Speeding up stack unwinding by compiling DWARF debugging data

Théophile Bastian
Under supervision of Francesco Zappa Nardelli, March – August 2018
PARKAS, INRIA

Internship synthesis

The general context

The standard debugging data format, DWARF (Debugging With Attributed Record Formats), contains tables permitting, for a given instruction pointer (IP), to understand how instructions from the assembly code relates to the original source code, where are variables currently allocated in memory or if they are stored in a register, what are their type and how to unwind the current stack frame. This information is generated when passing *eg.* the switch `-g` to `gcc` or equivalents.

Even in stripped (non-debug) binaries, a small portion of DWARF data remains: the stack unwinding data. This information is necessary to unwind stack frames, restoring machine registers to the value they had in the previous frame.

This data is structured into tables, each row corresponding to an IP range for which it describes valid unwinding data, and each column describing how to unwind a particular machine register (or virtual register used for various purposes). The vast majority of the rules actually used are basic – see Section 5.4 –, consisting in offsets from memory addresses stored in registers (such as %rbp or %rsp). Yet, the standard defines rules that take the form of a stack-machine expression that can access virtually all the process's memory and perform Turing-complete computations [8].

The research problem

As debugging data can easily grow larger than the program itself if stored carelessly, the DWARF standard pays a great attention to data compactness and compression. It succeeds particularly well at it, but at the expense of efficiency: accessing stack unwinding data for a particular program point is an expensive operation – the order of magnitude is 10 μs on a modern computer.

This is often not a problem, as stack unwinding is often thought of as a debugging procedure: when something behaves unexpectedly, the programmer might open their debugger and explore the stack. Yet, stack unwinding might, in some cases, be performance-critical: for instance, polling profilers repeatedly perform stack unwindings to observe which functions are active. Even worse, C++ exception handling relies on stack unwinding in order to find a suitable catch-block! For such applications, it might be desirable to find a different time/space trade-off, storing a bit more for a faster unwinding.

This different trade-off is the question that I explored during this internship: what good alternative trade-off is reachable when storing the stack unwinding data completely differently?

It seems that the subject has not been explored yet, and as of now, the most widely used library for stack unwinding, `libunwind` [6], essentially makes use of aggressive but fine-tuned caching and optimized code to mitigate this problem.

Your contribution

This internship explored the possibility to compile DWARF's stack unwinding data directly into native assembly on the x86_64 architecture, in order to provide fast access to the data at assembly level. This compilation process was fully implemented and tested on complex, real-world examples. The integration

of compiled DWARF into existing projects have been made easy by implementing an alternative version of the *de facto* standard library for this purpose, `libunwind`.

We explored and evaluated multiple approaches to determine which compilation process leads to the best time/space trade-off.

Unexpectedly, the part that proved hardest of the project was finding and implementing a benchmarking protocol that was both relevant and reliable. Unwinding one single frame is too fast to provide a reliable benchmarking on a few samples (around $10\,\mu s$ per frame) to avoid statistical errors. Having enough samples for this purpose – at least a few thousands – is not easy, since one must avoid unwinding the same frame over and over again, which would only benchmark the caching mechanism. The other problem is to distribute evenly the unwinding measures across the various IPs, among which those directly located into the loaded libraries (*eg.* the `libc`). The solution eventually chosen was to modify `perf`, the standard profiling program for Linux, in order to gather statistics and benchmarks of its unwindings. Modifying `perf` was an additional challenge that turned out to be harder than expected, since the source code is hard to read, and optimisations make some parts counter-intuitive. To overcome this, we designed an alternative version of `libunwind` interfaced with the compiled debugging data.

Arguments supporting its validity

The goal of this project was to design a compiled version of unwinding data that is faster than DWARF, while still being reliable and reasonably compact. Benchmarking has yielded convincing results: on the experimental setup created – detailed on Section 4 below –, the compiled version is around 26 times faster than the DWARF version, while it remains only around 2.5 times bigger than the original data.

We support the vast majority – more than 99.9 % – of the instructions actually used in binaries, although we do not support all of DWARF5 instruction set. We are almost as robust as libunwind: on a 27000 samples test, 885 failures were observed for `libunwind`, against 1099 for the compiled DWARF version (failures are due to signal handlers, unusual instructions, ...) – see Section 5.2.

The implementation is not yet release-ready, as it does not support 100 % of the DWARF5 specification [3] – see Section 3.2 below. Yet, it supports the vast majority – more than 99.9 % – of the cases seen in the wild, and is decently robust compared to `libunwind`, the reference implementation. Indeed, corner cases occur often, and on a 27000 samples test, 885 failures were observed for `libunwind`, against 1099 for the compiled DWARF version (see Section 5.2).

The implementation, however, is not yet production-ready: it only supports the x86_64 architecture, and relies to some extent on the Linux operating system. None of these pose a fundamental problem. Supporting other processor architectures and ABIs are only a matter of engineering. The operating system dependency is only present in the libraries developed in order to interact with the compiled unwinding data, which can be developed for virtually any operating system.

Summary and future work

In most cases of everyday's life, a slow stack unwinding is not a problem, left apart an annoyance. Yet, having a 26 times speed-up on stack unwinding-heavy tasks can be really useful to *eg.* profile large programs, particularly if one wants to profile many times in order to analyze the impact of multiple changes. It can also be useful for exception-heavy programs. Thus, we plan to address the limitations and integrate it cleanly with mainstream tools, such as `perf`.

Another research direction is to investigate how to compress even more the original DWARF unwinding data using outlining techniques, as we already do for the compiled data successfully.

Contents

Source code

Our implementation is available from `https://git.tobast.fr/m2-internship`. See the `abstract` repository for an introductive `README`.

1 Stack unwinding data presentation

1.1 Stack frames and x86_64 calling conventions

On every common platform, programs make use of a *call stack* to store information about the nested function calls at the current execution point, and keep track of their nesting. This call stack is conventionally a contiguous memory space mapped close to the top of the addressing space. Each function call has its own *stack frame*, an entry of the call stack, whose precise contents are often specified in the Application

Binary Interface (ABI) of the platform, and left to various extents up to the compiler. Those frames are typically used for storing function arguments, machine registers that must be restored before returning, the function's return address and local variables.

On the x86_64 platform, with which this report is mostly concerned, the calling convention followed on UNIX-like operating systems – among which Linux and MacOS – is defined by the System V ABI [9]. Under this calling convention, the first six arguments of a function are passed in the registers %rdi, %rsi, %rdx, %rcx, %r8, %r9, while additional arguments are pushed onto the stack. It also defines which registers may be overwritten by the callee, and which registers must be restored by the callee before returning. This restoration, for most compilers, is done by pushing the register value onto the stack during the function prelude, and restoring it just before returning. Those preserved registers are %rbx, %rsp, %rbp, %r12, %r13, %r14, %r15.

The register %rsp is supposed to always point to the last used address in the stack. Thus, when the process enters a new function, %rsp points to the location of the return address. Then, the compiler might use %rbp ("base pointer") to save this value of %rsp, writing the old value of %rbp below the return address on the stack and copying %rsp to %rbp. This makes it easy to find the return address from anywhere within the function, and allows for easy addressing of local variables. To some extents, it also allows for hot debugging, such as saving a useful core dump upon segfault. Yet, using %rbp to save %rip wastes a register, and the decision of using it, on x86_64 System V, is up to the compiler.

Usually, a function starts by subtracting some value to %rsp, allocating some space in the stack frame for its local variables. Then, it saves on the stack the values of the callee-saved registers that are overwritten later. Before returning, it pops the values of the saved registers back to their original registers and restore %rsp to its former value.

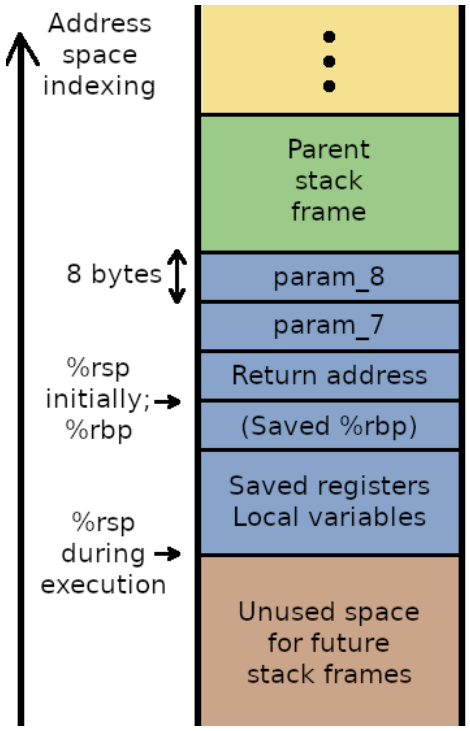

Figure 1: Program stack with x86_64 calling conventions

1.2 Stack unwinding

For various reasons, it is interesting, at some point of the execution of a program, to glance at its program stack and be able to extract information from it. For instance, when running a debugger, a frequent usage is to obtain a *backtrace*, that is, the list of all nested function calls at the current IP. This actually observes the stack to find the different stack frames, and decode them to identify the function names, parameter values, etc.

This operation is far from trivial. Often, a stack frame only makes sense when the machine registers hold the right values. These values, however, are to be restored from the previous stack frame, where they are stored. This imposes to *walk* the stack, reading the frames one after the other, instead of peeking at some frame directly. Moreover, it is often not even easy to determine the boundaries of each stack frame alone, making it impossible to just peek at a single frame.

Interpreting a frame in order to get the machine state *before* this frame, and thus be able to decode the next frame recursively, is called *unwinding* a frame.

Let us consider a stack with x86_64 calling conventions, such as shown in Figure 1. Assuming the compiler decided here *not* to use %rbp, and assuming the function allocates *eg.* a buffer of 8 integers, the area allocated for local variables is at least 32 bytes long (for 4-bytes integers), and %rsp points below this area. Left apart analyzing the assembly code produced, there is no way to find where the return address

is stored, relatively to %rsp, at some arbitrary point of the function. Even when %rbp is used, there is no easy way to guess where each callee-saved register is stored in the stack frame, since the compiler is free to do as it wishes. Even worse, it is not trivial to know callee-saved registers were at all, since if the function does not alter a register, it does not have to save it.

With this example, it seems pretty clear that some additional data is necessary to perform stack unwinding reliably, without only performing a guesswork. This data is stored along with the debugging information of a program, and one common format of debugging data is DWARF.

1.3 Unwinding usage and frequency

Stack unwinding is more frequent that one might think at first. The use case mostly thought of is simply to get a stack trace of a program, and provide a debugger with the information it needs. For instance, when inspecting a stack trace in `gdb`, a common operation is to jump to a previous frame:

```
(gdb) backtrace
#0  0x0000555555554625 in fct_b (m=0x5c) at segfault.c:5
#1  0x0000555555554663 in fct_a (n=42) at segfault.c:10
#2  0x0000555555554674 in main () at segfault.c:14
(gdb) frame 1
#1  0x0000555555554663 in fct_a (n=42) at segfault.c:10
10          fct_b((int*)(some_fct_a_var + 8));
(gdb) print some_fct_a_var
$1 = 84
```

To be able to do this, `gdb` must be able to restore `fct_a`'s context, by unwinding `fct_b`'s frame.

Yet, stack unwinding, and thus, debugging data, *is not limited to debugging*.

Another common usage is profiling. A profiler, such as `perf` under Linux – see Section 4.2 –, is used to measure and analyze in which functions a program spends its time, and find out which parts are critical to optimize. To do so, modern profilers pause the traced program at regular, short intervals, inspect their stack, and determine which function is currently being run. They also perform a stack unwinding to figure out the call path to this function, in order to determine which function indirectly takes time: for instance, a function `fct_a` can call both `fct_b` and `fct_c`, which both take a lot of time; spend practically no time directly in `fct_a`, but spend a lot of time in calls to the other two functions that were made from `fct_a`. Knowing that after all, `fct_a` is the culprit can be useful to a programmer.

Exception handling also requires a stack unwinding mechanism in some languages. Indeed, an exception is completely different from a `return`: while the latter returns to the previous function, at a well-defined IP, the former can be caught by virtually any function in the call path, at any point of the function. It is thus necessary to be able to unwind frames, one by one, until a suitable `catch` block is found. The C++ language, for one, includes a stack-unwinding library similar to `libunwind` in its runtime.

Technically, exception handling could be implemented without any stack unwinding, by using `setjmp` and `longjmp` mechanics [7]. However, it is not possible to implement this straight away in C++ (among others), because the stack needs to be properly unwound in order to trigger the destructors of stack-allocated objects. Furthermore, this is often undesirable: `setjmp` introduces an overhead, which is hit whenever a `try` block is encountered. Instead, it is often preferred to have strictly no overhead when no exception happens, at the cost of a greater overhead when an exception is actually fired – after all, they are supposed to be *exceptional*. For more details on C++ exception handling, see [5] (especially Section 16.5). Possible implementation mechanisms are also presented in [2].

In both of these two previous cases, performance *can* be a problem. In the latter, a slow unwinding directly impacts the overall program performance, particularly if a lot of exceptions are thrown and caught far away in their call path. As for the former, profiling *is* performance-heavy and slow: for a session analyzing the `tor-browser` for two and a half minutes, `perf` spends 100 μs analyzing each of the 325679 samples, that is, 300 ms per second of program run with default settings.

One of the causes that inspired this internship were also Stephen Kell's `libcrunch` [4], which makes a heavy use of stack unwinding through `libunwind` and had to force `gcc` to use a frame pointer (%rbp) everywhere through `-fno-omit-frame-pointer` in order to mitigate the slowness.

1.4 DWARF format

The DWARF format was first standardized as the format for debugging information of the ELF executable binaries (Extensible Linking Format), which are standard on UNIX-like systems, including Linux and MacOS – but not Windows. It is now commonly used across a wide variety of binary formats to store debugging information. As of now, the latest DWARF standard is DWARF 5 [3], which is openly accessible.

The DWARF data commonly includes type information about the variables in the original programming language, correspondence of assembly instructions with a line in the original source file, ... The format also specifies a way to represent unwinding data, as described in Section 1.2 above, in an ELF section originally called `.debug_frame`, but most often found as `.eh_frame`.

For any binary, debugging information can easily take up space and grow bigger than the program itself if no attention is paid at keeping it as compact as possible when designing the file format. On this matter, DWARF does an excellent job, and everything is stored in a very compact way. This, however, as we will see, makes it both difficult to parse correctly and relatively slow to interpret.

1.5 DWARF unwinding data

The unwinding data, which we will call from now on the `.eh_frame`, contains, for each possible IP, a set of "registers" that can be unwound, and a rule describing how to do so.

The DWARF language is completely agnostic of the platform and ABI, and in particular, is completely agnostic of a particular platform's registers. Thus, as far as DWARF is concerned, a register is merely a numerical identifier that is often, but not necessarily, mapped to a real machine register by the ABI.

In practice, this data takes the form of a collection of tables, one table per Frame Description Entry (FDE). A FDE, in turn, is a DWARF entry describing such a table, that has a range of IPs on which it has authority. Most often, but not necessarily, it corresponds to a single function in the original source code. Each column of the table is a register (*eg.* %rsp), along with two additional special registers, CFA (Canonical Frame Address) and RA (Return Address), containing respectively the base pointer of the current stack frame and the return address of the current function. For instance, on a x86_64 architecture, RA would contain the unwound value of %rip, the instruction pointer. Each row has a certain validity interval, on which it describes accurate unwinding data. This range starts at the instruction pointer it is associated with, and ends at the start IP of the next table row – or the end IP of the current FDE if it was the last row. In particular, there can be no "IP hole" within a FDE – unlike FDEs themselves, which can leave holes between them.

For instance, the C source code in Listing 1, when compiled with `gcc -O1 -fomit-frame-pointer -fno-stack-protector`, yields the assembly code in Listing 2. The memory layout of the stack frame is presented in Table 1, to help understanding how the stack frame is constructed. When interpreting the generated `.eh_frame` with `readelf -wF`, we obtain the (slightly edited) Listing 4. During the function prelude, *ie.* for $0x615 \leq \%rip < 0x619$, the stack frame only contains the return address, thus the CFA is 8 bytes above %rsp, and the return address is precisely at %rsp – that is, stored between %rsp and $\%rsp + 8$. Then, the contents of `fibo`, 8 integers of 4 bytes each, are allocated on the stack, which puts the CFA 32 bytes above %rsp; the return address still being 8 bytes below the CFA. The variable `pos` is optimized out in the generated assembly code, thus no stack space is allocated for it. Yet, `gcc` decided to allocate a total space of 48 bytes for the stack frame for memory alignment reasons, which means subtracting 40 bytes to %rsp (address 0x615 in the assembly). Then, by the end of the function, the local variables are discarded and %rsp is reset to its value from the first row.

```c
void fib7() {
    int fibo[8];
    fibo[0] = 1;
    fibo[1] = 1;
    for(int pos = 2; pos < 8; ++pos)
        fibo[pos] =
            fibo[pos - 1]
            + fibo[pos - 2];
    printf("%d\n", fibo[7]);
}
```

Listing 1: Original C

```
0000000000000615 <fib7>:
 615:   sub    $0x28,%rsp ; Alloc stack
 619:   movl   $0x1,(%rsp) ; fibo[0]
 620:   movl   $0x1,0x4(%rsp) ; fibo[1]
 628:   mov    %rsp,%rax ;     BEGIN FOR
 62b:   lea    0x18(%rax),%rcx
 62f:   mov    (%rax),%edx
 631:   add    0x4(%rax),%edx
 634:   mov    %edx,0x8(%rax)
 637:   add    $0x4,%rax
 63b:   cmp    %rcx,%rax
 63e:   jne    62f <fib7+0x1a> ; END FOR
 640:   mov    0x1c(%rsp),%esi
 644:   lea    0xb9(%rip),%rdi
 64b:   mov    $0x0,%eax
 650:   callq  520 <printf@plt>
 655:   add    $0x28,%rsp ; Restore rsp
 659:   retq
```

Listing 2: Generated assembly

```
[...] FDE [...] pc=615..65a
  DW_CFA_def_cfa: r7 (rsp) ofs 8
  DW_CFA_offset: r16 (rip) at cfa-8
  DW_CFA_advance_loc: 4 to 0619
  DW_CFA_def_cfa_offset: 48
  DW_CFA_advance_loc1: 64 to 0659
  DW_CFA_def_cfa_offset: 8
```

Listing 3: Raw DWARF

```
[...] FDE [...]  pc=615..65a
   LOC           CFA      ra
0000000000000615 rsp+8    c-8
0000000000000619 rsp+48   c-8
0000000000000659 rsp+8    c-8
```

Listing 4: Processed DWARF

$\%rsp + 0x30$	$\%rsp + 0x28$	$\%rsp + 0x20$	$\%rsp + 0x1c$	$\%rsp + 0x4$	$\%rsp$
Return Address	*Alignment space*	`fibo[7]`	...	`fibo[0]`	*Next frame*

Table 1: Stack frame schema for fib7 (horizontal layout)

However, DWARF data isn't actually stored as a table in the binary files, but is instead stored as in Listing 3. The first row has the location of the first IP in the FDE, and must define at least its CFA. Then, when all relevant registers are defined, it is possible to define a new row by providing a location offset (*eg.* here 4), and the new row is defined as a clone of the previous one, which can then be altered (*eg.* here by setting `CFA` to $\%rsp + 48$). This means that every line is defined *wrt.* the previous one, and that the IPs of the successive rows cannot be determined without evaluating every row that comes before in the first place. Thus, unwinding a frame from an IP close to the end of the frame requires evaluating pretty much every DWARF row in the table before reaching the relevant information, slowing down drastically the unwinding process.

1.6 How big are FDEs?

Since evaluating an `.eh_frame` FDE entry is, as seen in the previous section, roughly linear in time in its rows number, we must wonder what is the distribution of FDE rows count. The histogram in Figure 2 was generated on a random sample of around 2000 ELF files present on an ArchLinux system.

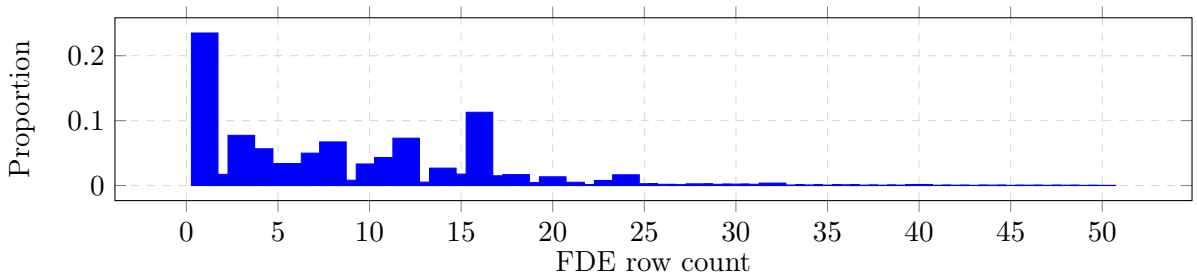

Figure 2: FDE line count density

1.7 Unwinding state-of-the-art

The most commonly used library to perform stack unwinding, in the Linux ecosystem, is `libunwind` [6]. While it is very robust and decently efficient, most of its optimization comes from fine-tuned code and good caching mechanisms. When parsing DWARF, `libunwind` is forced to parse the relevant FDE from its start, until it finds the row it was seeking.

2 DWARF semantics

The DWARF 5 standard [3] is written in English prose, and our first task is to formalize it. Thus, in this section, we first recall the informal behaviour of DWARF instructions as provided by the standard; and then we formalize their semantics by mapping them to well-defined C code. We omit the translation of DWARF expressions, because they form a rich language and would take a lot of time and space to formalize, while in the mean time being seldom used – see Section 5.4.

These semantics are defined *wrt.* the well-formalized C language, and are passing through an intermediate language. The DWARF language can read the whole memory, as well as registers, and is always executed for some instruction pointer. The C function representing it thus takes as parameters an array of the registers' values as well as an IP, and returns another array of registers values, which represents the evaluated DWARF row.

2.1 Original language: DWARF instructions

These are the DWARF instructions used for CFI description, that is, the instructions that contain the stack unwinding table information. Below is an exhaustive list of instructions from the DWARF5 specification [3] concerning CFI, with reworded descriptions for brevity and clarity. All these instructions are up to variants – most instructions exist in multiple formats to handle various operands formatting for space optimisation. Since we won't be talking about the underlying file format here, those variations between *eg.* `DW_CFA_advance_loc1` and `DW_CFA_advance_loc2` – which differ only on the number of bytes of their operand – are irrelevant and are eluded.

As said before, we also elude here references to DWARF expressions, as they are complex and are mostly not implemented in the actual compiler anyway – left apart some special cases. Those expressions bring in complexity, as they are turing-complete stack machine expressions that can access virtually the whole computer's memory. Formalizing them would require designing semantics for such a language.

- `DW_CFA_set_loc(loc)`: start a new table row from address loc
- `DW_CFA_advance_loc(delta)`: start a new table row at address $prev\_loc + delta$
- `DW_CFA_def_cfa(reg, offset)`: sets this row's CFA at $(\%reg + offset)$
- `DW_CFA_def_cfa_register(reg)`: sets CFA at $(\%reg + prev\_offset)$
- `DW_CFA_def_cfa_offset(offset)`: sets CFA at $(\%prev\_reg + offset)$

- `DW_CFA_def_cfa_expression(expr)`: sets CFA as the result of *expr*
- `DW_CFA_undefined(reg)`: sets the register %reg as undefined in this row
- `DW_CFA_same_value(reg)`: declares that the register %reg hasn't been touched, or was restored to its previous value, in this row. An unwinding procedure can leave it as-is.
- `DW_CFA_offset(reg, offset)`: the value of the register %reg is stored in memory at the address $CFA + \mathit{offset}$.
- `DW_CFA_val_offset(reg, offset)`: the value of the register %reg is the value $CFA + \mathit{offset}$
- `DW_CFA_register(reg, model)`: the register %reg has, in this row, the value of $\%\mathit{model}$.
- `DW_CFA_expression(reg, expr)`: the value of %reg is stored in memory at the address defined by *expr*
- `DW_CFA_val_expression(reg, expr)`: %reg has the value of *expr*
- `DW_CFA_restore(reg)`: %reg has the same value as in this FDE's preamble (CIE) in this row. This is *not implemented in this semantics* for simplicity and brevity, as we would have to introduce CIE (preamble) and FDE (body) independently. This is also not often used in actual ELF files: the analysis in Section 5.4 found no such instruction, on a random uniform sample of 4000 ELF files.
- `DW_CFA_remember_state()`: push the state of all the registers of this row on a state-saving stack
- `DW_CFA_restore_state()`: pop an entry of the state-saving stack, and restore all registers in this row to the value held in the stack record.
- `DW_CFA_nop()`: do nothing (padding)

2.2 Intermediary language $\mathcal{I}$

A first pass translates DWARF instructions into this intermediate language $\mathcal{I}$. It is designed to be more mathematical, representing the same thing, but abstracting all the data compression of the DWARF format away, so that we can better reason on it and transform it into C code.

Its grammar is as follows:

$$
\begin{array}{rll}
\text{FDE} ::= & (\mathbb{Z} \times \text{Row})^* & \text{FDE (set of rows)} \\
\text{Row} ::= & \mathbb{V}^{\mathbb{R}} & \text{A single table row} \\
\mathbb{R} ::= & \{0, 1, \ldots, \text{NB\_REGS-1}\} & \text{Machine registers} \\
\mathbb{V} ::= & \bot & \text{Values: undefined,} \\
& \mid \text{Addr}\,(\mathbb{E}) & \text{at address } x, \\
& \mid \text{Val}\,(\mathbb{E}) & \text{of value } x \\
& \mid \text{Expr} & \text{of expression } x\text{, see in text} \\
\mathbb{E} ::= & \mathbb{R} \times \mathbb{Z} & \text{A ``simple'' expression } \%\mathit{reg} + \mathit{offset}
\end{array}
$$

The entry point of the grammar is a FDE, which is a set of rows, each annotated with the machine address from which it is valid. The addresses are necessarily increasing within a FDE.

Each row is as a function mapping registers to values, and represents a row of the unwinding table.

We implicitly consider that $\%\mathit{reg}$ maps to a number. We use here `x86_64` names for convenience, although in DWARF, registers are merely identifiers. Thus, we can safely state that $\%\mathit{reg} \in \mathbb{R}$.

A value can then be undefined, stored at memory address x or be directly a value x, x being here a simple expression consisting of $\%\mathit{reg} + \mathit{offset}$. The CFA is seen just as any other register here, although DWARF makes a distinction between it and other columns. For instance, to define $\%rax$ to the value contained in memory 16 bytes below the CFA, we would have $\%rax \mapsto \text{Addr}\,(\%CFA, -16)$, since the stack

grows downwards. We also leave open the possibility to extend the language with DWARF expressions support as Expr , although we *do not* specify them here.

2.3 Target language: a C function body

The target language of these semantics is a C function, to be interpreted *wrt.* the C11 standard [1]. The function is supposed to be run in the context of the program being unwound. In particular, it must be able to dereference some pointer derived from DWARF instructions that points to the execution stack, or even the heap.

This function takes as arguments an instruction pointer – supposedly extracted from $\%rip$ – and an array of register values; and returns a fresh array of register values after unwinding this call frame. The function is compositional: it can be called twice in a row to unwind two stack frames, unless the IP obtained after the first unwinding comes from another shared object file, for instance a call to `libc`. In this case, unwinding the second frame requires loading the corresponding DWARF information.

The function is the following:

```c
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

#define NB_REGS 32 /* put the number of registers of your platform here */

typedef uintptr_t* regs_t; // Array of size at least NB_REGS

regs_t unwind_frame(uintptr_t ip, regs_t old_ctx) {
    regs_t new_ctx = (regs_t) malloc(sizeof(uintptr_t) * NB_REGS);
    assert(new_ctx != NULL);

    // ===== INSERT GENERATED CODE HERE =====

end_ifs:
    return new_ctx;
}
```

The translation of $\mathcal{I}$ as produced by the later-defined function are then to be inserted in this context, where the comment states so.

In pseudo-C code (for brevity) and assuming the functions and types used are duly defined elsewhere, unwinding multiple frames would then look like this:

```c
while(!unwinding_done()) {
    unwind_fct_t unwind_fct = get_unwinder_for_IP(current_context[RA]);
    current_context = unwind_fct(current_context[RA], current_context);
    do_something_with_context(current_context);
}
```

Thus, if we hold for true that the IP remains in the same memory segment – *ie.* binary file – for two frames, we can safely unwind two frames this way:

```c
for(int i = 0; i < 2; ++i)
    current_context = unwind_frame(current_context[RA], current_context);
```

2.4 From DWARF to $\mathcal{I}$

In DWARF, the instructions have a meaning that refer to previously interpreted instructions, sequentially. For instance, many registers are defined at offsets from the current CFA, which in turn was previously defined *wrt.* the former CFA value, etc. Thus, to give a meaning to a DWARF instruction, knowledge of the current row's values is needed. Let us consider a given point of the interpretation of $d = h \cdot t$, where we already have interpreted h, the first instructions, and interpreted it as $H \in \text{FDE}$, while

t remains to be interpreted. We then define the interpretation function $[\![t]\!](H)$, interpreting the remainder t of the DWARF instructions, having the knowledge of H, the current interpreted row.

But we also need to keep track of this state-saving stack DWARF uses, which is kept in subscript. Thus, we define $[\![\bullet]\!]^{\mathcal{I}}_s(\bullet) : \text{DWARF} \times \text{FDE} \to \text{FDE}$, for s a stack of Row, that is,

$$s \in \mathbb{S} := \text{Row}^*$$

Implicitly, $[\![\bullet]\!]^{\mathcal{I}} := [\![\bullet]\!]^{\mathcal{I}}_{\varepsilon}$

For convenience, we define $\xleftarrow{r \in \mathbb{R}}$, an operator changing the value assigned to a register, its right-hand side operand, in the last row of a given FDE, its left-hand side operand.

$$(f \in \text{FDE}) \xleftarrow{r \in \mathbb{R}} (v \in values) \quad := \quad (f\,[0\ \cdots\ (|f|-2)]) \cdot \begin{cases} r' \neq r & \mapsto (f[-1])\,(r') \\ r & \mapsto v \end{cases}$$

Note that for convenience, we allow ourselves to index negatively an array to retrieve its values from the end; thus, $f[-1]$ refers to the last entry of f. If we consider the fictive following fictive row R_0,

$$R \in \text{Row} := \begin{cases} CFA & \mapsto \text{Val}\,(\%rsp - 48) \\ \%rbx & \mapsto \text{Addr}\,(\%rsp - 16) \end{cases}$$

then, we would have

$$R \xleftarrow{\%\text{rbx}} (\text{Addr}\,(\%rip - 24)) \quad = \quad \begin{cases} CFA & \mapsto \text{Val}\,(\%rsp - 48) \\ \%rbx & \mapsto \text{Addr}\,(\%rsp - 24) \end{cases}$$

The same way, we define $\xrightarrow{\text{reg}}$ that *extracts* the rule currently applied for $\%reg$ in the last row of a FDE, *eg.* $F \xrightarrow{\text{CFA}} \text{Val}\,(\%reg + \text{off})$. If the rule currently applied in such a case is *not* of the form $\%reg + \text{off}$, then the program is considered erroneous. One can see this $\xrightarrow{\text{reg}}$ as a `match` statement in OCaml, but with only one case, allowing to retrieve packed data, all the other unmatched cases corresponding to an error.

$$\begin{aligned}
[\![\varepsilon]\!]^{\mathcal{I}}_s(F) &:= F \\
[\![\texttt{DW_CFA_set_loc(loc)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= [\![d]\!]^{\mathcal{I}}_s\,(F \cdot (loc, F[-1].row)) \\
[\![\texttt{DW_CFA_adv_loc(delta)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= [\![d]\!]^{\mathcal{I}}_s\,(F \cdot (F[-1].addr + delta, F[-1].row)) \\
[\![\texttt{DW_CFA_def_cfa(reg, offset)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= [\![d]\!]^{\mathcal{I}}_s\left(F \xleftarrow{\text{CFA}} \text{Val}\,(\%reg + offset)\right) \\
[\![\texttt{DW_CFA_def_cfa_register(reg)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= \text{let F} \xrightarrow{\text{CFA}} \text{Val}\,(\%oldreg + oldoffset) \text{ in} \\
&\qquad [\![d]\!]^{\mathcal{I}}_s\left(F \xleftarrow{\text{CFA}} \text{Val}\,(\%reg + oldoffset)\right) \\
[\![\texttt{DW_CFA_def_cfa_offset(offset)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= \text{let F} \xrightarrow{\text{CFA}} \text{Val}\,(\%oldreg + oldoffset) \text{ in} \\
&\qquad [\![d]\!]^{\mathcal{I}}_s\left(F \xleftarrow{\text{CFA}} \text{Val}\,(\%oldreg + offset)\right) \\
[\![\texttt{DW_CFA_undefined(reg)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= [\![d]\!]^{\mathcal{I}}_s\left(F \xleftarrow{\%\text{reg}} \bot\right) \\
[\![\texttt{DW_CFA_same_value(reg)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= \text{Val}\,(\%reg) \\
[\![\texttt{DW_CFA_offset(reg, offset)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= [\![d]\!]^{\mathcal{I}}_s\left(F \xleftarrow{\text{reg}} \text{Addr}\,(CFA + offset)\right) \\
[\![\texttt{DW_CFA_val_offset(reg, offset)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= [\![d]\!]^{\mathcal{I}}_s\left(F \xleftarrow{\text{reg}} \text{Val}\,(CFA + offset)\right) \\
[\![\texttt{DW_CFA_register(reg, model)} \cdot d]\!]^{\mathcal{I}}_s(F) &:= \text{let } F \xrightarrow{\text{model}} r \text{ in } [\![d]\!]^{\mathcal{I}}_s\left(F \xleftarrow{\text{reg}} r\right)
\end{aligned}$$

$$[\![\texttt{DW\_CFA\_remember\_state}\,() \cdot d]\!]^{\mathcal{I}}_s(F) := [\![d]\!]^{\mathcal{I}}_{s\,\cdot\,(F[-1].row)}(F)$$
$$[\![\texttt{DW\_CFA\_restore\_state}\,() \cdot d]\!]^{\mathcal{I}}_{s\,\cdot\,t}(F) := [\![d]\!]^{\mathcal{I}}_s(F\,[0\ldots|F|-2] \cdot (F[-1].addr, t))$$
$$[\![\texttt{DW\_CFA\_nop}\,() \cdot d]\!]^{\mathcal{I}}_s(F) := [\![d]\!]^{\mathcal{I}}_s(F)$$

The state-saving stack is used only for `remember_state` and `restore_state`. If we were to omit those two operations, we could plainly remove the stack from our notations.

2.5 From $\mathcal{I}$ to C

The C code provided thereafter is a correct but inefficient reference implementation, which is only provided to specify DWARF wrt. C. In particular, the actual compiler is not *implemented this way.*

We now define $[\![\bullet]\!]^C : \text{DWARF} \to C$, in the context presented earlier in Section 2.3. The translation from $\mathcal{I}$ to C is defined as follows:

- $[\![\varepsilon]\!]^C =$

```
for(int reg=0; reg < NB_REGS; ++reg)
    new_ctx[reg] = [⊥]^R;
```

- $[\![(\text{loc}, \text{row}) \cdot t]\!]^C = C\_code \cdot [\![t]\!]^C$, where C_code is

```
if(ip >= loc) {
    for(int reg=0; reg < NB_REGS; ++reg)
        new_ctx[reg] = [row[reg]]^R;
    goto end_ifs; // Avoid using 'else if' (easier for generation)
}
```

while $[\![\bullet]\!]^R$ is defined as

$$\begin{aligned}
[\![\bot]\!]^R &= \texttt{ERROR_VALUE} \\
[\![\,\text{Addr}\,(\text{reg}, \textit{offset})]\!]^R &= \texttt{*(old_ctx[reg] + offset)} \\
[\![\,\text{Val}\,(\text{reg}, \textit{offset})]\!]^R &= \texttt{(old_ctx[reg] + offset)}
\end{aligned}$$

2.6 Concerning correctness

The semantics described in this section are designed in a concern of *formalization* of the original standard. This standard, sadly, only describes in plain English each instruction's action and result. This basis cannot be used to *prove* anything correct without relying on informal interpretations.

3 Stack unwinding data compilation

In this section, we will study all the design options we explored for the actual C implementation.

3.1 Code availability

All the code produced during the course of this internship is available on the various repositories from https://git.tobast.fr/m2-internship/. The repositories contain `README` files describing them; a summary and global description can be found in the `abstract` repository. This should be detailed enough to run the project. The source code is entirely under free software licenses.

3.2 Compilation: `eh_elfs`

The rough idea of the compilation is to produce, out of the `.eh_frame` section of a binary, C code close to that of Section 2 above. This C code is then compiled by GCC in `-O2` mode. This saves us the trouble of optimizing the generated C code whenever GCC does that by itself.

The generated code consists in a single function, `_eh_elf`, taking as arguments an instruction pointer and a memory context (*ie.* the value of the various machine registers) as defined in Listing 5. The function then returns a fresh memory context loaded with the values the registers after unwinding this frame.

The body of the function itself consists in a single monolithic switch, taking advantage of the non-standard – yet overwhelmingly implemented in common C compilers – syntax for range switches, in which each `case` can refer to a range, *eg.* `case 17 ... 42:`. All the FDEs are merged together into this switch, each row of a FDE being a switch case. Separating the various FDEs in the C code – other than with comments – is, unlike what is done in DWARF, pointless, since accessing a "row" has a linear cost, and the C code is not meant to be read, except maybe for debugging purposes. The switch cases bodies then fill a context with unwound values before return it.

A setting of the compiler also optionally enables another parameter to the `_eh_elf` function, `deref`, which is a function pointer. This `deref` function, when present, replaces everywhere the dereferencing `*` operator, and can be used to generate `eh_elfs` that works on remote address spaces, that is, whenever the unwinding is not done on the process reading the `eh_elf` itself, but some other process, or even on a stack dump of a long-terminated process.

Unlike in the `.eh_frame`, and unlike what should be done in a release, real-world-proof version of the `eh_elfs`, the choice was made to keep this implementation simple, and only handle the few registers that were needed to simply unwind the stack. Thus, the only registers handled in `eh_elfs` are %rip, %rbp, %rsp and %rbx, the latter being used a few times in `libc` and other less common libraries to hold the CFA address in common functions. This is enough to unwind the stack reliably, and thus enough for profiling, but is not sufficient to analyze every stack frame as `gdb` would do after a `frame n` command. Yet, if one was to enhance the code to handle every register, it would not be much harder and would probably be only a few hours worth of code refactoring and rewriting.

```
1 typedef struct {
2     uint8_t flags;
3     uintptr_t rip, rsp, rbp, rbx;
4 } unwind_context_t;
```

Listing 5: Unwinding context

In the unwind context from Listing 5, the values of type `uintptr_t` are the values of the corresponding registers, and `flags` is a 8-bits value, indicating for each register whether it is present or not in this context, plus an error bit, indicating whether an error occurred during unwinding. Such errors can be due *eg.* to an unsupported operation in the original DWARF. This context differs from the one presented in Section 2.3, since the previous one was only an array of values, and the one from the real implementation is more robust, in particular by including an error flag by lack of $\perp$ value.

This generated data is stored in separate shared object files, which we call `eh_elfs`. It would have been possible to alter the original ELF file to embed this data as a new section, but getting it to be executed just as any portion of the `.text` section would probably have been painful, and keeping it separated during the experimental phase is convenient. It is possible to have multiple versions of `eh_elfs` files in parallel, with various options turned on or off, and it doesn't require to alter the base system by editing *eg.* `/usr/lib/libc-*.so`. Instead, when the `eh_elf` data is required, those files can simply be `dlopen`'d. It is also possible to imagine, in a future environment production, packaging `eh_elfs` files separately, so that people interested in better performance can have the choice to install them.

This, in particular, means that each ELF file has its unwinding data in a separate `eh_elf` file,

implying that the unwinding data for a given program is scattered among various `eh_elf` files, one for each shared object loaded – just like with DWARF, where each ELF retains its own DWARF data. Thus, an unwinder must first acquire a *memory map*, a table listing the various ELF files loaded and *mapped* in memory, and on which memory segment. This memory map is provided by the operating system – for instance, on Linux, it is available as a file in `/proc`, a special part of the file system that the kernel uses to communicate with the userland processes. Once this map is acquired, when unwinding from a given IP, the unwinder must identify the memory segment from which it comes, deduce the source ELF file, and deduce the corresponding `eh_elf`.

```
unwind_context_t _eh_elf(unwind_context_t ctx, uintptr_t pc) {
    unwind_context_t out_ctx;
    switch(pc) {
        // [...] Previous FDEs redacted for brevity
        case 0x615 ... 0x618:
            out_ctx.rsp = ctx.rsp + (8);
            out_ctx.rip = *((uintptr_t*)(out_ctx.rsp + (-8)));
            out_ctx.flags = 3u;
            return out_ctx;
        // [...] Further lines and FDEs redacted for brevity
        default:
            out_ctx.flags = 128u;
            return out_ctx;
        }
}
```

Listing 6: `eh_elf` for the previous example

The C code in Listing 6 is the relevant part of what was generated for the C code in Listing 1.

3.3 First results

Without any particular care to efficiency or compactness, it is already possible to produce a compiled version very close to the one described in Section 2. Although the unwinding speed cannot yet be actually benchmarked, it is already possible to write in a few hundred lines of C code a simple stack walker printing the functions traversed. It already works well on the standard cases that are easily tested, and can be used to unwind the stack of simple programs.

The major drawback of this approach, without any particular care taken, is the waste of space. The space taken by those tentative `eh_elfs` is analyzed in Table 2 for `hackbench`, a small program introduced later in Section 4.3, and the libraries on which it depends.

Shared object	Original program size	Original `.eh_frame`	Generated `eh_elf .text`	% of original program size	Growth factor
libc-2.27.so	1.4 MiB	130.1 KiB	914.9 KiB	63.92	7.03
libpthread-2.27.so	58.1 KiB	11.6 KiB	70.5 KiB	121.48	6.09
ld-2.27.so	129.6 KiB	9.6 KiB	71.7 KiB	55.34	7.44
hackbench	2.9 KiB	568.0 B	2.1 KiB	74.78	3.97
Total	1.6 MiB	151.8 KiB	1.0 MiB	65.32	6.98

Table 2: Basic `eh_elfs` space usage

The first column only includes the sizes of the ELF sections `.text` (the program itself) and `.rodata`, the read-only data (such as static strings, etc.). Only the weight of the `.text` section of the generated `eh_elfs` is considered, because it is self-contained (few data or none is stored in `.rodata`), and the other sections could be removed if the `eh_elfs .text` was somehow embedded in the original shared object.

This first tentative version of `eh_elfs` is roughly 7 times heavier than the original `.eh_frame`, and represents a far too significant proportion of the original program size (65 %).

3.4 Space optimization

A lot of small space optimizations, such as filtering out empty FDEs, merging together the rows that are equivalent on all the registers kept, etc. were made in order to shrink the size of the `eh_elfs`.

The optimization that most reduced the output size was to use an if/else tree implementing a binary search on the instruction pointer relevant intervals, instead of a single monolithic switch. In the process, we also *outline* code whenever possible, that is, find out identical "switch cases" bodies – which are not switch cases anymore, but `if` bodies –, move them outside of the if/else tree, identify them by a label, and jump to them using a `goto`, which de-duplicates a lot of code and contributes greatly to the shrinking. In the process, we noticed that the vast majority of FDE rows are actually taken among very few "common" FDE rows. For instance, in the `libc`, out of a total of 20827 rows, only 302 (1.5 %) unique rows remain after the outlining.

This makes this optimization really efficient, as seen later in Section 5.3, but also makes it an interesting question – not investigated during this internship – to find out whether standard DWARF data could be efficiently compressed in this way.

```
unwind_context_t _eh_elf(unwind_context_t
     ctx, uintptr_t pc) {
    unwind_context_t out_ctx;
    if(pc < 0x619) {
        // IP=0x615 ... 0x618
        goto _factor_3;
    } else {
        if(pc < 0x659) {
            // IP=0x619 ... 0x658
            goto _factor_4;
        } else {
            // IP=0x659 ... 0x659
            goto _factor_3;
        }
    }

    _factor_default:
    out_ctx.flags = 128u;
    return out_ctx;

    /* ===== LABELS  ======== */

    _factor_4:
    out_ctx.rsp = ctx.rsp + (48);
    out_ctx.rip = *((uintptr_t*)(out_ctx.
    rsp + (-8)));
    out_ctx.flags = 3u;
    return out_ctx;

    _factor_3:
    out_ctx.rsp = ctx.rsp + (8);
    out_ctx.rip = *((uintptr_t*)(out_ctx.
    rsp + (-8)));
    out_ctx.flags = 3u;
    return out_ctx;
}
```

Listing 7: `eh_elf` for the previous example

3.5 Implementation correctness

The core part of the code, the `if/else if` bodies, are in substance strictly equivalent to what was devised in the semantics in Section 2. Indeed, the semantics were designed *wrt.* C code to be able to export them directly to the C code generated by the compiler. As a result, this part's proof is limited to checking that no errors were made when copying the code snippets.

The only things that remain to be proved are thus code optimisations, namely the binary search transformation of the `switch` statement and the outlining. The binary search part proof is a common proof, which in substance requires proving that binary search is equivalent to iterative, exhaustive search. The outlining only consists in proving that the program flow remains unchanged, which is easy to prove.

4 Benchmarking

Benchmarking turned out to be, quite surprisingly, the hardest part of the project. It ended up requiring a good deal of investigation to find a working protocol, and afterwards, a good deal of code reading and coding to get the solution working.

4.1 Requirements

To provide relevant benchmarks of the `eh_elfs` performance, one must sample at least a few hundreds or thousands of stack unwindings, since a single frame unwinding with regular DWARF takes the order of magnitude of $10\,\mu s$, and `eh_elfs` were expected to have significantly better performance.

However, unwinding over and over again from the same program point would have had no interest at all, since `libunwind` would have simply cached the relevant DWARF rows. In the mean time, making sure that the various unwindings are made from different locations is somehow cheating, since it makes useless `libunwind`'s caching and does not reproduce "real-world" unwinding distribution. All in all, the benchmarking method must have a "natural" distribution of unwindings.

Another requirement is to also distribute evenly enough the unwinding points across the program to mimic real-world unwinding: we would like to benchmark stack unwindings crossing some standard library functions, starting from inside them, etc.

Finally, the unwound program must be interesting enough to call functions often, building a stack of nested function calls (at least frequently 5), have FDEs that are not as simple as in Listing 4, etc.

4.2 Presentation of `perf`

`Perf` is a *profiler* that comes with the Linux ecosystem, and is even developed within the Linux kernel source tree. A profiler is an important tool from the developer's toolbox that analyzes the performance of programs by recording the time spent in each function, including within nested calls. This analysis often enables programmers to optimize critical paths and functions in their programs, while leaving unoptimized functions that are seldom traversed.

`Perf` is a *polling* profiler, to be opposed with *instrumenting* profilers. This means that with `perf`, the basic idea is to stop the traced program at regular intervals, unwind its stack, write down the current nested function calls, and integrate the sampled data in the end. Instrumenting profilers, on the other hand, do not interrupt the program, but instead inject code in it.

4.3 Benchmarking with `perf`

In the context of this internship, the main advantage of `perf` is that it unwinds the stack on a regular, controllable basis, easily unwinding thousands of time in a few seconds. It also meets all the requirements from Section 4.1 above: since it stops at regular intervals and unwinds, the unwindings are evenly distributed *wrt.* the frequency of execution of the code, which is a natural enough setup for the benchmarks to be meaningful, while still unwinding from diversified locations, preventing caching from being be overwhelming – as can be observed later in Section 5.2. It also has the ability to unwind from within any function, included functions of linked shared libraries. It can also be applied to virtually any program, which allows unwinding "interesting" code.

The program that was chosen for `perf`-benchmarking is `hackbench` [11]. This small program is designed to stress-test and benchmark the Linux scheduler by spawning processes or threads that communicate with each other. It has the interest of generating stack activity, being linked against `libc` and `pthread`, and being very light.

Interfacing `eh_elfs` with `perf` required, in a first place, to fork `libunwind` and implement `eh_elfs` support for it. In the process, it turned out necessary to slightly modify `libunwind`'s interface to add a parameter to an initialisation function, since `libunwind` is made to be agnostic of the system and process

as much as possible, to be able to unwind in any context. This very restricted information lacked a memory map (see Section 3.2) in order to use `eh_elfs` – while, on the other hand, providing information about the original DWARF that are now useless. Apart from this, the modified version of `libunwind` produced is entirely compatible with the vanilla version. This means that the only modifications required to use `eh_elfs` within any project using `libunwind` should be changing one line of code to add one parameter to a function call and linking against the modified version of `libunwind` instead of the system version.

Once this was done, plugging it in `perf` was the matter of a few lines of code only, left apart the benchmarking code. The major difficulty was to understand how `perf` works. To avoid perturbing the traced program, `perf` does not unwind at runtime, but rather records at regular intervals the program's stack, and all the auxiliary information that is needed to unwind later. This is done when running `perf record`. Then, a subsequent call to `perf report` unwinds the stack to analyze it; but at this point of time, the traced process is long dead. Thus, any PID-based approach, or any approach using `/proc` information fails. However, as this was the easiest method, the first version of `eh_elfs` used those mechanisms; it took some code rewriting to move to a PID- and `/proc`-agnostic implementation.

4.4 Other explored methods

The first approach tried to benchmark was trying to create some specific C code that would meet the requirements from Section 4.1, while calling itself a benchmarking procedure from time to time. This was quickly abandoned, because generating C code interesting enough to be unwound turned out hard, and the generated FDEs invariably ended out uninteresting. It would also never have met the requirement of unwinding from fairly distributed locations anyway.

Another attempt was made using CSmith [10], a random C code generator designed for random testing on C compilers. The idea was still to craft a C program that would unwind on its own frequently, but to integrate CSmith-randomly generated C code within hand-written C snippets that would generate large enough FDEs and nested calls. This was abandoned as well as the call graph of a CSmith-generated code is often far too small, and the CSmith code is notoriously hard to understand and edit.

5 Results

5.1 Hardware used

All the measures in this report were made on a computer with an Intel Xeon E3-1505M v6 CPU, with a clock frequency of 3.00 GHz and 8 cores. The computer has 32 GB of RAM, and care was taken never to fill it and start swapping – using the hard drive to store data instead of the RAM when it is full, degrading harshly the performance.

5.2 Measured time performance

A benchmarking of `eh_elfs` against the vanilla `libunwind` was made using the exact same methodology as in Section 4.3, only linking `perf` against the vanilla `libunwind`. It yields the results in Table 3.

Unwinding method	Frames unwound	Total time unwinding (μs)	Average time per frame (ns)	Unwinding errors	Time ratio
`eh_elfs`	23506	14837	631	1099	1
`libunwind`, cached	27058	441601	16320	885	25.9
`libunwind`, uncached	27058	671292	24809	885	39.3

Table 3: Time benchmarking on hackbench

The performance of `eh_elfs` is probably overestimated for a production-ready version, since `eh_elfs` do not handle all the registers from the original DWARF, lightening the computation. However, this overhead, although impossible to measure without first implementing supports for every register, would probably not be that big, since most of the time is spent finding the relevant row. Support for every DWARF instruction, however, would not slow down at all the implementation, since every instruction would simply be compiled to x86_64 without affecting the already supported code.

The fact that there is a sharp difference between cached and uncached `libunwind` confirm that our experimental setup did not unwind at totally different locations every single time, and thus was not biased in this direction, since caching is still very efficient.

The compilation time of `eh_elfs` is also reasonable. On the machine described in Section 5.1, and without using multiple cores to compile, the various shared objects needed to run `hackbench` – that is, `hackbench`, `libc`, `ld` and `libpthread` – are compiled in an overall time of 25.28 seconds, which a developer is probably prepared to wait for.

The unwinding errors observed are hard to investigate, but are most probably due to truncated stack records. Indeed, since `perf` dumps the last n bytes of the call stack (for a given n), and only keeps those for later unwinding, large stacks leads to lost information when analyzing the results. The difference between `eh_elfs` and the vanilla library could be due either to unsupported DWARF instructions or registers, `libdwarfpp` bugs or bugs in the custom `libunwind` implementation that were not spotted.

5.3 Measured compactness

A first measure of compactness was made for one of the earliest working versions in Table 2. The same data, generated for the latest version of `eh_elfs`, can be seen in Table 4.

The effect of the outlining mentioned in Section 3.4 is particularly visible in this table: `hackbench` has a significantly bigger growth than the other shared objects. This is because `hackbench` has a way smaller `.eh_frame`, thus, the outlined data is reused only a few times, compared to *eg.* `libc`, in which the outlined data is reused a lot.

Just as with time performance, the measured compactness would be impacted by supporting every register, but probably lightly, since the four supported registers represent most columns – see Section 5.4.

Shared object	Original program size	Original `.eh_frame`	Generated `eh_elf .text`	% of original program size	Growth factor
libc-2.27.so	1.4 MiB	130.1 KiB	313.2 KiB	21.88	2.41
libpthread-2.27.so	58.1 KiB	11.6 KiB	25.4 KiB	43.71	2.19
ld-2.27.so	129.6 KiB	9.6 KiB	28.6 KiB	22.09	2.97
hackbench	2.9 KiB	568.0 B	2.8 KiB	93.87	4.99
Total	1.6 MiB	151.8 KiB	370.0 KiB	22.81	2.44

Table 4: `eh_elfs` space usage

5.4 Instructions coverage

In order to determine which DWARF instructions should be implemented to have meaningful results, as well as to assess the instruction coverage of our compiler and `eh_elfs`, we must look at real-world ELF files and inspect the instructions used.

The method chosen was to take a random uniform sample of 4000 ELFs among those present on a basic ArchLinux system setup, in the directories `/bin`, `/lib`, `/usr/bin`, `/usr/lib` and their subdirectories, making sure those files were ELF64 files, then gathering statistics on those files.

	Unsupported register rule	Register rules seen	% supp.	Unsupported expression	Expressions seen	% supp.
Only supp. columns	1603	42959683	99.996 %	1114	5977	81.4 %
All columns	1607	67587841	99.998 %	1154	13869	91.7 %

Table 5: Instructions coverage statistics

	`Undefined`	`Same_value`	`Offset`	`Val_offset`	`Register`
Only supp. columns	1698 (0.006 %)	0	30038255 (99.9 %)	0	14 (0 %)
All columns	1698 (0.003 %)	0	54666405 (99.9 %)	0	22 (0 %)

	`Expression`	`Val_expression`	`Architectural`	Total
Only supp. columns	4475 (0.015 %)	0	0	30044442
All columns	12367 (0.02 %)	0	0	54680492

Table 6: Instruction type statistics

The Table 5 gives statistics about the proportion of instructions encountered that were not supported by `eh_elfs`. The first row is only concerned about the columns CFA, %rip, %rsp, %rbp and %rbx (the supported registers – see Section 3.2). The second row analyzes all the columns that were encountered, no matter whether supported or not in `eh_elfs`.

The Table 6 analyzes the proportion of each command – the formal way a register is set – for non-CFA columns in the sampled data. For a brief explanation, `Offset` means stored at offset from CFA, `Register` means the value from a machine register, `Expression` means stored at the address of an expression's result, and the `Val_` prefix means that the value must not be dereferenced. Overall, it can be seen that supporting `Offset` already means supporting the vast majority of registers. The data gathered (not reproduced here) also suggests that supporting a few common expressions is enough to support most of them. This is further supported by the fact that we already support more than 80 % of expressions only by supporting two basic constructs.

It is also worth noting that among all of the 4000 analyzed files, all the unsupported expressions are clustered in only 12 of them, and only 24 contained unsupported instructions at all.

Conclusion

From this data, we can deduce that

- compilation of the DWARF unwinding data is effective to speed up drastically unwinding procedures: speedup of $\times 25.9$;
- code outlining is effective to reduce the produced binary size: from 1 MiB to 370 KiB, from a growth factor of 7 compared to DWARF unwinding data to a growth factor of 2.45;
- unwinding relies on small subset of DWARF instructions and expressions, while most instructions are not used at all in DWARF code produced by compilers.

The overall size of the project is

- compiler: 1628 lines,
- `libunwind`: 810 lines,
- `perf`: 222 lines

for a total of 2660 lines of code on the main project. The various statistics, benchmarking, testing and analyzing code modules add up to around 1500 more lines.

References

[1] C11. *ISO/IEC 9899:2011*. International Organization for Standardization.

[2] Christophe De Dinechin. "C++ exception handling". In: *IEEE Concurrency* 8.4 (2000), pp. 72–79.

[3] DWARF5. *DWARF Debugging Information Format version 5*. DWARF Debugging Information Format Committee. 2017. URL: http://dwarfstd.org.

[4] Stephen Kell. "Dynamically diagnosing type errors in unsafe code". In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 800–819.

[5] Andrew Koening and Bjarne Stroustrup. "Exception handling for C++". In: *Journal of Object-Oriented Programming* 3.2 (1990), pp. 16–33. URL: http://www.stroustrup.com/except89.pdf.

[6] *Libunwind webpage*. URL: http://www.nongnu.org/libunwind/.

[7] Francesco Nidito. *Exceptions in C with Longjmp and Setjmp*. URL: https://www.di.unipi.it/~nids/docs/longjump_try_trow_catch.html (visited on 08/04/2018).

[8] James Oakley and Sergey Bratus. "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code." In: *WOOT*. 2011, pp. 91–102.

[9] *System V Application Binary Interface, AMD64 architecture*. URL: https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.

[10] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 2011, pp. 283–294. DOI: 10.1145/1993498.1993532. URL: http://doi.acm.org/10.1145/1993498.1993532.

[11] Yanmin Zhang. *Hackbench*. 2008. URL: https://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.