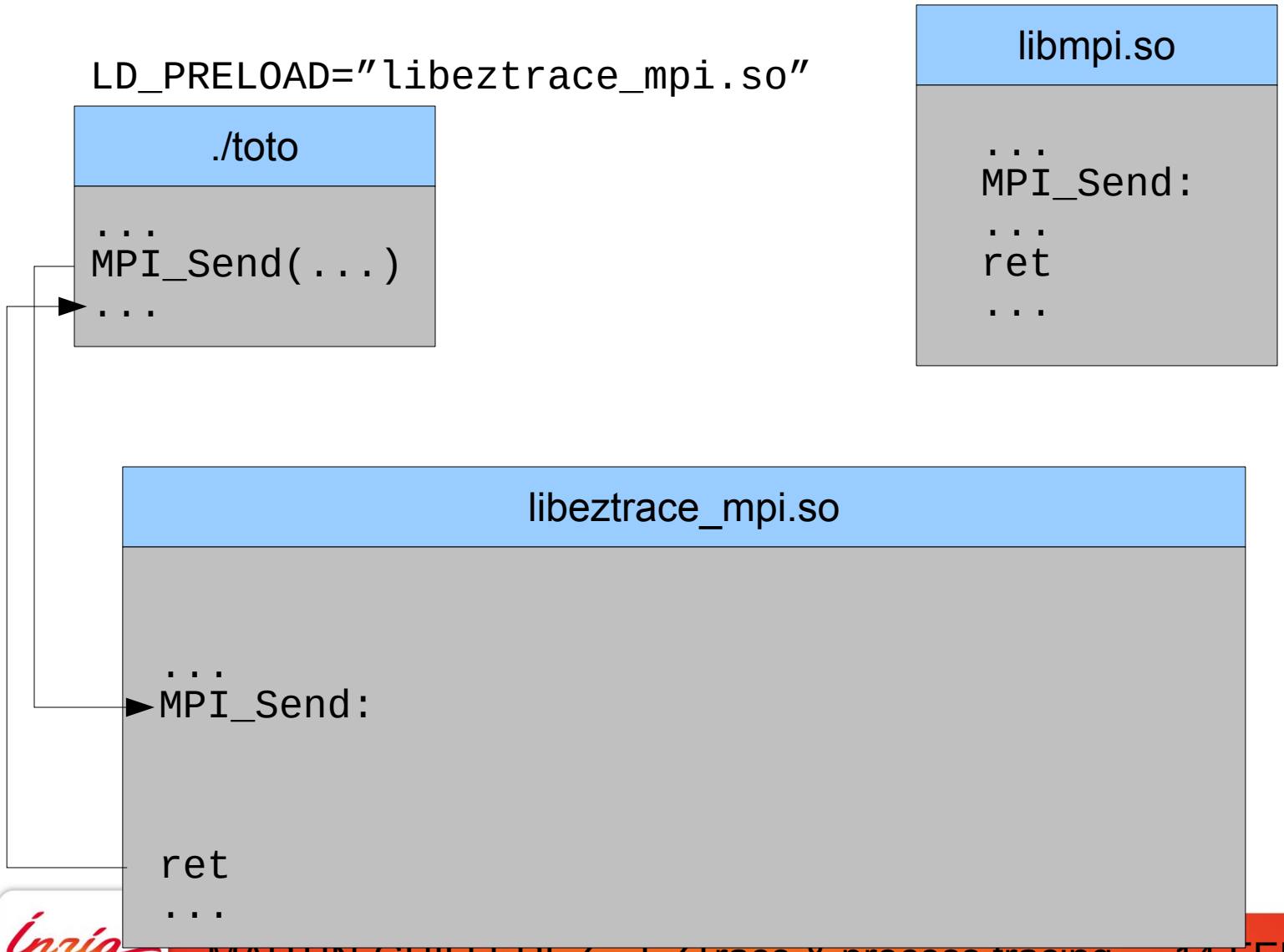
Dynamic library preloading
With an EZTrace module

```
LD_PRELOAD=”libeztrace_mpi.so”
```

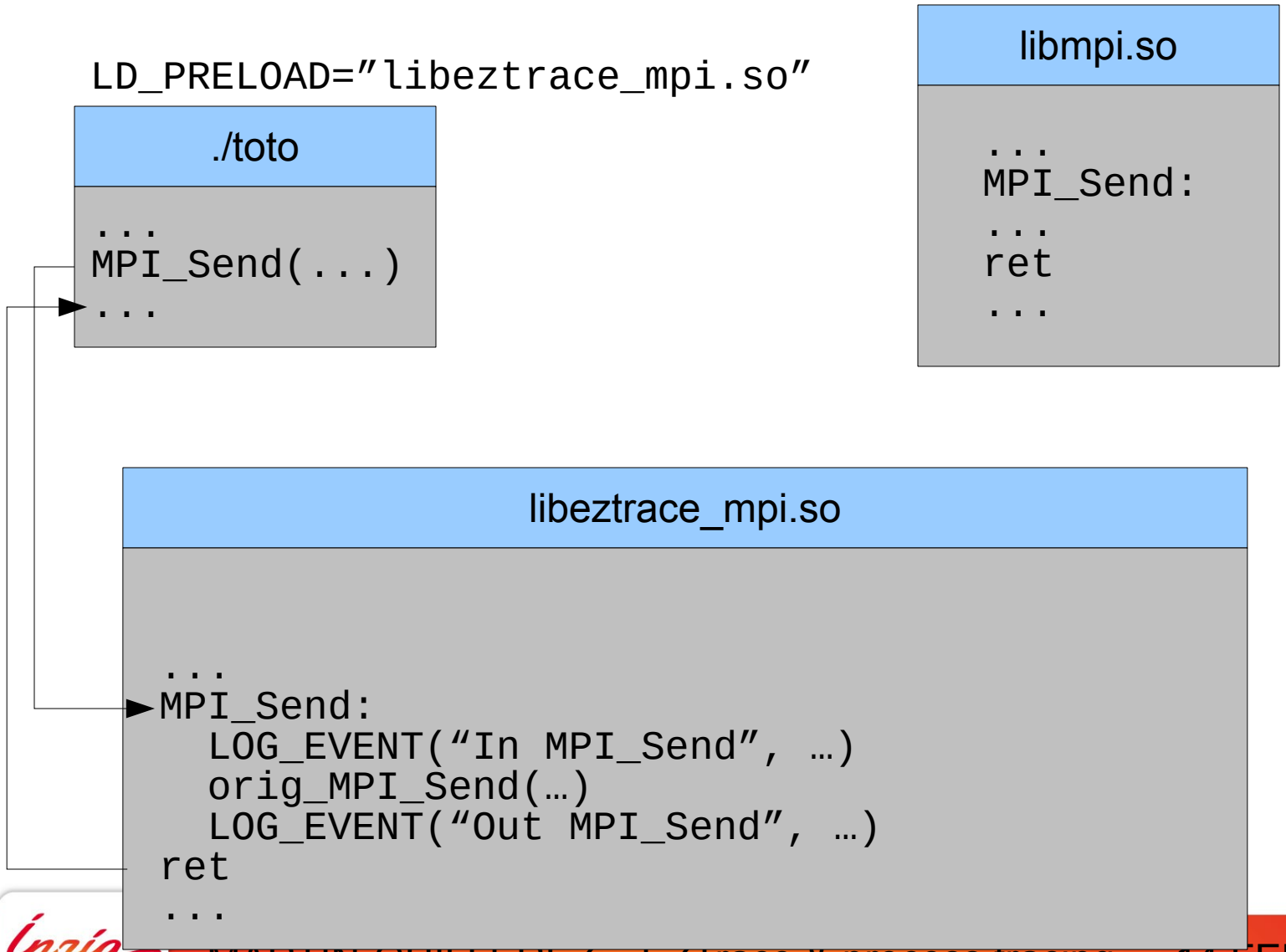




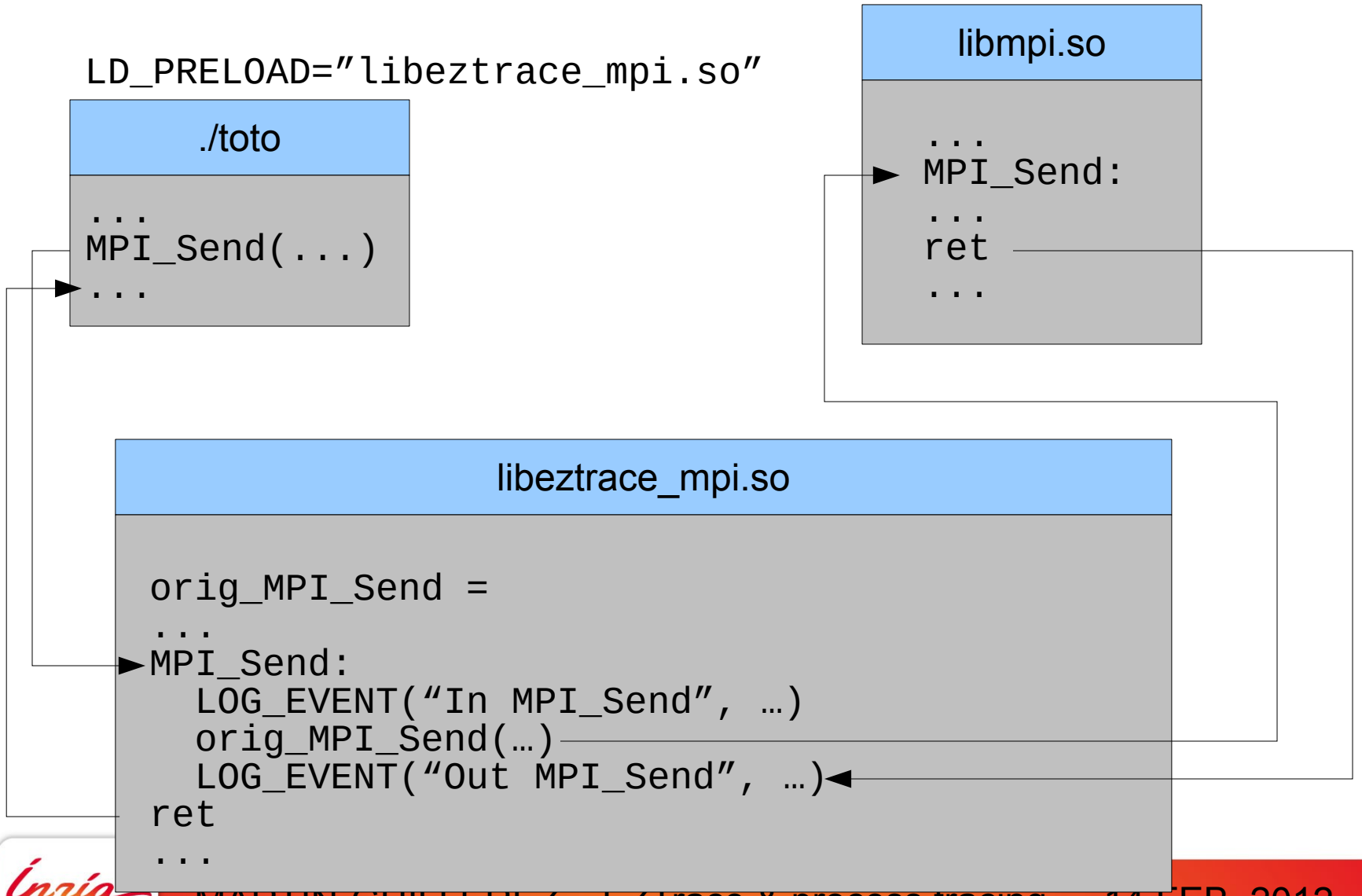
Dynamic library preloading
With an EZTrace module



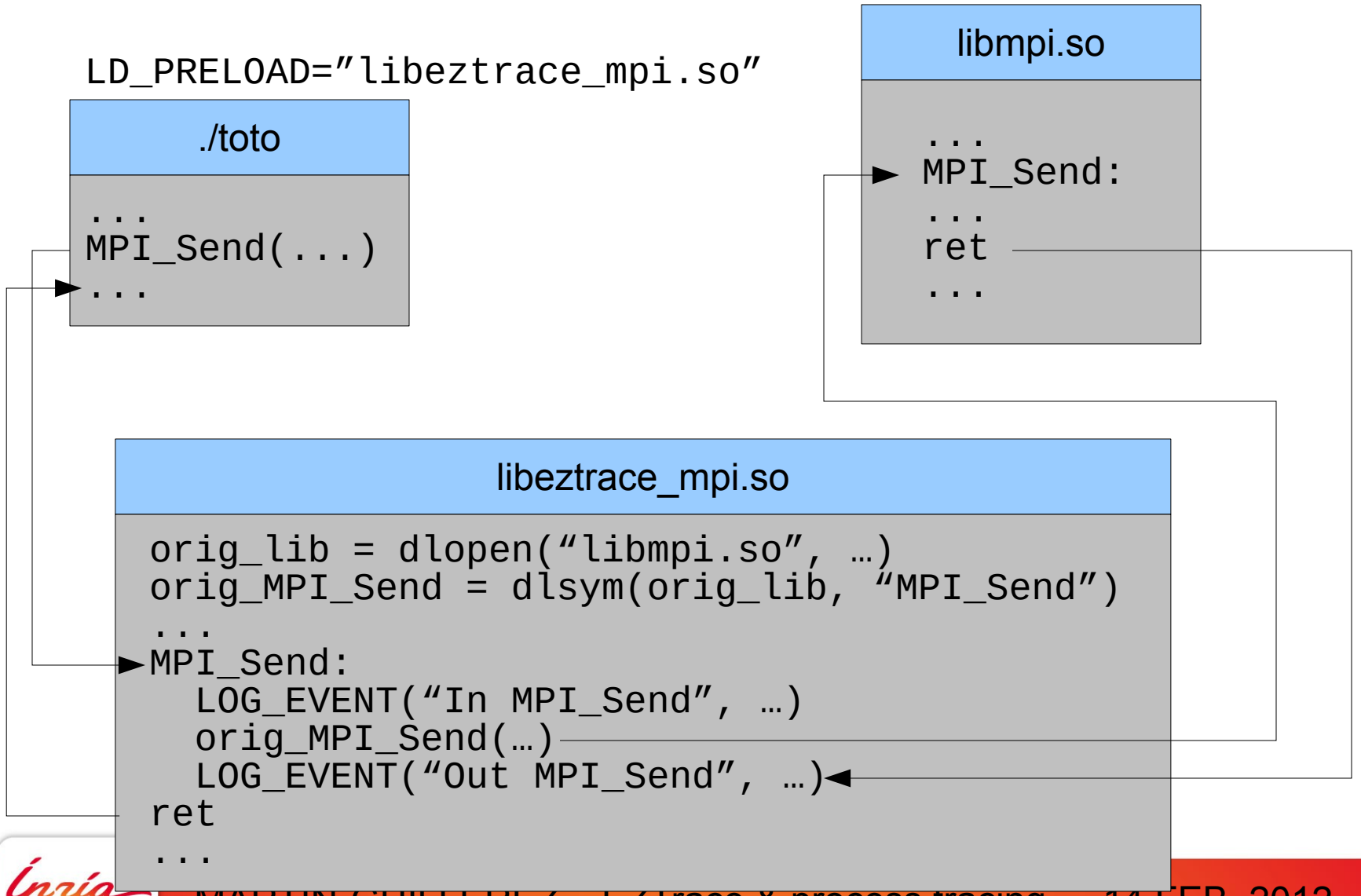
Dynamic library preloading
With an EZTrace module



Dynamic library preloading
With an EZTrace module



Dynamic library preloading
With an EZTrace module



EZTrace basics

EZTrace is a trace generator:

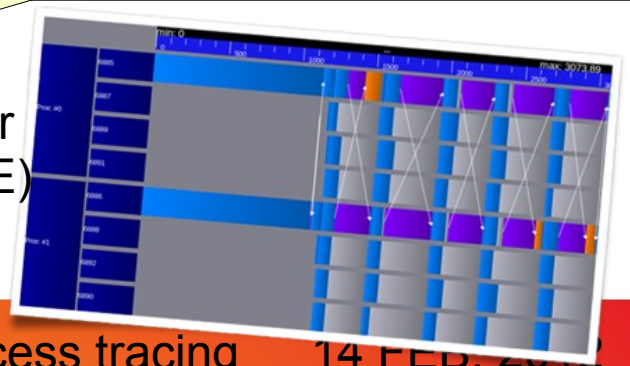
- Look out for some events during execution and rec… e
- Convert them to a standard format
- View the events

```
$ export EZTRACE_TRA…
$ mpirun -np 1 …                                    eadapp
```

ONLY ON DYNAMIC LIBRARIES!

```
$ eztrace_…                                    …ame>_eztrace_log*
```

Paje File… → Visualizer (ex.: ViTE)



It is the story of a fool that says to another fool who says to another fool that...

2

The ptrace() system call

ptrace() stands for process tracing

ptrace() is:

- A POSIX system call
- Extended in its Linux version
- A process tracing mechanism
  - Attach and detach a target process
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers

```
#include <sys/ptrace.h>

int ptrace(int requête, int pid, void* addr, void* data);
```

man 2 ptrace

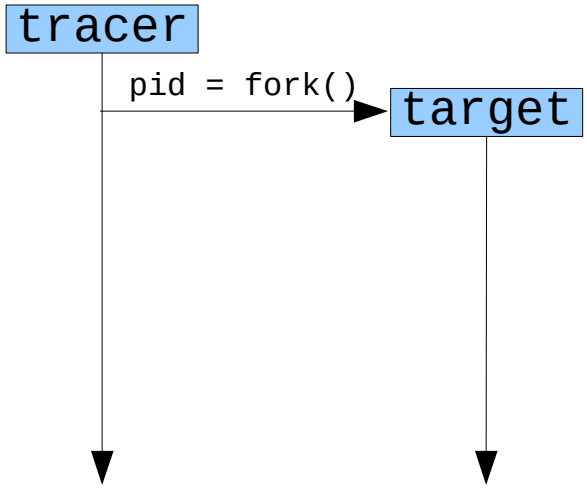
ptrace() process attachment

- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers

- Two methods:

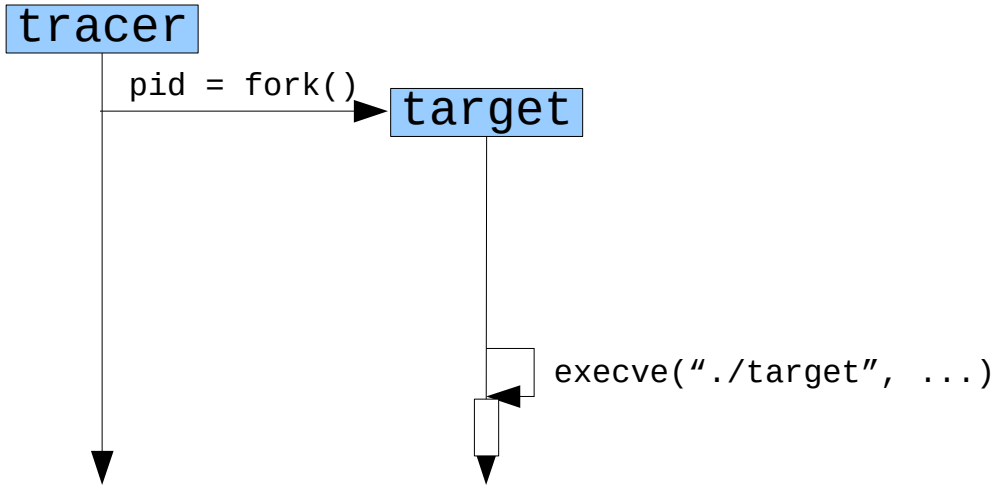
ptrace() process attachment

- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers
- Two methods:
  - `fork()` and `PTRACE_TRACEME`



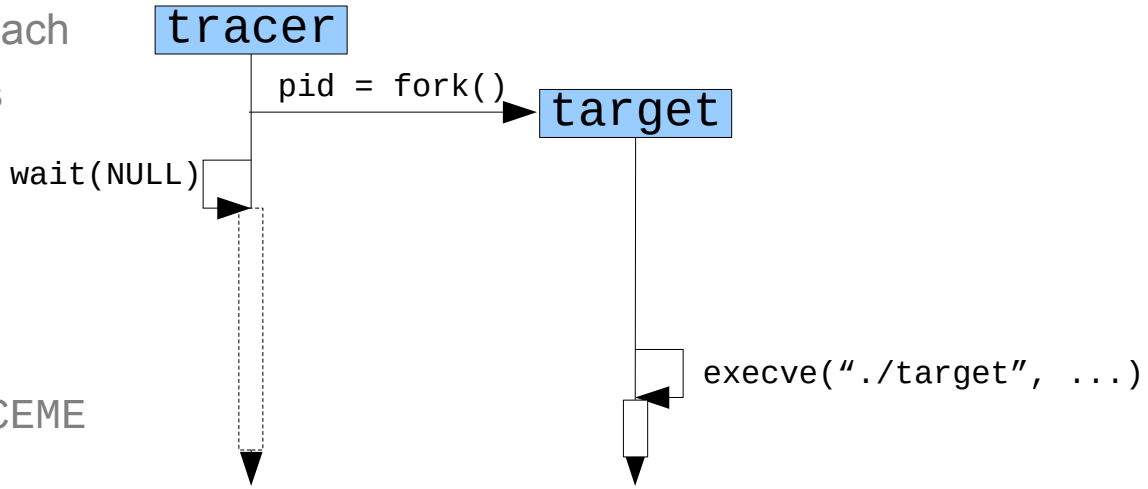
ptrace() process attachment

- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers
- Two methods:
  - `fork()` and `PTRACE_TRACEME`



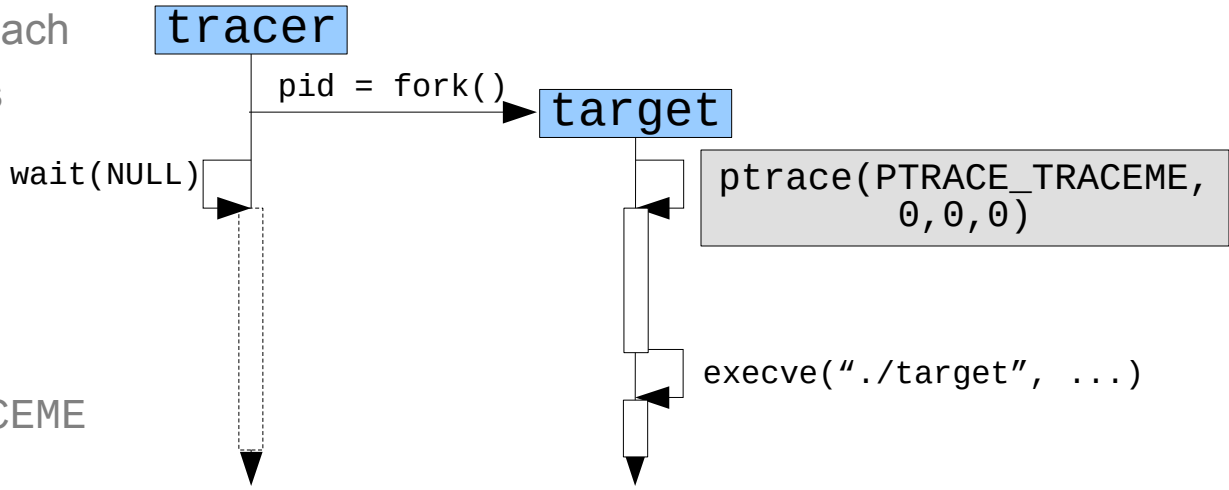
ptrace() process attachment

- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers
- Two methods:
  - `fork()` and `PTRACE_TRACEME`



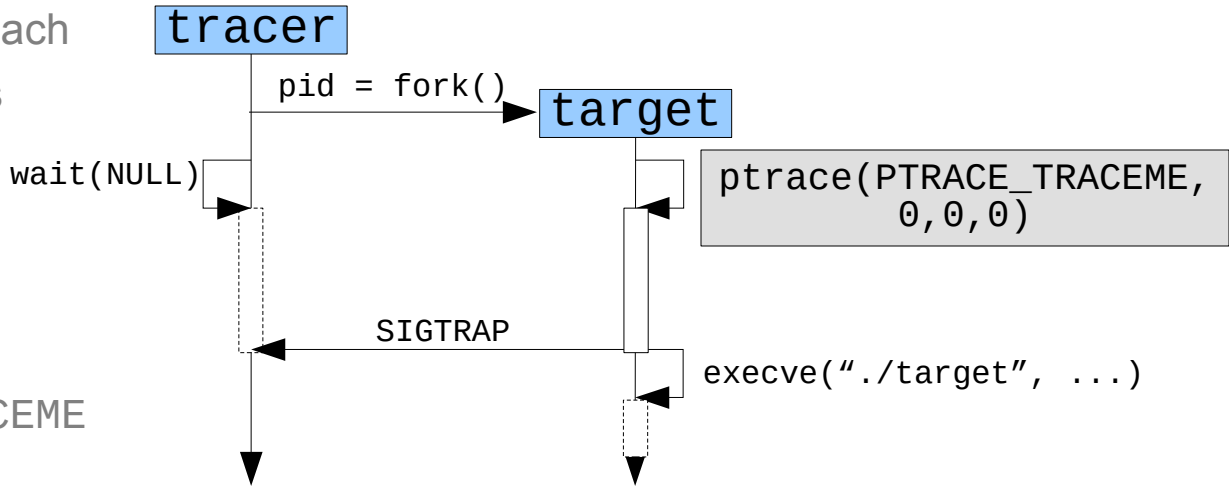
ptrace() process attachment

- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers
- Two methods:
  - `fork()` and `PTRACE_TRACEME`



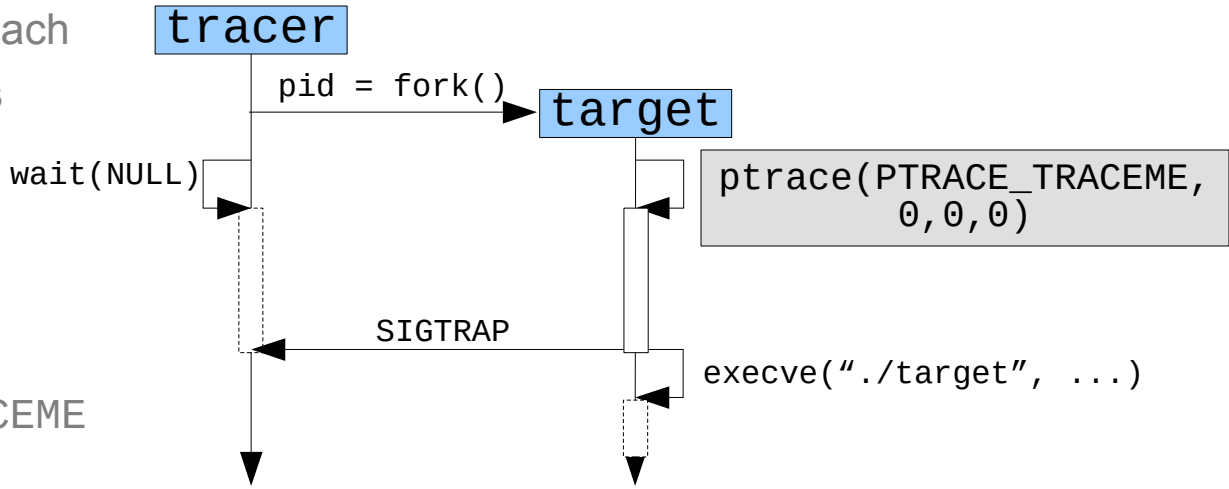
ptrace() process attachment

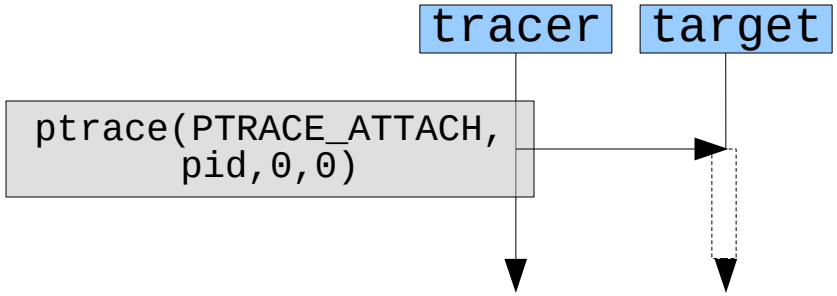
- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers
- Two methods:
  - `fork()` and `PTRACE_TRACEME`



ptrace() process attachment

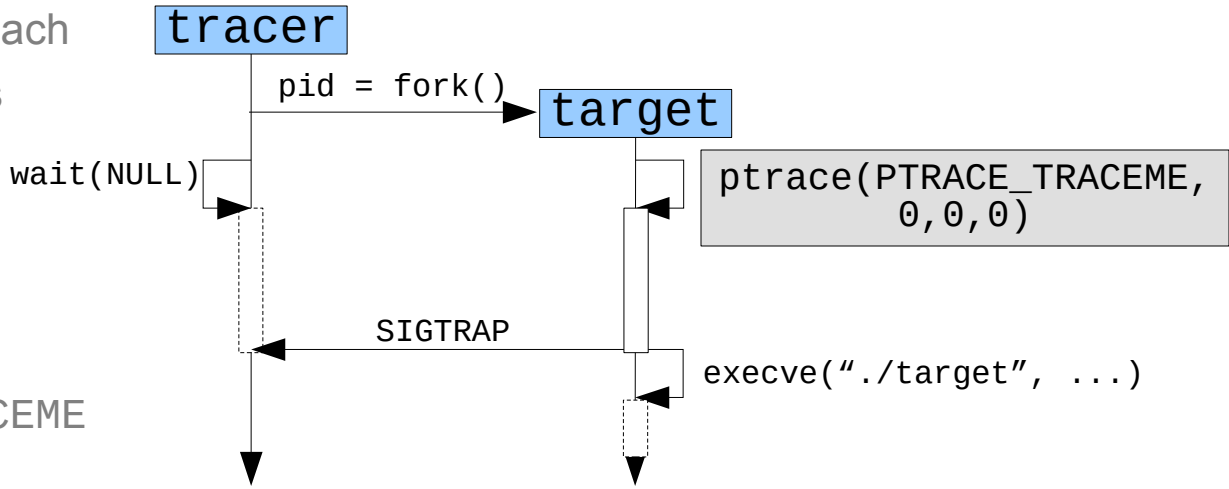
- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers
- Two methods:
  - `fork()` and `PTRACE_TRACEME`
  - `PTRACE_ATTACH`

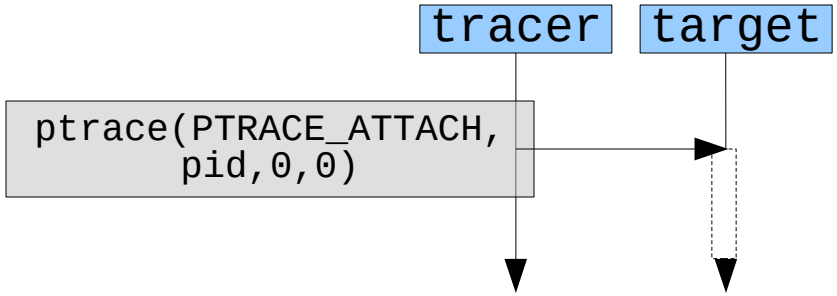




ptrace() process attachment

- An attached process can be traced
  - Stop at each system call / each instruction of target process
  - Examine and modify target memory and registers
- Two methods:
  - `fork()` and `PTRACE_TRACEME`
  - `PTRACE_ATTACH`
- Detaching using `PTRACE_DETACH`





ptrace() stepping

- Continue until next signal

```
ptrace(PTRACE_CONT, pid, NULL, NULL);
ptrace(PTRACE_CONT, pid, NULL, (void*)_signal);
```

ptrace() stepping

- Continue until next signal

```
ptrace(PTRACE_CONT, pid, NULL, NULL);
ptrace(PTRACE_CONT, pid, NULL, (void*)_signal);
```

- Continue until next system call
  - Stops at system call entry and exit

```
ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
ptrace(PTRACE_SYSCALL, pid, NULL, (void*)_signal);
```

ptrace() stepping

- Continue until next signal

```
ptrace(PTRACE_CONT, pid, NULL, NULL);
ptrace(PTRACE_CONT, pid, NULL, (void*)_signal);
```

- Continue until next system call
  - Stops at system call entry and exit

```
ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
ptrace(PTRACE_SYSCALL, pid, NULL, (void*)_signal);
```

- Stop after next instruction

```
ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
ptrace(PTRACE_SINGLESTEP, pid, NULL, (void*)_signal);
```

ptrace() memory reading and altering

- Reading / writing target memory

```
type_of_word data;
data = ptrace(PTRACE_PEEKTEXT, pid, (void*)addr, NULL);
ptrace(PTRACE_POKETEXT, pid, (void*)addr, (void*)data);
```

ptrace() memory reading and altering

- Reading / writing target memory

```
type_of_word data;
data = ptrace(PTRACE_PEEKTEXT, pid, (void*)addr, NULL);
ptrace(PTRACE_POKETEXT, pid, (void*)addr, (void*)data);
```

- Reading / writing target registers

```
#include <sys/user.h>
struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, pid, NULL, (void*)(&regs));
ptrace(PTRACE_SETREGS, pid, NULL, (void*)(&regs));
```

ptrace() memory reading and altering

- Reading / writing target memory

```
type_of_word data;
data = ptrace(PTRACE_PEEKTEXT, pid, (void*)addr, NULL);
ptrace(PTRACE_POKETEXT, pid, (void*)addr, (void*)data);
```

- Reading / writing target registers

```
#include <sys/user.h>
struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, pid, NULL, (void*)(&regs));
ptrace(PTRACE_SETREGS, pid, NULL, (void*)(&regs));
```

- Reading / writing target user data

```
#include <sys/user.h>
struct user u;
ptrace(PTRACE_PEEKUSER, pid, NULL, (void*)(&u));
ptrace(PTRACE_POKEUSER, pid, NULL, (void*)(&u));
```

ptrace() memory reading and altering

- Reading / writing target memory

```
type_of_word data;
data = ptrace(PTRACE_PEEKTEXT, pid, (void*)addr, NULL);
ptrace(PTRACE_POKETEXT, pid, (void*)addr, (void*)data);
```

- Reading / writing target registers

```
#include <sys/user.h>
struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, pid, NULL, (void*)(&regs));
ptrace(PTRACE_SETREGS, pid, NULL, (void*)(&regs));
```

- Reading / writing target user data

```
#include <sys/user.h>
struct user u;
ptrace(PTRACE_PEEKUSER, pid, NULL, (void*)(&u));
ptrace(PTRACE_POKEUSER, pid, NULL, (void*)(&u));
```

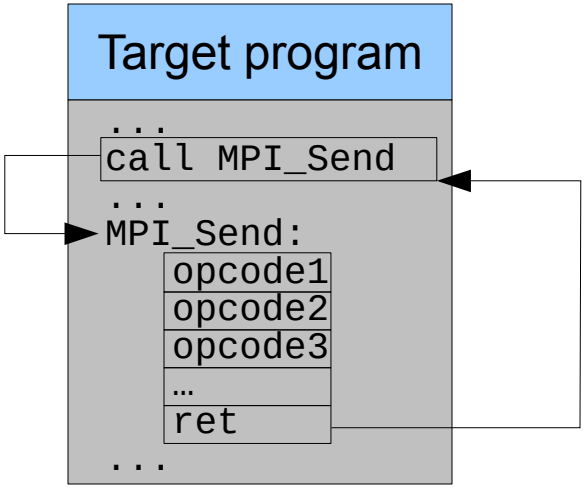
- Reading target signal informations

```
siginfo_t siginfo;
data = ptrace(PTRACE_GETSIGINFO, pid, NULL, (void*)(&siginfo));
```

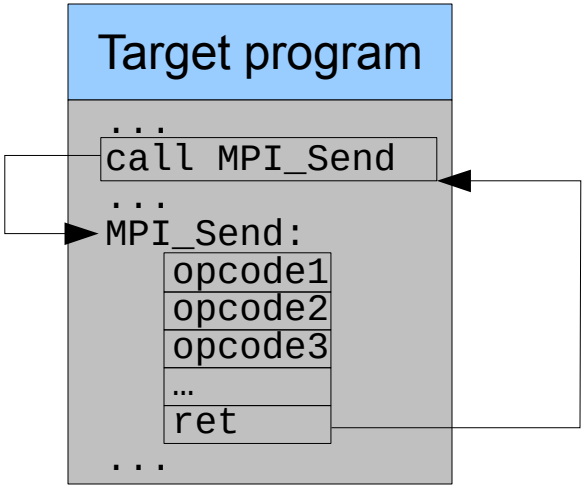
3

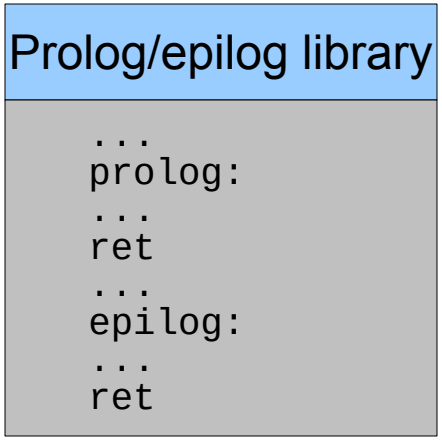
Instrumentation of static functions in EZTrace

How to instrument static functions ?

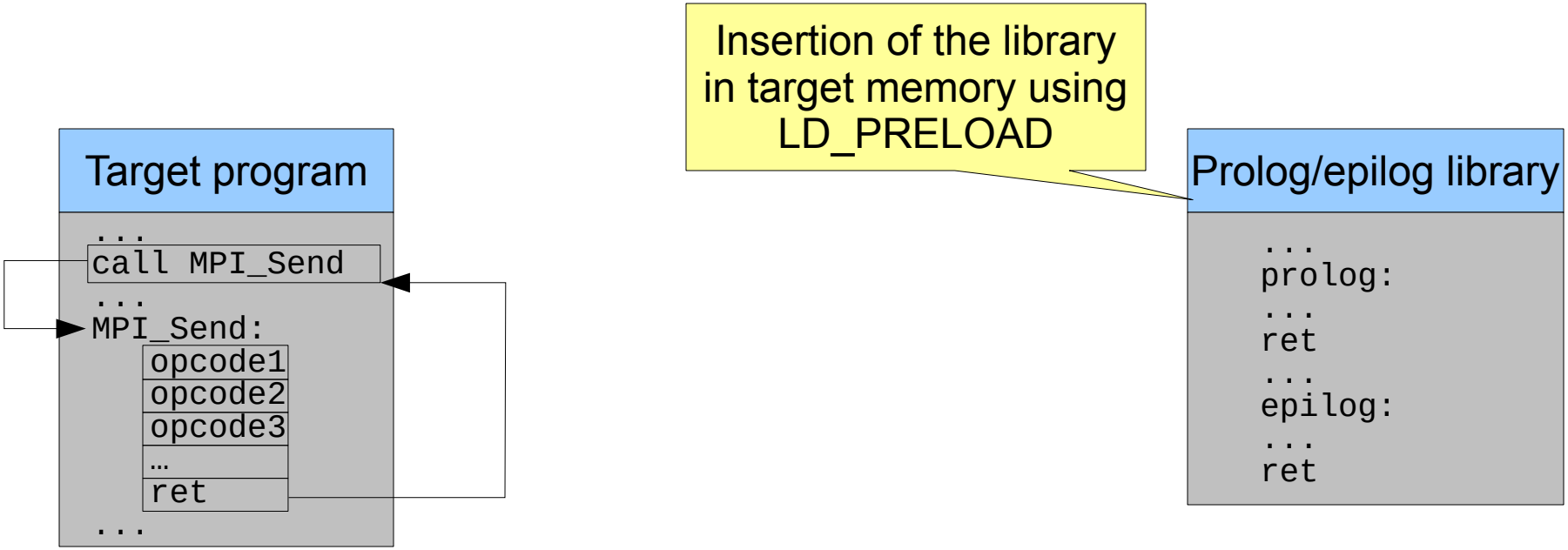


How to instrument static functions ?

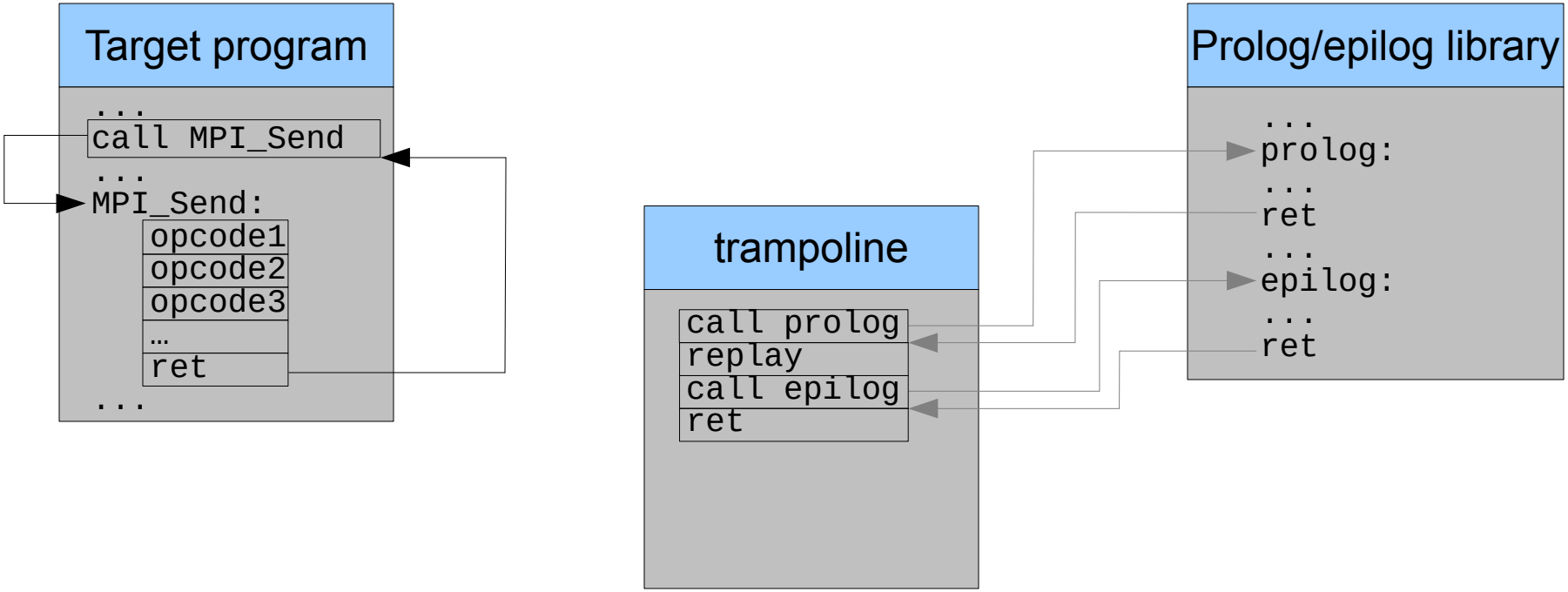




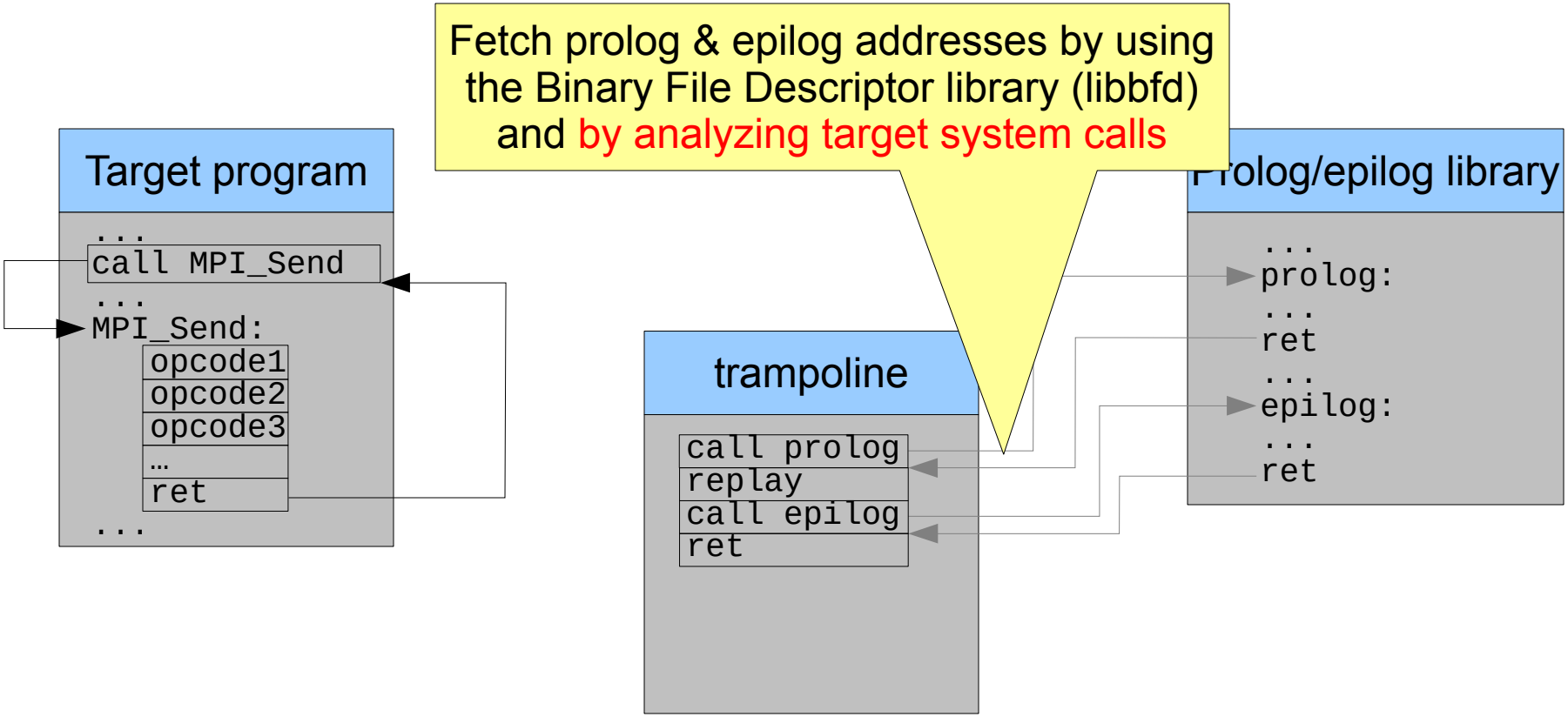
How to instrument static functions ?



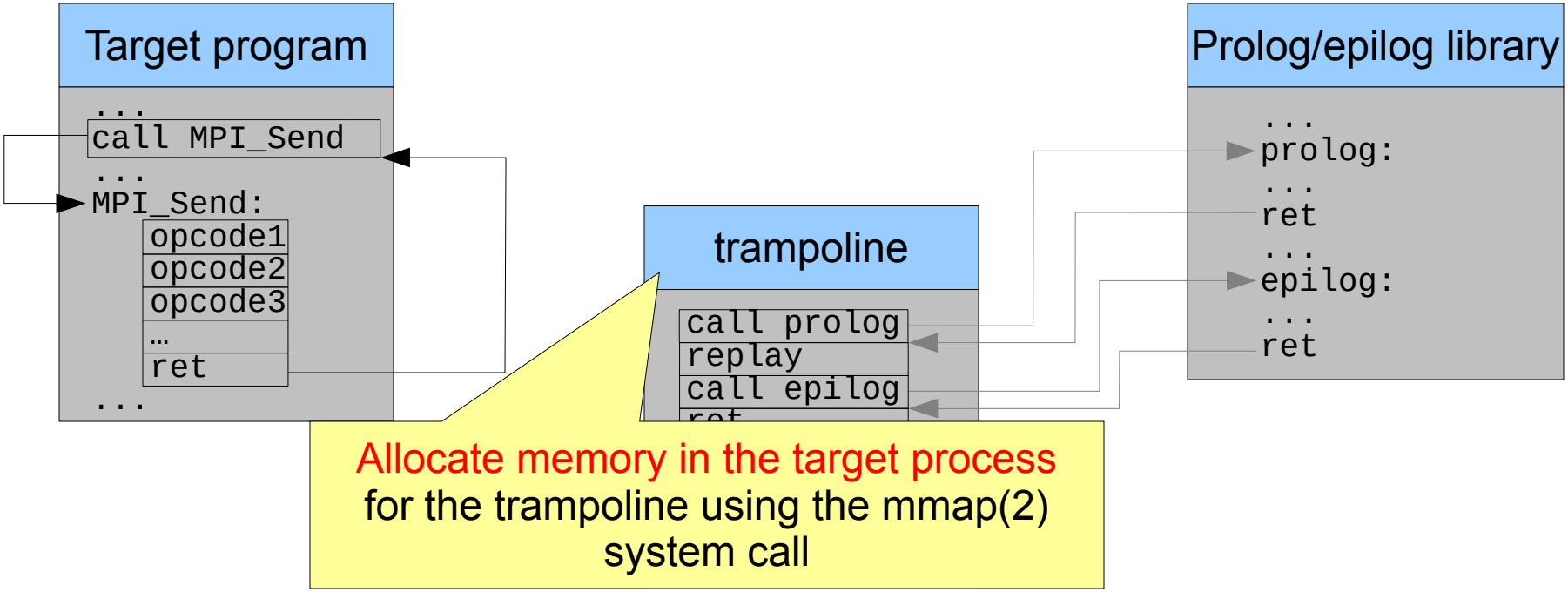
How to instrument static functions ?



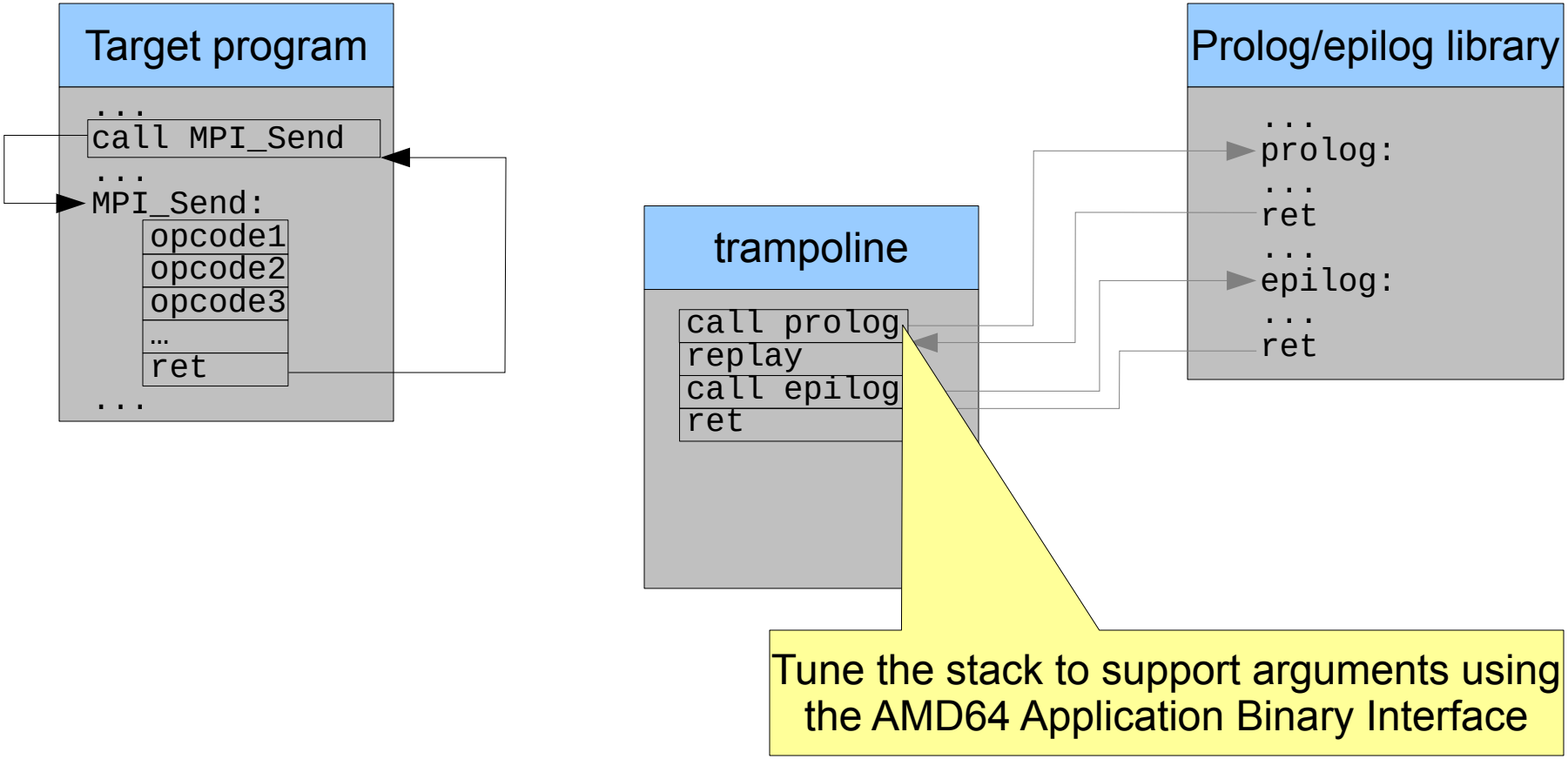
How to instrument static functions ?



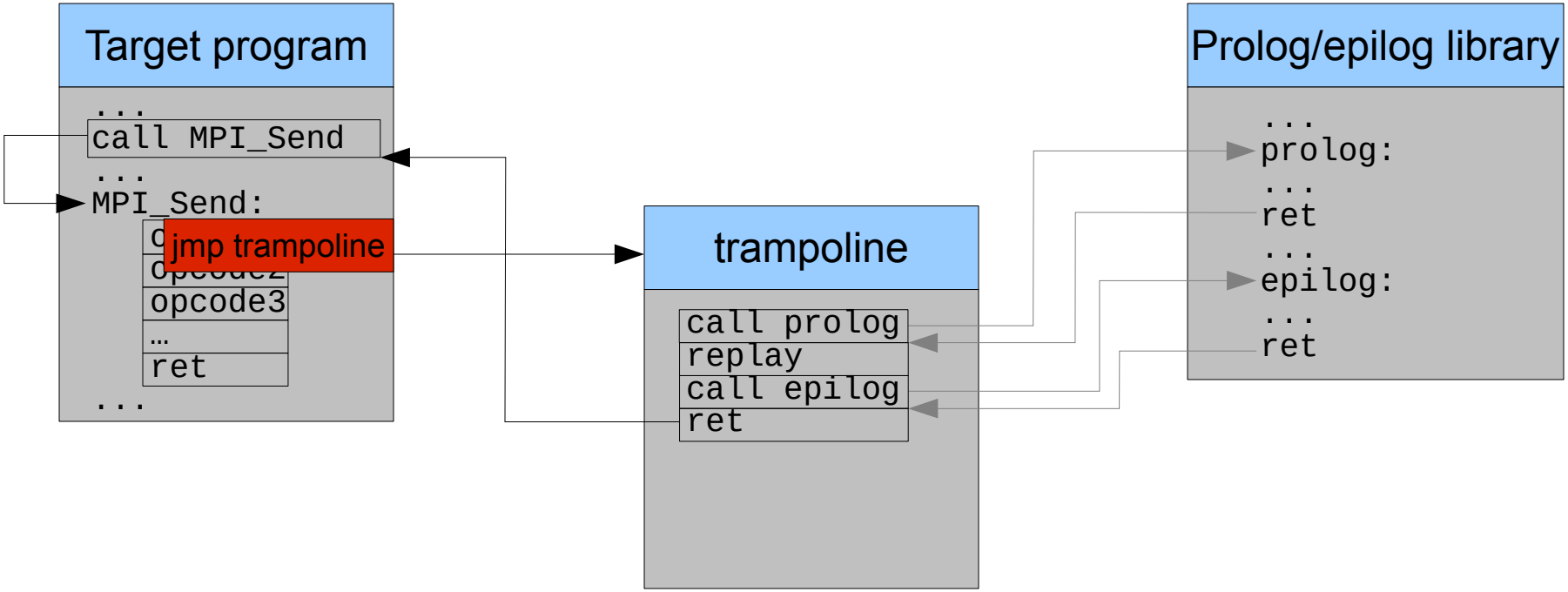
How to instrument static functions ?



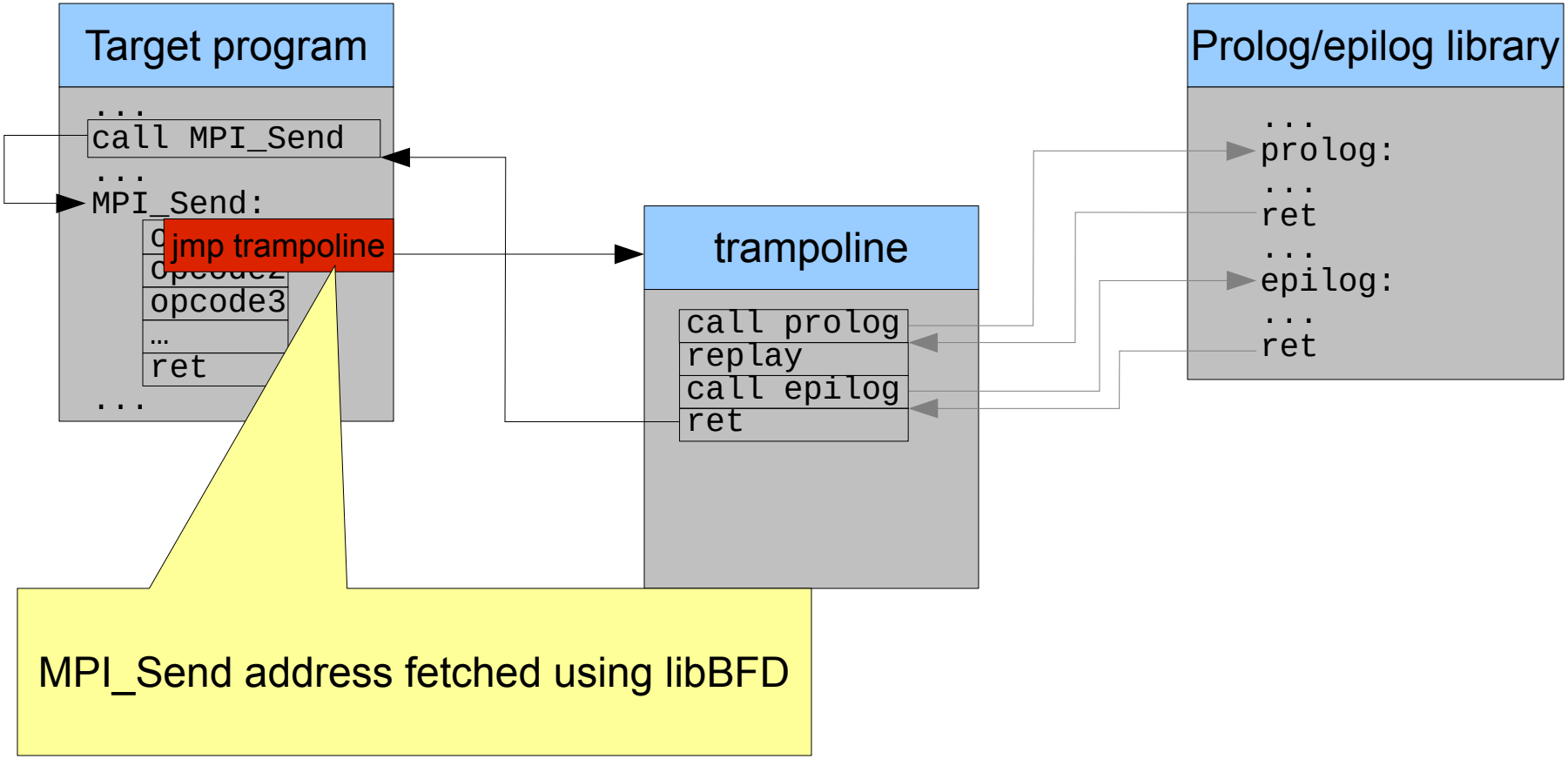
How to instrument static functions ?



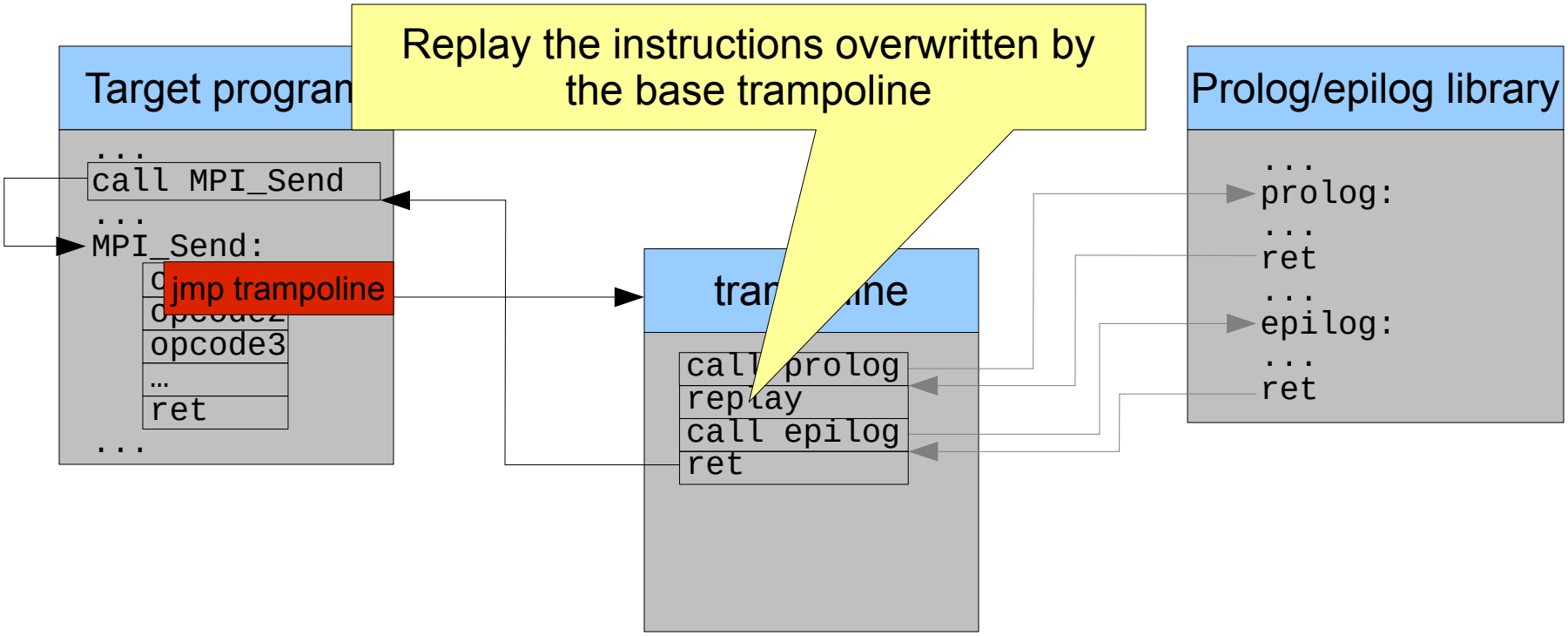
How to instrument static functions ?



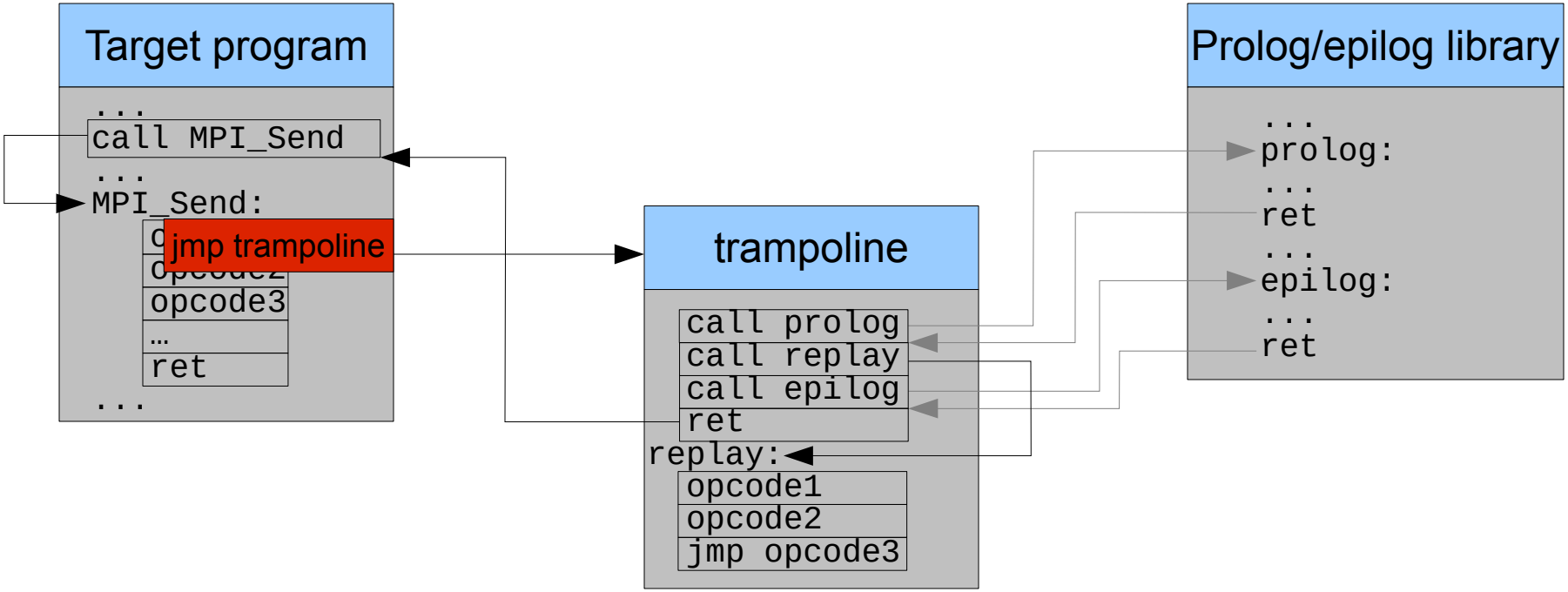
How to instrument static functions ?



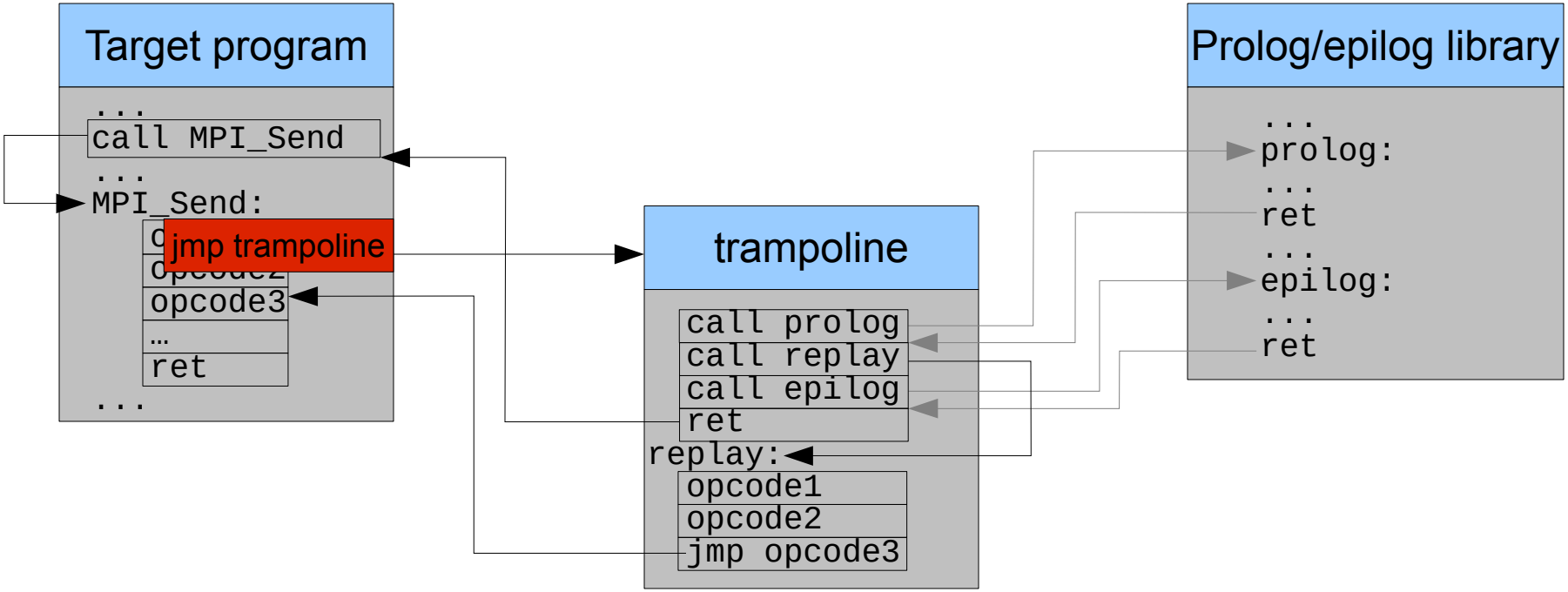
How to instrument static functions ?



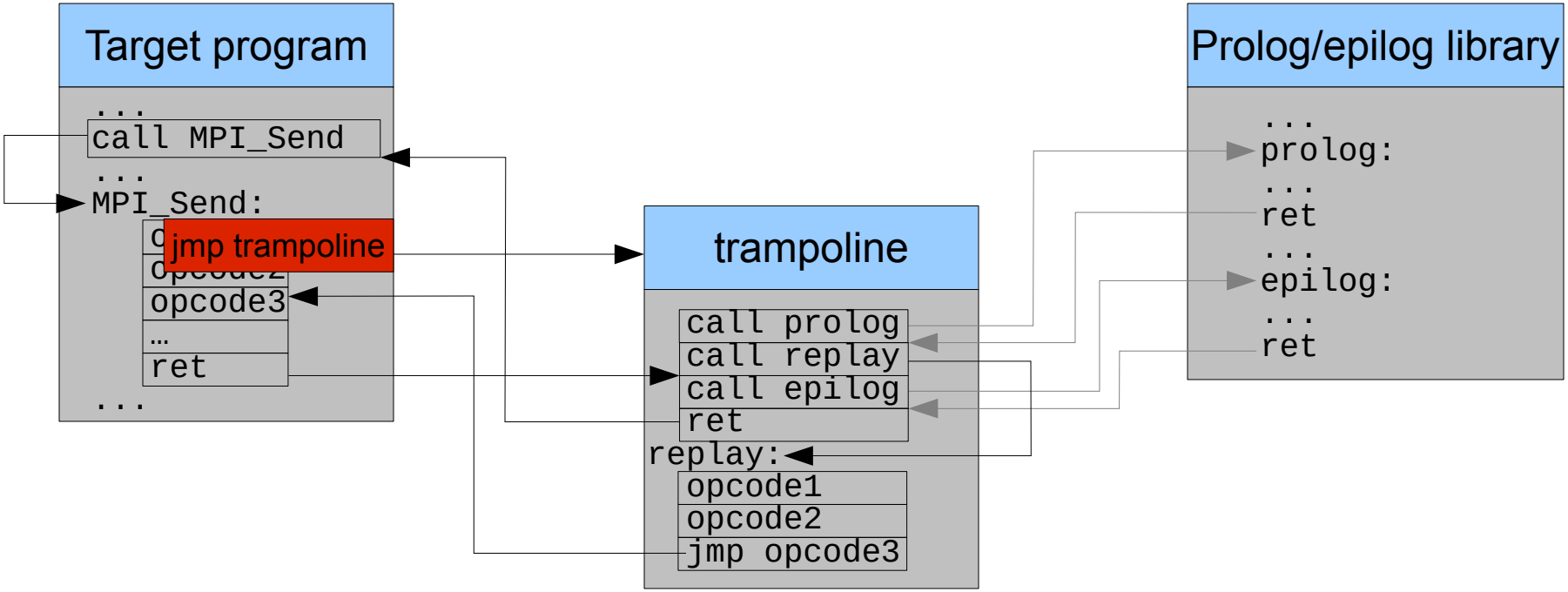
How to instrument static functions ?



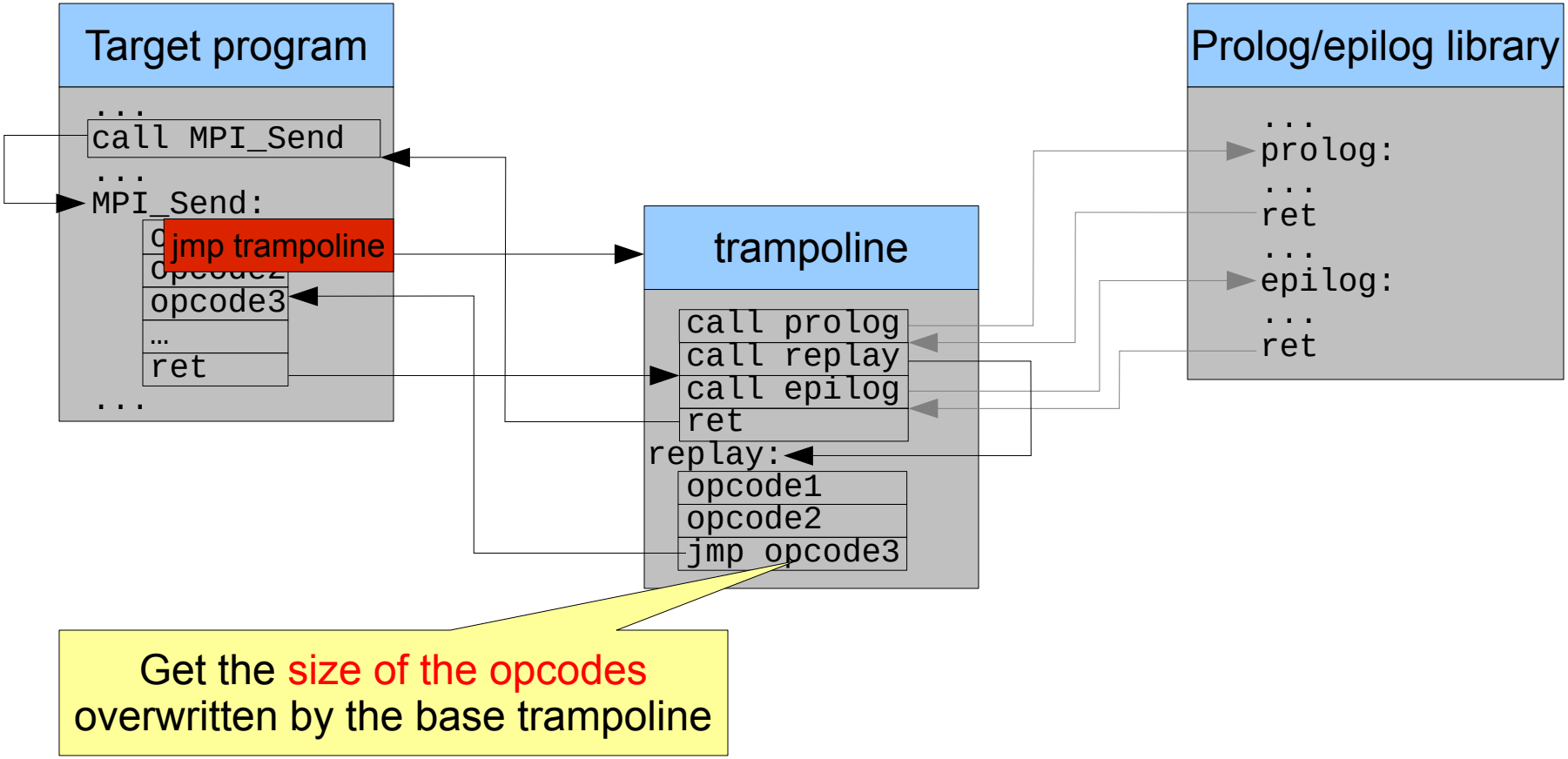
How to instrument static functions ?



How to instrument static functions ?



How to instrument static functions ?



4

Technical details for the geek

Technical challenges left

- Allocate memory in the target process
- Get the address of the preloaded library
  - The offset of the prolog and epilog inside the library are fetched with libbfd
  - The library loading mechanisms is spyed to find the mmaping of the library
- Get the size of the overwritten opcodes

The mmap() system call

The mmap() system call allocate a memory page

```
#include <sys/mman.h>

void* mmap(void *addr, size_t len, int prot,
                       int flags, int fd, off_t offset)
```

man 2 mmap

The `mmap()` system call

The `mmap()` system call allocate a memory page

```
#include <sys/mman.h>

void* mmap(void *addr, size_t len, int prot,
                       int flags, int fd, off_t offset)
```

man 2 mmap

- Can be used to map a library in memory

The mmap() system call

The mmap() system call allocate a memory page

```
#include <sys/mman.h>

void* mmap(void *addr, size_t len, int prot,
                       int flags, int fd, off_t offset)
```

man 2 mmap

- Can be used to map a library in memory
- Can also be used to allocate an executable segment for the trampoline

Calling a system call from the parent process

To call `mmap()` to allocate memory for the trampoline, we need to:

- Stop the target process
- Replace the current instruction by an `int 0x80 (syscall)`
- Set the accumulator (`rax`) to the system call number
- Fill the other registers according to the ABI
- Executes two `ptrace(PTRACE_SYSCALL, pid, NULL, NULL)`
- Get the result from the registers
- Restores the registers and the current instruction

Fetching the library base address

The base address of a preloaded library is determined at runtime.

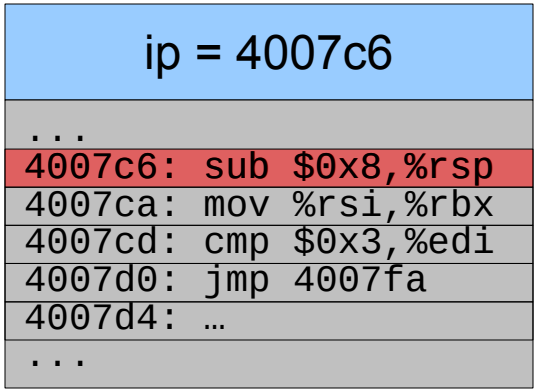
- The dlopen() mechanism do it by:
  - Opening the library file using the open() system call
  - Mapping it in memory using the mmap() system call

- Thus to fetch the base address:
  - Wait for `open(«library.so», *) = fd`
  - Wait for `mmap(*, *, *, *, fd, *) = baseAddress`

- A wait is performed using the `PTRACE_SYSCALL` method and by looking into registers

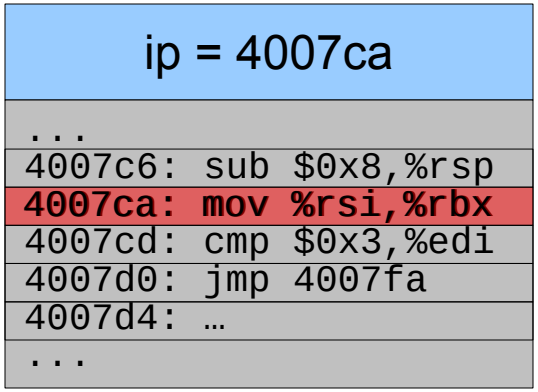
Getting the overwritten opcode size

Obtained by single stepping...



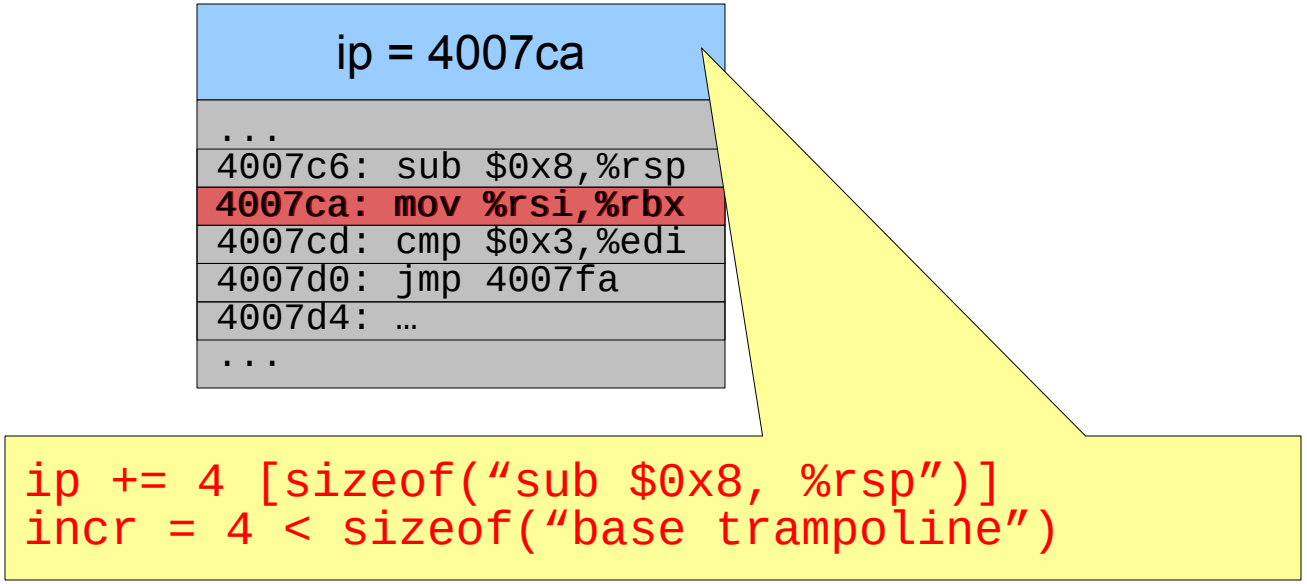
Getting the overwritten opcode size

Obtained by single stepping...



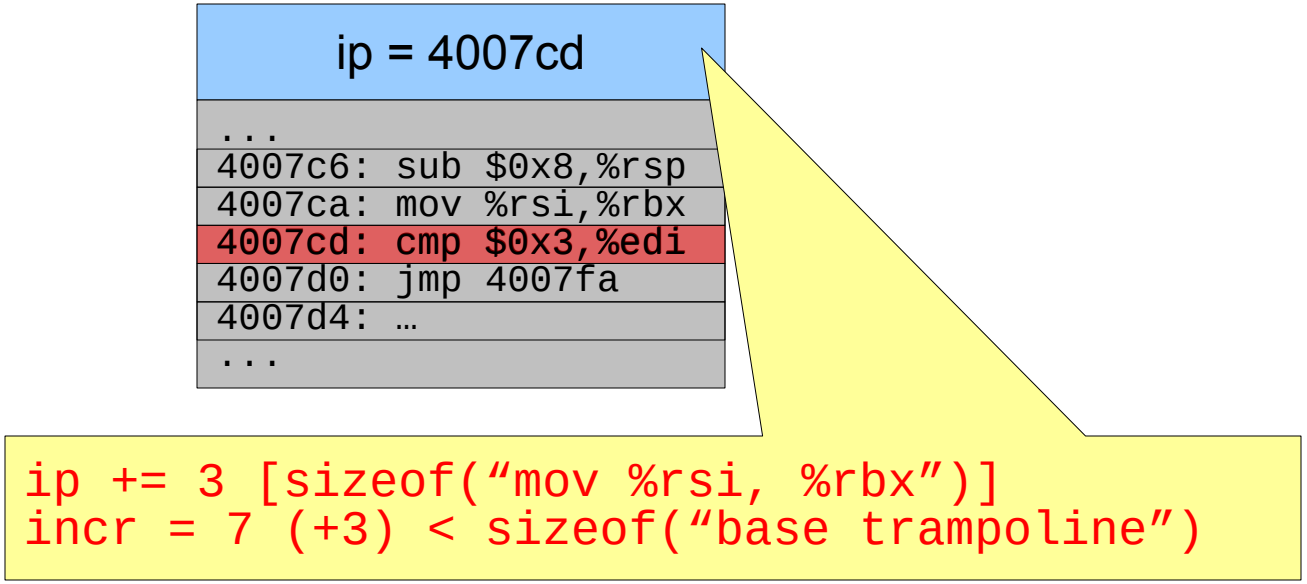
Getting the overwritten opcode size

Obtained by single stepping...



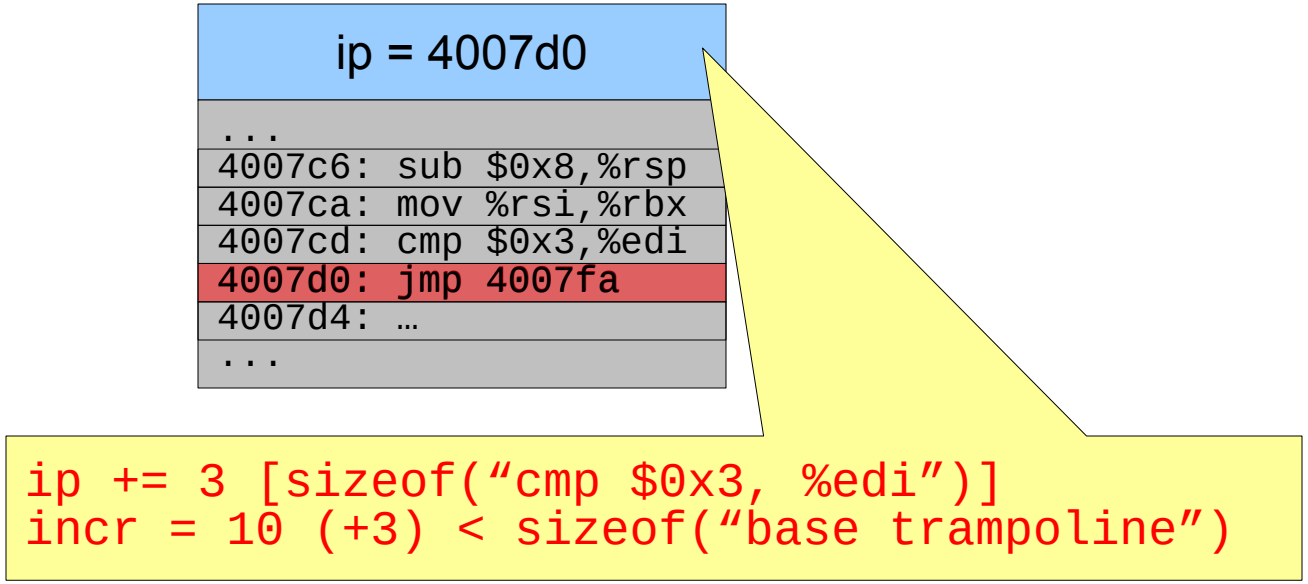
Getting the overwritten opcode size

Obtained by single stepping...



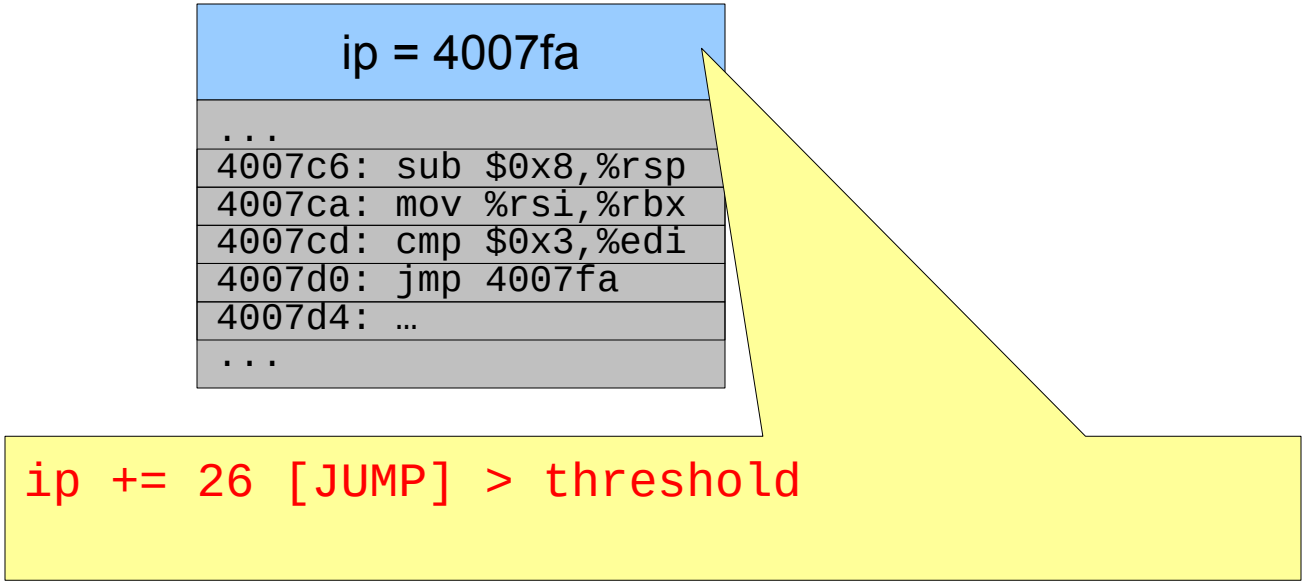
Getting the overwritten opcode size

Obtained by single stepping...



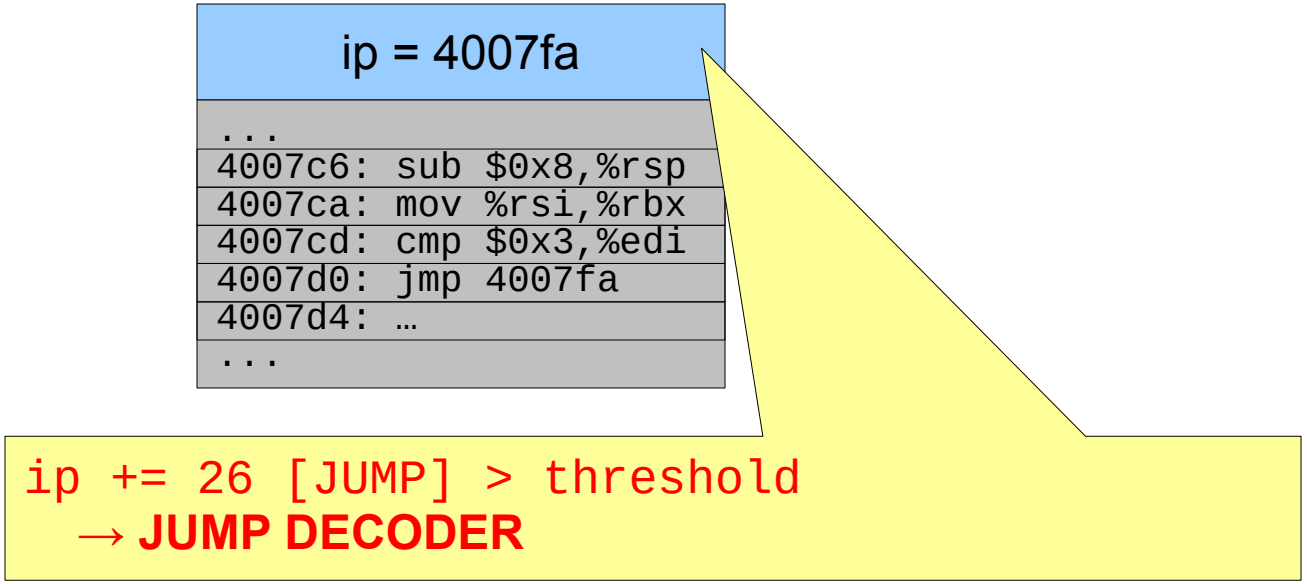
Getting the overwritten opcode size

Obtained by single stepping...



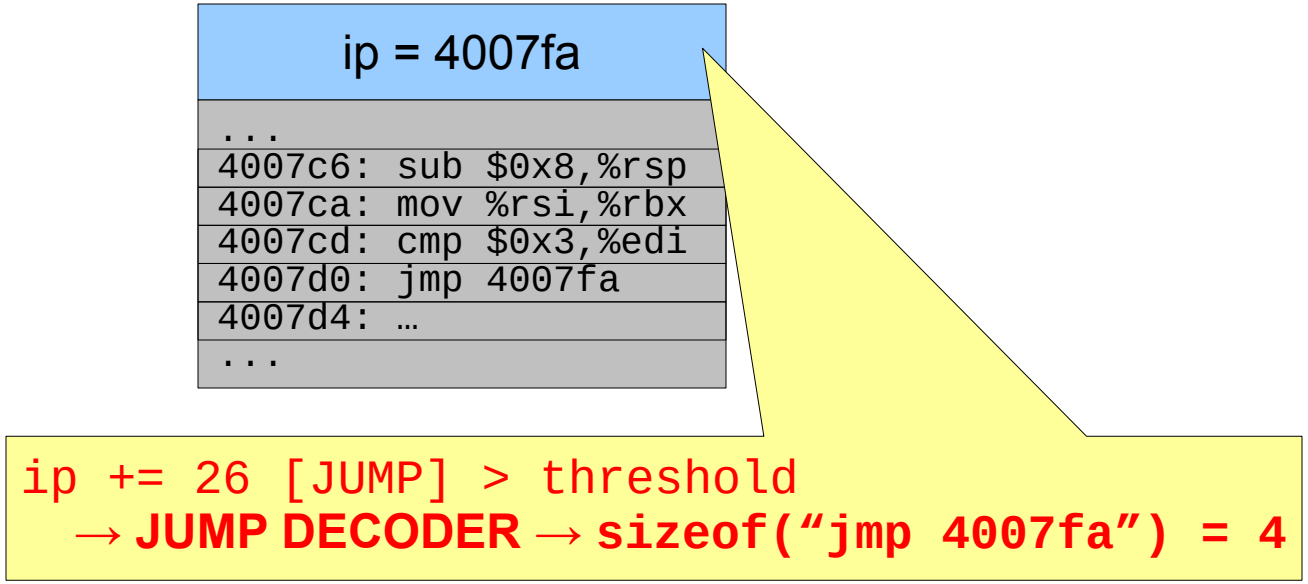
Getting the overwritten opcode size

Obtained by single stepping...



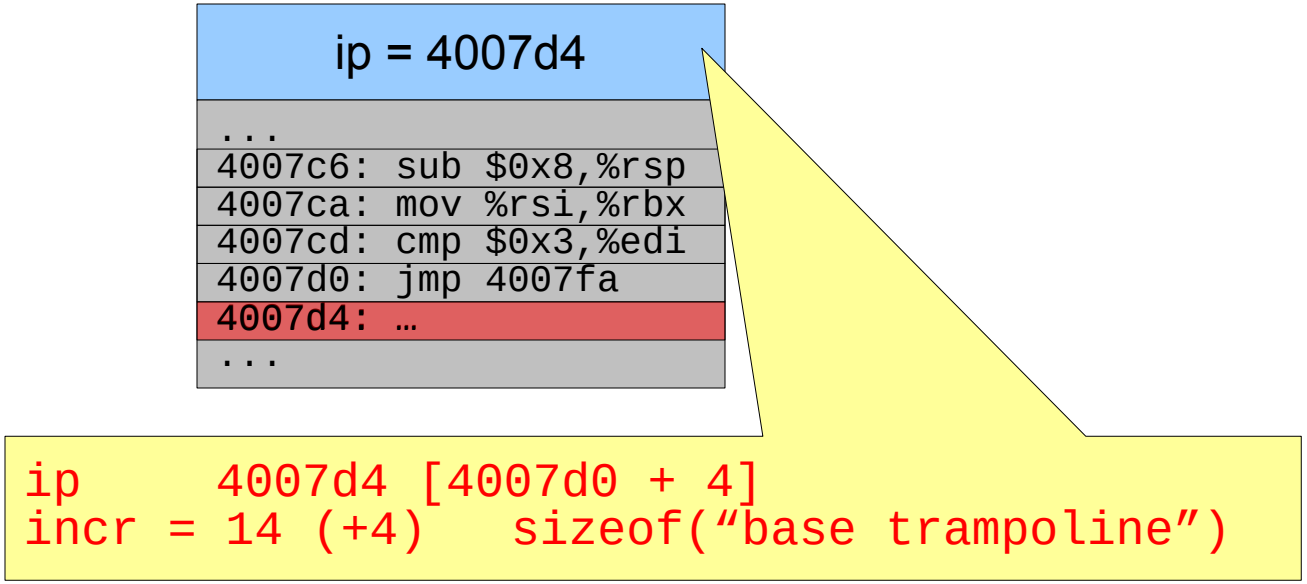
Getting the overwritten opcode size

Obtained by single stepping...



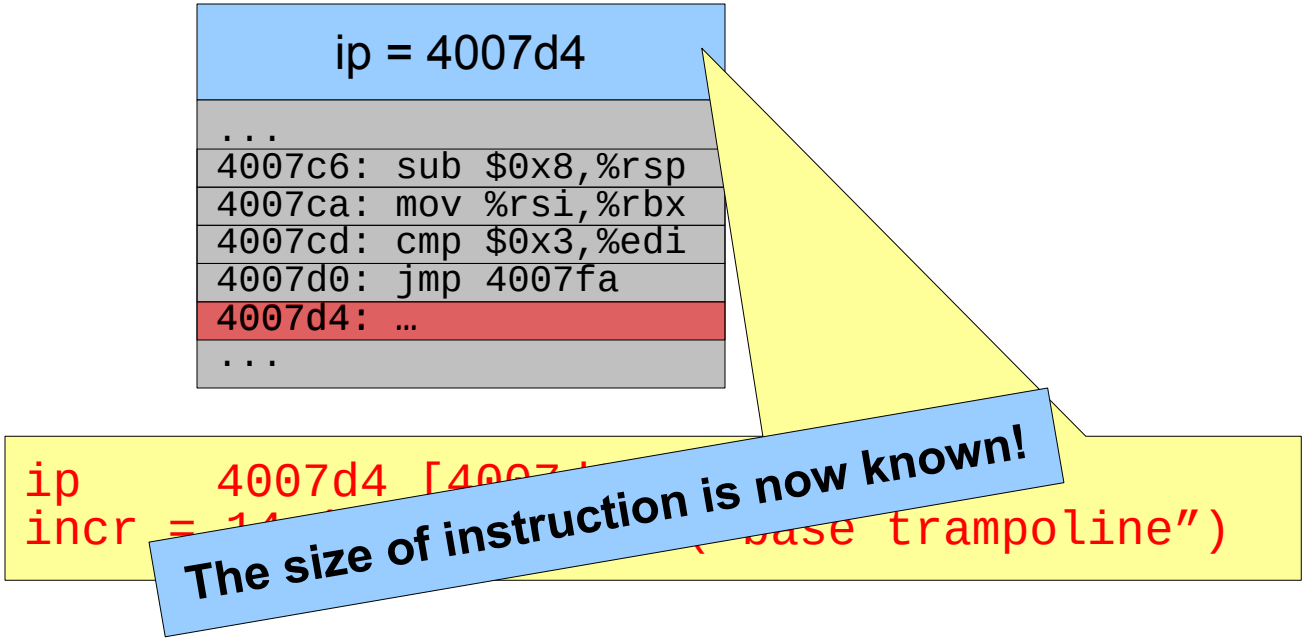
Getting the overwritten opcode size

Obtained by single stepping...



Getting the overwritten opcode size

Obtained by single stepping...



5
Other architectures

Other architectures

Other CPUs...

- 32-bit Intel Architecture is straighforward
  - Just a simple modification of the ABI
- RISC processors are easier
  - Simplified ABI
  - Fixed-size instructions

And other operating system...

- *BSD should not be a problem
  - The `ptrace()` system call changes but offers similar functionalities
  - The ABI compatibility needs to be verified
- MacOS X is problematic

The MacOS X issue

MacOS X `ptrace()` system call is outdated...

- The `task_for_pid()` mechanism replaces it
- The ABI changes
- MacOS X has a complex capability scheme needed for tracing
- Even the LD_PRELOAD mechanism changes

6
Future

In the future...

EZTrace will:

In the future...

EZTrace will:

- instrument functions of static libraries,

In the future...

EZTrace will:

- instrument functions of static libraries,
- instrument user-defined functions,

In the future...

EZTrace will:

- instrument functions of static libraries,
- instrument user-defined functions,
- do it easily thanks to a module generator,

```
BEGIN_MODULE
NAME example_lib
DESC "module for the example library"
LANGUAGE C
ID 42

int example_function2(int* array, int array_size)
BEGIN
  RECORD_STATE("running example_function2")
END
END_MODULE
```

In the future...

EZTrace will:

- instrument functions of static libraries,
- instrument user-defined functions,
- do it easily thanks to a module generator,

```
BEGIN_MODULE
NAME example_lib
DESC "module for the example library"
LANGUAGE C
ID 42

int example_function2(int* array, int array_size)
BEGIN
  RECORD_STATE("running example_function2")
END
END_MODULE
```

- and save the world...

In the future...

EZTrace will:

- instrument functions of static libraries,
- instrument user-defined functions,
- do it easily thanks to a module generator,

```
BEGIN_MODULE
NAME example_lib
DESC "module for the example library"
LANGUAGE C
ID 42

int example_function2(int* array, int array_size)
BEGIN
  RECORD_STATE("running example_function2")
END
END_MODULE
```

- and save the world...

...and the ornithorhynchus

Thank you!

http://eztrace.gforge.inria.fr