Девятая независимая
научно-практическая конференция
«Разработка ПО 2013»

23 - 25 октября, Москва

2013 CEE-SEC(R)
Software Engineering Conference in Russia

Легковесное профилирование разделяемых библиотек в Linux

для встраиваемых систем

Кирилл Кринкин, Марк Заславский, Эдуард Рябиков

EMC²

OSLL

Motivation

Popular Linux Profilers (gprof, gcov, GPT, Valgrind) have following **problems**:

- Need to **recompile** with special options (gprof, gcov)
- Need to **relink** program with 3rd-party libraries (GPT)
- Need to use **special environment** for profiling (Valgrind)
- Need to use **superuser rights**

Project goals

We need a tool for system-wide ELF executables performance analysis.

This tool should **allow user** next things **do easily**:

- Profile function calls **without recompilation and relinking** with 3rd-party libraries
- Profile **only given set of C/C++ functions** from shared libraries in Linux
- Profile both **dynamically linked** and **dynamically loaded** functions
- Profile **without creating of special environment**
- Get information about **number** and **total duration** of function calls
- Perform profiling on **x86/x64** platforms

“Non-invasive” profiling

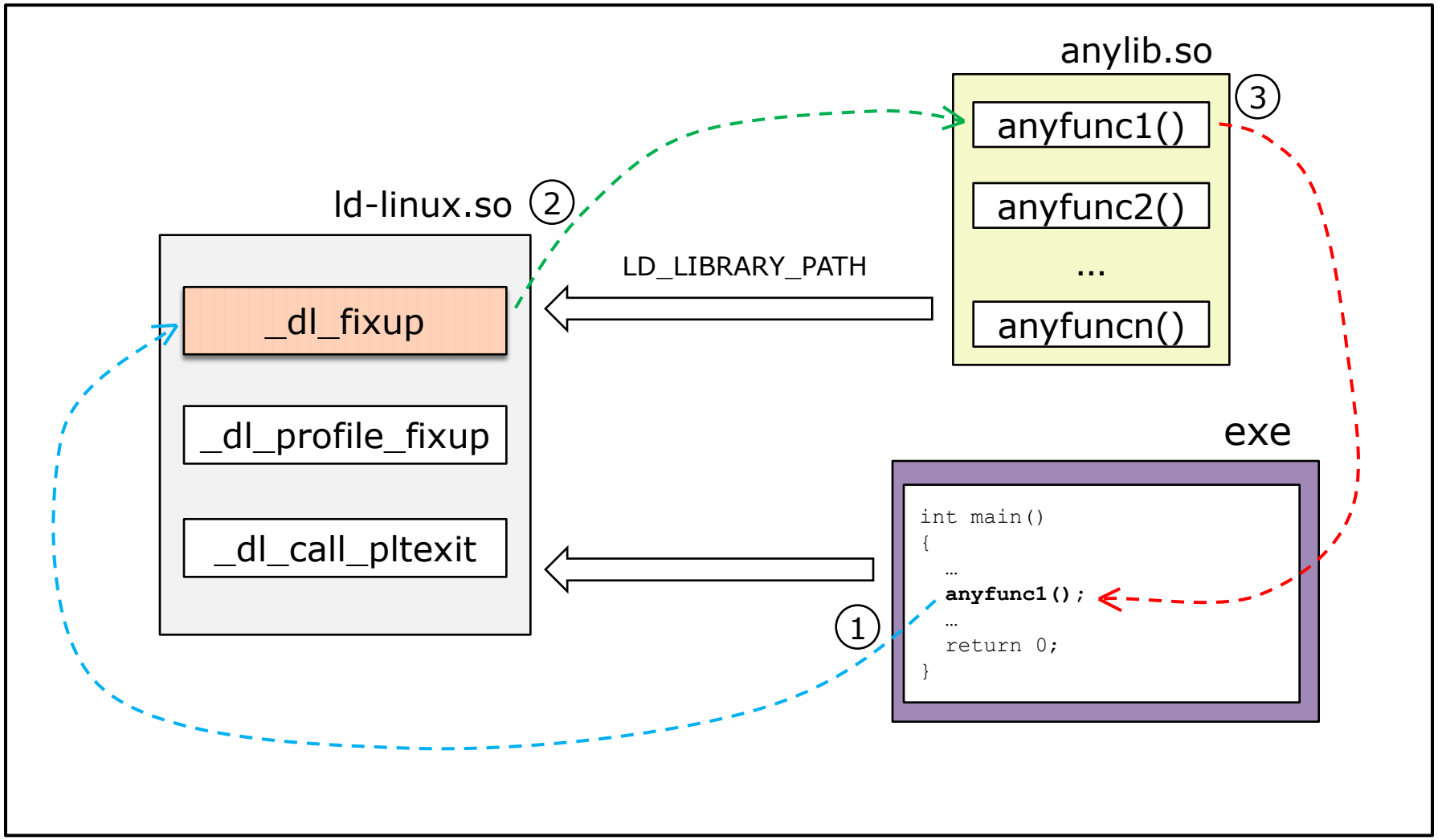
Main ideas:

- Profiler **can not be implemented** in the program code
- Profiling should be performed at **well-defined points of function calls**
- Profiling process **should not corrupt the algorithm** of profiled application
- Profiling process should use **minimum amount of system resources**
- Results of profiling should be **as accurate as possible**

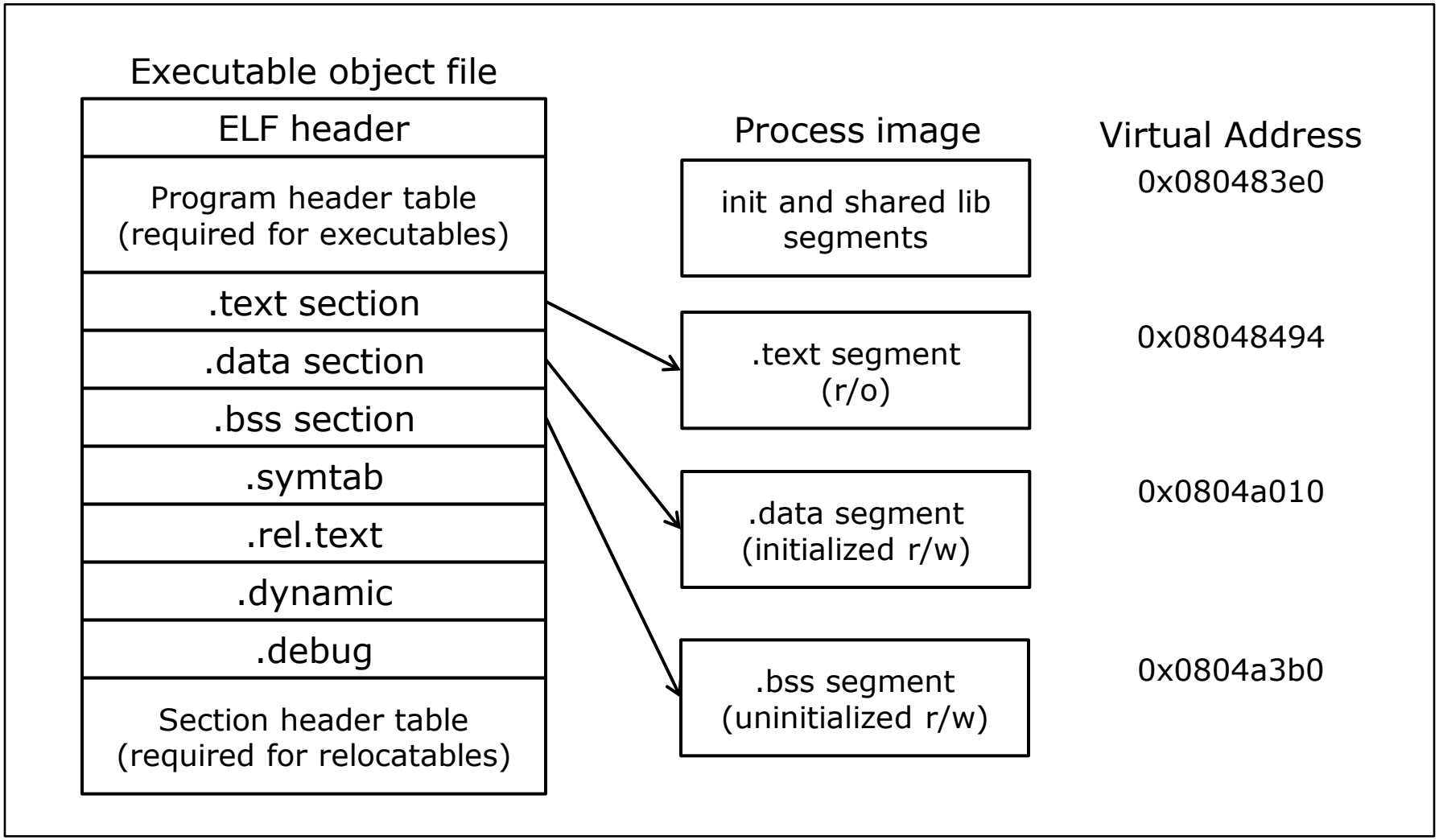
Ways to implement

- Infiltration into the **symbol relocation process**
- Modification of Linux **dynamic linker** (ld-linux.so)
- Modification of **dynamic loading library** (libdl.so)

Dynamic Linking



ELF Parsing by Dynamic Linker



1. Resolving the Dependencies

- When linking a dynamic executable, one or more shared objects are explicitly referenced. These objects are recorded as dependencies within the dynamic executable.

- The runtime linker uses this dependency information to locate, and load, the associated objects.

- Once all the dynamic executable's dependencies are loaded, each dependency is inspected, in the order the dependency is loaded, to locate any additional dependencies.

1. Resolving the Dependencies

- The Linux runtime linker looks in two default locations for dependencies /lib and /usr/lib.
- The dependencies of a dynamic executable or shared object can be displayed using **ldd**. For example, the file /usr/bin/cat has the following dependencies:
  $ ldd /usr/bin/cat
  libc.so.1 => /lib/libc.so.1
  libm.so.2 => /lib/libm.so.2
- The dependencies recorded in an object can be inspected using **dump**. Use this command to display the file's .dynamic section, and look for entries that have a NEEDED tag.
  $ dump -Lvp prog
  prog:
  [INDEX] Tag Value
  [1] NEEDED libfoo.so.1
  [2] NEEDED libc.so.1
  [3] RUNPATH /home/me/lib:/home/you/lib
  .....

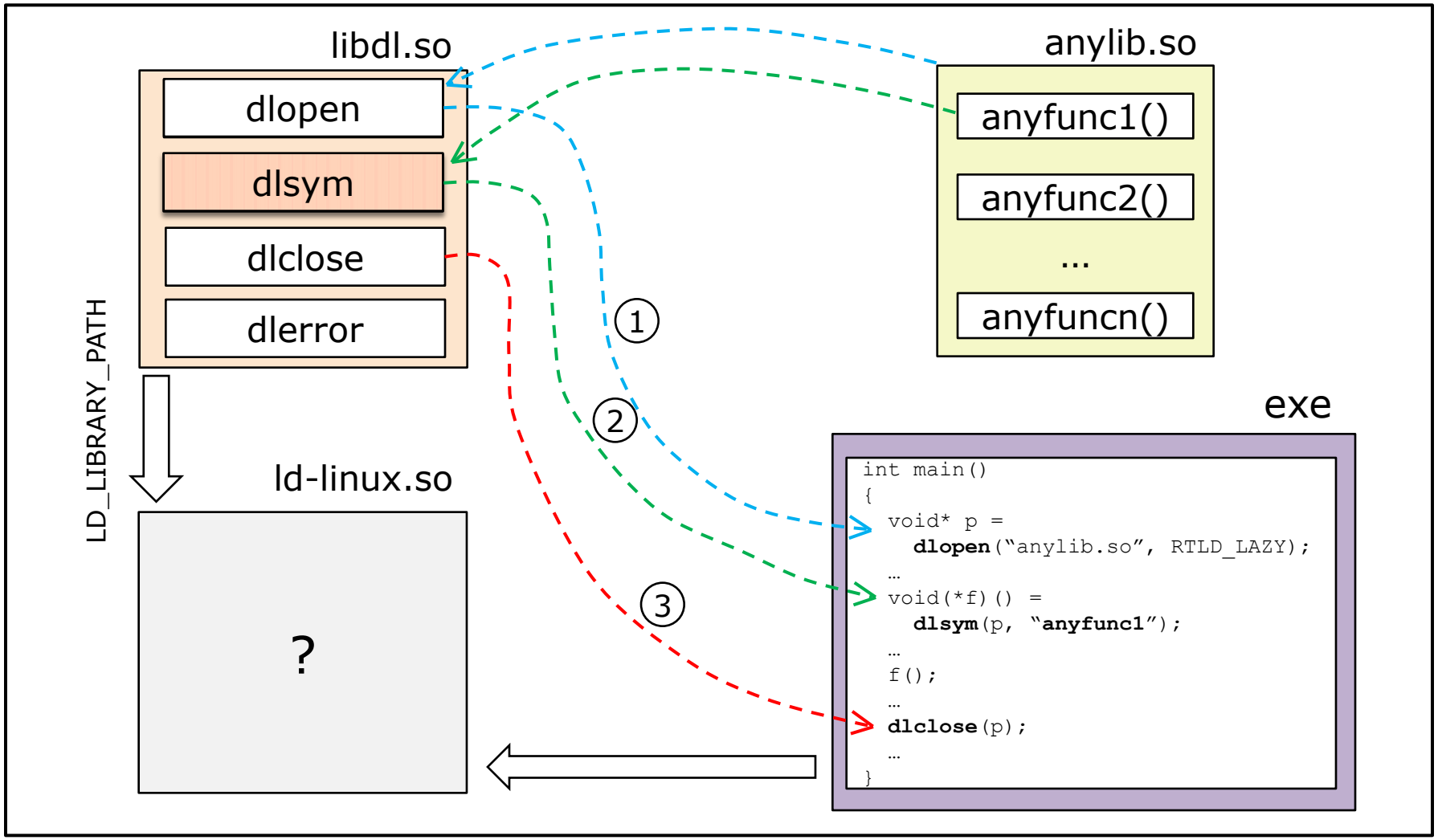
Symbol Table Structure

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

Parsing other sections of ELF

- For dynamic linking, the Dynamic linker primarily uses two processor-specific tables:
  - **Global Offset Table** (GOT)
  - **Procedure Linkage Table** (PLT)
- Dynamic linkers support PIC Code through the GOT in each shared library
- The GOT contains absolute addresses to all of the static data referenced in the program.

Dynamic Loading



Profiler components

- Shared library **libelfperf.so**
  - **Call redirection** and **function wrapping** mechanisms
  - Collecting of calls statistics
  - Memory management
- Modified **dynamic linker** (ld-linux.so)
  - Uses libelfperf.so for profiling of **dynamically linked** functions
  - Displays the results of profiling
- Modified **dynamic loading library** (libdl.so)
  - Uses libelfperf.so for profiling of **dynamically loaded** functions

Call redirection mechanism

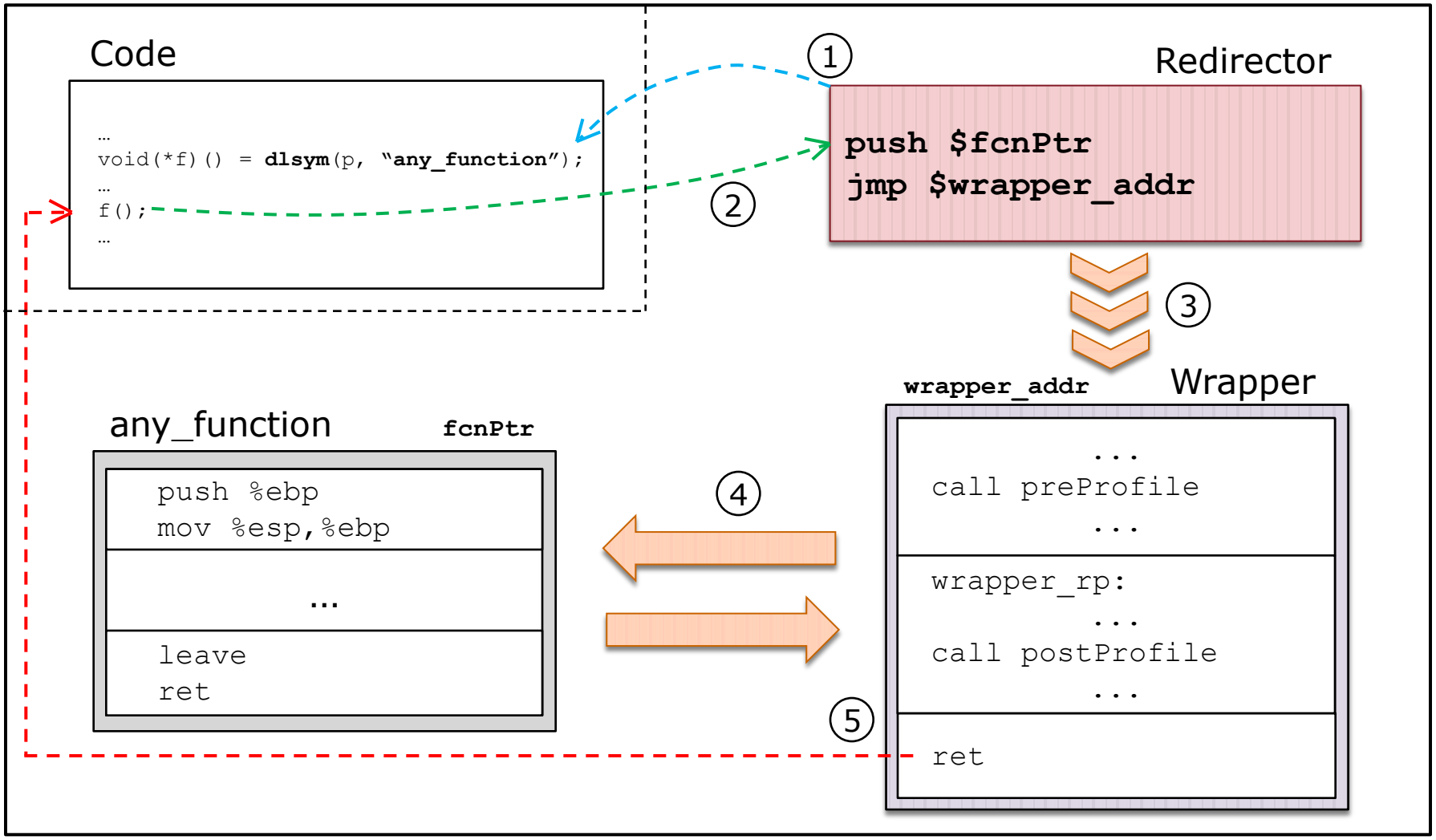
Calls redirection mechanism (**Redirector**) is a set of machine codes for the next assembly instructions:

```
push $fcnPtr
jmp $wrapper_addr
```

All they do is:

- Save address of profiled function **in program stack**
- Jump to **wrapper-function**

Redirector workflow



Redirector details

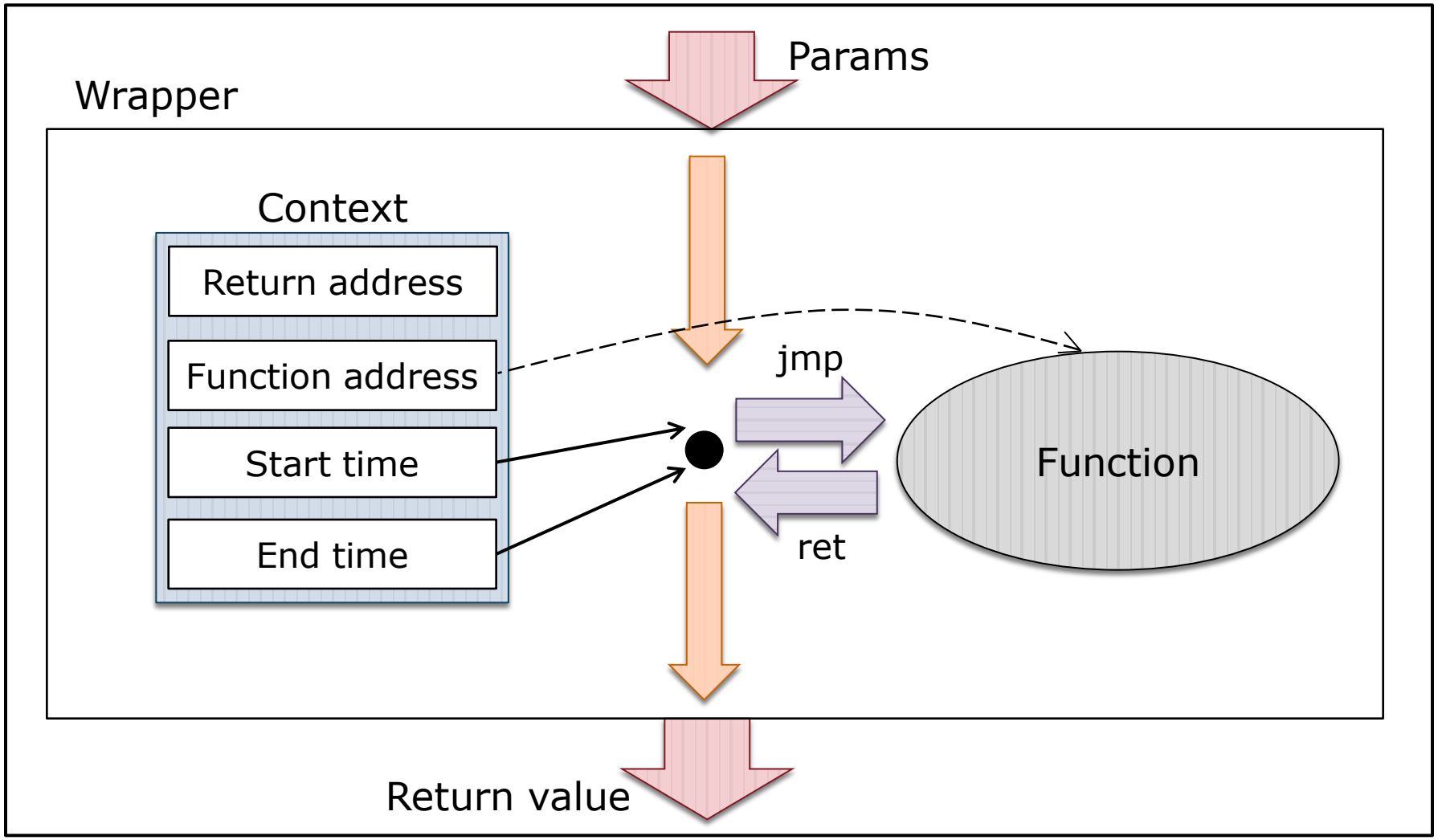
- Each redirector is created **individually** for each profiled function
- Redirectors are placed into **data segment** of process virtual memory
- The operating system allows to **mark** these memory areas **as executable**

Wrapping mechanism

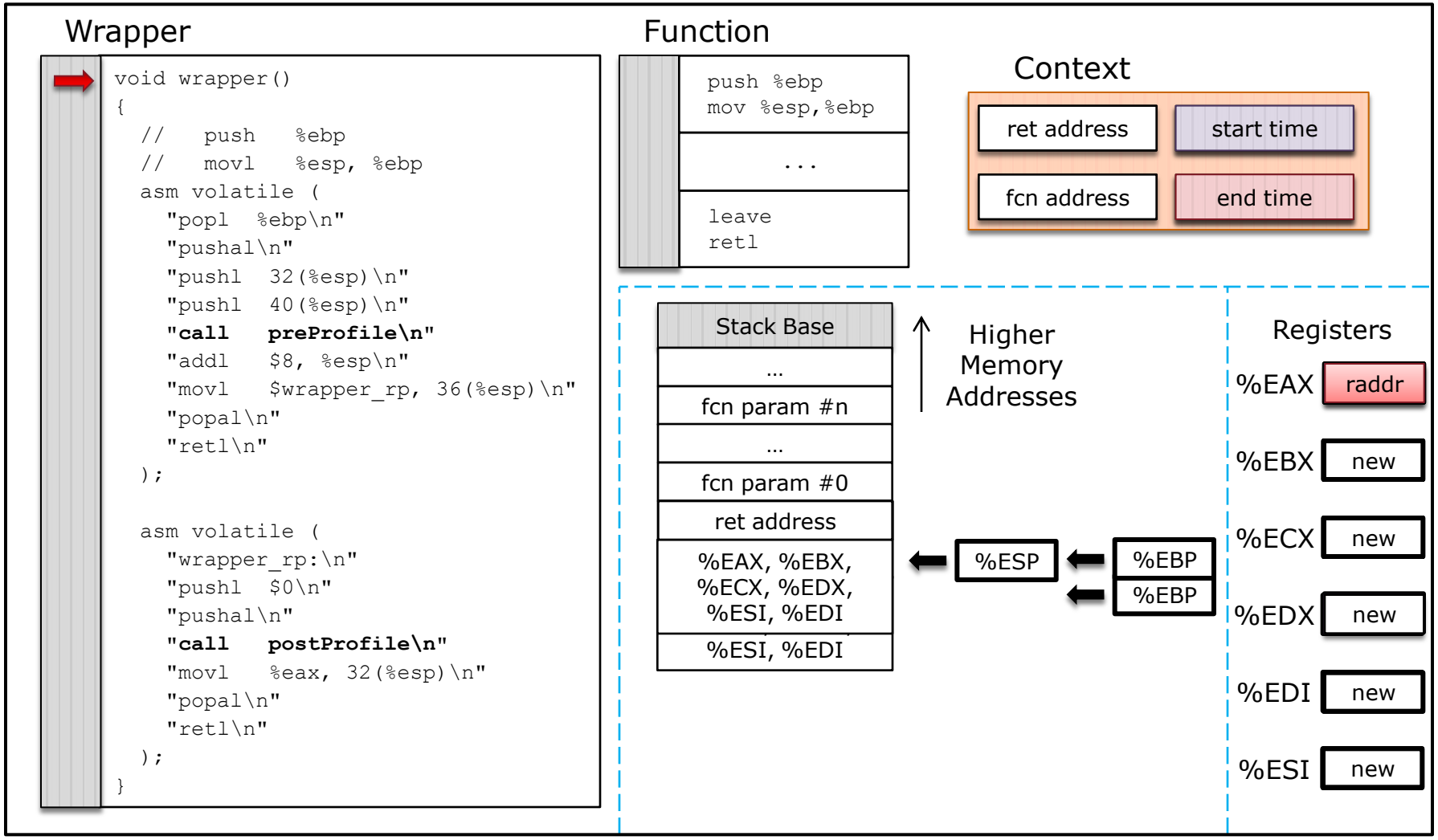
Function Wrapping mechanism (or **Wrapper**) is a function that does next things:

- **Takes control** from redirector
- Performs **pre-profile operations**
- Performs **replacement of return address**
- Performs **jump into** profiled **function**
- Again **takes control after** the work of profiled **function**
- Performs **post-profile operations**
- Returns **to caller**

Working scheme of Wrapper



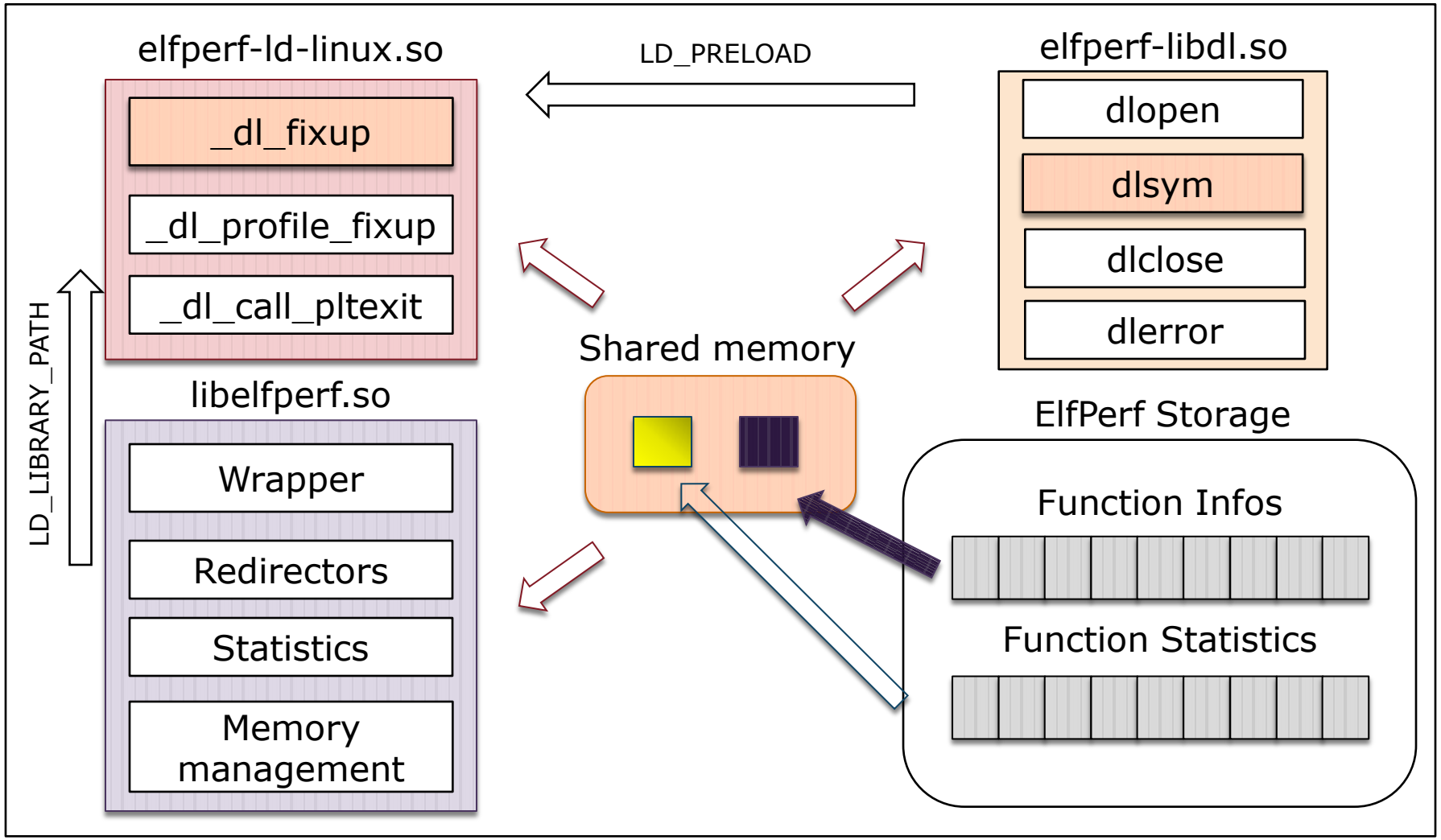
Implementation details (x86)



Wrapper details

- Wrapper **doesn't corrupt stack** content
- Wrapper exists in a **single copy for all functions** in each profiler implementation (x86 or x64)
- **Saving/Restoring of registers' state** allows to escape of uncontrollable changes in the program state
- Allows to profile **wide set of C/C++ functions**

Interaction of ElfPerf's components



Conclusion

Now we have:

- **«Light» profiler** based on «patched» ld-linux.so and libdl.so
- Support of profiling for **C/C++** functions from shared libraries
  (including libs compiled with **–fomit-frame-pointer** flag)
- Collecting of information about **number** and **total duration** of function calls
- Support of **both** x86 and x64 **platforms**

Links

- **Project resources:**
  - https://github.com/OSLL/elfperf
  - http://dev.osll.ru/projects/epat/wiki/
- **Contacts:**
  - http://osll.ru/
  - kirill.krinkin@gmail.com
  - edward.ryabikov@gmail.com
  - mark.zaslavskiy@gmail.com