View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Kent Academic Repository

Kent Academic Repository

Full text document (pdf)

Citation for published version

Bastian, Théophile and Kell, Stephen and Zappa Nardelli, Francesco (2019) Reliable and Fast DWARF-Based Stack Unwinding. Proceedings of the ACM on Programming Languages, 3 (OOPSLA). (In press)

DOI

Link to record in KAR

https://kar.kent.ac.uk/76575/

Document Version

Publisher pdf

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

Reliable and Fast DWARF-Based Stack Unwinding

THÉOPHILE BASTIAN, ENS, France
STEPHEN KELL, University of Kent, UK
FRANCESCO ZAPPA NARDELLI, Inria, France

Debug information, usually encoded in the DWARF format, is a hidden and obscure component of our computing infrastructure. Debug information is obviously used by debuggers, but it also plays a key role in program analysis tools, and, most surprisingly, it can be relied upon by the runtime of high-level programming languages. For instance the C++ runtime leverages DWARF stack unwind tables to implement exceptions! Alas, generating debug information adds significant burden to compiler implementations, and the debug information itself can be pervaded by subtle bugs, making the whole infrastructure unreliable. Additionally, interpreting the debug tables is a time-consuming task and, for some applications as sampling profilers, it turns out to be a performance bottleneck.

In this paper we focus on the DWARF `.eh_frame` table, that enables stack unwinding in absence of frame-pointers. We will describe two techniques to perform *validation* and *synthesis* of the DWARF stack unwinding tables, and their implementation for the x86_64 architecture. The validation tool has proven effective for compiler and inline assembly testing, while the synthesis tool can generate DWARF unwind tables for arbitrary binaries lacking debug information. Additionally, we report on a technique to precompile unwind tables into native x86_64 code, which we have implemented and integrated into `libunwind`, resulting in 11x-25x DWARF-based unwind speedups.

CCS Concepts: • **Software and its engineering → Compilers**; *Software testing and debugging*.

Additional Key Words and Phrases: debugging, stack unwinding, DWARF

ACM Reference Format:
Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. 2019. Reliable and Fast DWARF-Based Stack Unwinding. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 146 (October 2019), 24 pages. https://doi.org/10.1145/3360572

1 DWARF-BASED STACK UNWINDING

Upon encountering a segmentation fault error, most of us take for granted that a debugger can reliably print a backtrace and navigate the stack frames:

```
Program received signal SIGSEGV.
  0x54625 in fct_b at segfault.c:5   5 printf("%l\n", *b);
(gdb) backtrace
  #0 0x54625 in fct_b at segfault.c:5
  #1 0x54663 in fct_a at segfault.c:10
  #2 0x54674 in main at segfault.c:14
(gdb) frame 1
  #1 0x54663 in fct_a at segfault.c:10 10 fct_b((int*) a);
```

If the binary has been compiled with a frame pointer register, then identifying where the return address has been stored and recursively walking the stack is simple: for example, the x86_64 ABI specifies that the frame pointer register is `rbp`, that the return address of the current frame is stored at the address pointed by `rbp+8`, and that the base pointer of the previous frame is stored at the

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART146
https://doi.org/10.1145/3360572

LOC	CFA	rbx	rbp	r12	r13	r14	r15	ra
0084950	rsp+8	u	u	u	u	u	u	c-8
0084952	rsp+16	u	u	u	u	u	c-16	c-8
0084954	rsp+24	u	u	u	u	c-24	c-16	c-8
0084956	rsp+32	c-32	u	u	u	c-24	c-16	c-8
0084958	rsp+40	c-32	c-40	u	u	c-24	c-16	c-8
0084959	rsp+144	c-32	c-40	u	u	c-24	c-16	c-8
0084962	rsp+40	c-32	c-40	u	u	c-24	c-16	c-8
0084964	rsp+32	c-32	c-40	u	u	c-24	c-16	c-8
0084966	rsp+24	c-32	u	u	u	c-24	c-16	c-8

Fig. 1. An interpreted `.eh_frame` table (excerpt)

address pointed by `rbp`. In this paper we focus on the x86_64 architecture, but similar conventions hold for other architectures. If instead, as suggested by the x86_64 ABI, the code has been compiled without a frame pointer register, then the only information available to the the debugger is stack pointer register `rsp`, but this is useless if the current stack frame size is not known as well.

DWARF debug information [DWARF 2017] comes to the rescue. In general, DWARF debug information attempts to communicate an accurate picture of the source program to debuggers and program analysis tools. Separate tables cover different debugger capabilities; among them, the `.debug_line` table maps object code addresses to source code locations, the `.debug_info` table maps source variables to the registers or stack locations where they are stored, while the `.eh_frame` [1] table provides the information necessary to unwind the stack.

Conceptually `.eh_frame` is a large table, that for each machine instruction of the executable specifies how to compute where the return address and callee-saved registers have been stored. A typical excerpt, for a function beginning at address `0x84950`, is in Figure 1. The rows of the table correspond to machine instructions in the program text, and the columns correspond to registers (columns `rbx` to `r15`) and to the return address (column `ra`). Each cell holds a rule detailing how the contents of the register, or return address, will be restored for the previous stack frame. Arbitrary complex expressions reading all the machine state are allowed, but most often these are expressed in terms of registers, constants, and of the *canonical frame address*, shortened `CFA` or just `c`: this is a stack location invariant during the execution of the function (conventionally the value of the register `rsp` immediately after the execution of the call instruction). Upon entering the function the return address has just been pushed on the stack: the table confirms that at IP `0x84950`, the return address is stored at `CFA-8`, that is at the address stored in `rsp`, as expected. Callee-saved registers have not been pushed to the stack yet, hence the undefined entries that complete the row. Each row covers a range of machine instruction, from its address (`LOC`) to that of the row below. For instance, at IP `0x0084a00`, according to the table the return address will be found at address `rsp+64-8`.

Although conceptually simple, this table, if constructed in its entirety, would be extremely large, more than the program text itself. The DWARF standard thus encodes the `.eh_frame` table using a compact bytecode; the bytecode is then interpreted on demand to build the unwind table. The first few lines of the table above are for instance computed by the following bytecode:

```
00009b30 48 009b34 FDE cie=0000 pc=0084950..0084b37
  DW_CFA_advance_loc: 2 to 0000000000084952
  DW_CFA_def_cfa_offset: 16
  DW_CFA_offset: r15 (r15) at cfa-16
  DW_CFA_advance_loc: 2 to 0000000000084954
  DW_CFA_def_cfa_offset: 24
```

[1] The `.eh_frame` table is generated by default by mainstream compilers, but in some cases the `.debug_frame` table is used instead. This differs form `.eh_frame` only for minor details of the bytecode encoding, and can be ignored for our purposes.

```
DW_CFA_offset: r14 (r14) at cfa-24
DW_CFA_advance_loc: 2 to 0000000000084956
DW_CFA_def_cfa_offset: 32
DW_CFA_offset: r13 (r13) at cfa-32
DW_CFA_advance_loc: 2 to 0000000000084958
DW_CFA_def_cfa_offset: 40
DW_CFA_offset: r12 (r12) at cfa-40
DW_CFA_advance_loc: 1 to 0000000000084959
```

The bytecode defines the rows of the table by expressing the delta with the previous line. The `advance_loc` command provides the new location, and the new row is defined as a clone of the previous one, which can then be altered, for instance by setting `CFA` to `rsp+48`. This process of defining every line with respect to the previous one is compact, but implies that to get the row for a given address, all the bytecode from the beginning of the function to that address must be evaluated, potentially a very slow process.

In the example expressions are simply computing offsets from `rsp`, but DWARF was designed to be as flexible as possible, and expressions can be arbitrarily complex. For instance, the `.eh_frame` table associated with the PLT sections routinely uses this entry:

```
DW_CFA_def_cfa_expression (DW_OP_breg7 (rsp): 8; DW_OP_breg16 (rip): 0; DW_OP_lit15;
  DW_OP_and; DW_OP_lit11; DW_OP_ge; DW_OP_lit3; DW_OP_shl; DW_OP_plus)
```

Each instruction is executed by a stack-based interpreter, and without dwelling on the details, this code is computing the expression:

```
rsp + 8 + ((((rip & 15)>= 11)? 1 : 0)<< 3)
```

a clever hack that relies on the `rip` register and the regular structure of PLT tables to concisely encode the unwind information for all the PLT entries with only one unwind expression. In general the bytecode has both conditional and conditional jump expressions and can read arbitrary memory locations. In one of the few academic articles addressing DWARF, Oakley and Bratus [2011] not only argue that the bytecode is Turing complete, but they show how a malicious attacker, by injecting crafted DWARF unwind tables into a binary, can create undetected trojans.

The current state. Being able to generate backtraces and, more generally, to walk the stack, is an operation that is relied upon not only by debuggers to print stack traces, but also by program analysis tools (e.g. by sampling-based profilers) and, most surprisingly, by the runtime of some high-level programming languages (e.g. to implement exceptions, continuations, global gotos, or asynchronous garbage collectors). It is an essential component of our computational infrastructure. However DWARF based unwinding has three major drawbacks: it is *unreliable*, *slow*, and *not always available*.

Unreliable Generating the DWARF tables tends to be a burden for compiler authors, as each optimisation pass potentially invalidates them; keeping tables and code synchronised pass after pass requires a tedious and error prone logic to be added to the already convoluted optimiser passes. In practice, there are bugs in the generated tables and these are hard to detect because DWARF information generation lacks rigorous and complete test suites.

Unavailable Many important compilers do not generate DWARF tables at all; this is a major issue for JIT compilers as HotSpot (OpenJDK) and prototype implementations of new languages, but also concerns C and C++ compilers for complex optimisation passes (e.g. link-time optimisation) or inline-assembly snippets. This omission prevents debugging with standard tools and, more annoyingly, implies that hacks and workarounds are required to use a wide range of useful program analysis instruments. It also implies that the programmer must manually encode the DWARF bytecode in the inline-assembly snippets.

Slow Accessing a row of the table requires interpreting all the bytecode from the beginning of the enclosing function, and this must be repeated for each stack-frame on the call stack. Despite the aggressive caching performed by libraries as `libunwind`, DWARF-based unwinding can be a bottleneck for time-sensitive program analysis tools. For instance the `perf` profiler is forced to copy the whole stack on taking each sample and to build the backtraces offline: this solution has a memory and time overhead but also serious confidentiality and security flaws.

We are not the only ones to raise these concerns. We have been told that all critical code at Google is compiled with a frame pointer, to ensure fast and reliable backtraces [Engineer 2018]. Similarly, the Linux kernel by default relies on a frame pointer to provide reliable backtraces. This incurs in a space and time overhead; for instance it has been reported (https://lwn.net/Articles/727553/) that the kernel's `.text` size increases by about 3.2%, resulting in a broad kernel-wide slowdown. Measurements have shown a slowdown of 5-10% for some workloads (https://lore.kernel.org/lkml/20170602104048.jkkzssljsompjdwy@suse.de/T/#u).

Linus Torvalds uses colourful words to summarise why DWARF-based unwinding is unreliable:

> *Sorry, but last time was too f***ing painful. The whole (and *only*) point of unwinders is to make debugging easy when a bug occurs. But the f***ing dwarf unwinder had bugs itself, or our dwarf information had bugs, and in either case it actually turned several "trivial" bugs into a total undebuggable hell.*

and concludes:

> *If you can mathematically prove that the unwinder is correct — even in the presence of bogus and actively incorrect unwinding information — and never ever follows a bad pointer, I'll reconsider.*
>
> Linux Kernel mailing list, 2012. https://lkml.org/lkml/2012/2/10/356

Torvalds's concern is the starting point of this paper: we have designed and implemented three techniques to improve the reliability and efficiency of DWARF-based unwinding.

Contributions. Our *first contribution* is the design and implementation of a tool that *cross-checks binaries against their DWARF unwind tables.* The tool dynamically validates the DWARF informations, identifying hard-to-detected bugs in the `.eh_frame` tables, and preventing DWARF`.eh_frame` injection attacks. As we shall see this is feasible because the `.eh_frame` stack description tables do not depend on source code semantics, but capture instead an abstract symbolic execution of the binary machine instructions.

Once we can cross-checks binaries against their DWARF unwind tables, it is only a small extra step to *synthesise DWARF unwind tables from binaries.* Our *second contribution* is a technique to synthesise DWARF unwind information by automatic analysis of ELF binaries lacking unwind information. This is implemented and evaluated in a second tool.

Finally, our *third contribution* is to investigate various strategies to precompile the `.eh_frame` table into assembly code, and to have extended `libunwind` to support the precompiled tables. We have observed 25x speedups on DWARF-unwind intensive applications, at the cost of a 2.5x increase of the size of unwind tables on disk.

The first two contributions are presented in Section 2, after a precise description of the close relationship between assembly code and unwind tables. The third contribution is presented in Section 3. Related and future works, including ongoing discussion with developers of mainstream tools, conclude the paper.

All along the paper we focus on the columns needed to walk the stack and build a backtrace: this is the most common operation that involves `.eh_frame` tables. This allows us to keep the presentation compact and accessible, by ignoring some of the callee-saved register columns. Our techniques can

```
 0 <foo>:                                CFA        rbp     ra
 1   push   %r15                          rsp+8      u       c-8
 2   push   %r14                          rsp+16     u       c-8
 3   mov    $0x3,%eax                     rsp+24     u       c-8
 4   push   %rbx                          rsp+24     u       c-8
 5   push   %rbp                          rsp+32     c-40    c-8
 6   sub    $0x68,%rsp                    rsp+40     c-40    c-8
 7   cmp    $0x1,%edi                     rsp+144    c-40    c-8
 8   movl   $0x0,0x14(%rsp)               rsp+144    c-40    c-8
 9   je     .L2                           rsp+144    c-40    c-8
10   add    $0x68,%rsp                    rsp+40     c-40    c-8
11   pop    %rbp                          rsp+32     c-40    c-8
12   pop    %rbx                          rsp+24     u       c-8
```

Fig. 2. Comparing assembly and unwind tables (excerpt)

be extended to support all callee-saved registers. Overall, our contributions show that relatively simple techniques are effective in improving DWARF-based unwinding reliability and efficiency, hopefully paving the way for their adoption by mainstream tools.

Source code. The source code for all the tools described in this paper is available from https://www.di.ens.fr/~zappa/projects/frdwarf. Our tools and test cases have been evaluated as *Functional* and *Available* during artifact evaluation.

2 VALIDATION AND SYNTHESIS OF DWARF-UNWIND TABLES

To understand the close relation between assembly code and the associated unwind tables, we report in Figure 2 the assembly code for the unwind table of Figure 1, and the table itself. Entries have been aligned so that the table describe the stack state *before* executing the corresponding instruction. To make the presentation compact we focus on the `CFA`, `rbp` and `ra` columns, and omit the other callee-saved register columns. Their treatment is analogous to that of the `rbp` column.

Upon invoking the function `foo`, before its first instruction is executed, the semantics of the `call` instruction ensures that the address where the return address is stored is in `rsp`. The table reflects this by setting the `CFA` at `rsp+8` and the return address at `c-8`. Next, the first `push` instruction updates the stack and decrements the value of `rsp` by `8`: the table updates the `CFA` computation to `rsp+16`, reflecting the new value of the stack pointer. Similarly for the other `push` or `pop` instructions and the `sub $0x68, %rsp` instruction at line 6, which allocates space on the stack: all these modify the stack pointer and the `CFA` computation is updated accordingly. Instructions that do not modify the stack pointer leave entries unchanged in the unwind table.

In this example all the expressions to compute the `CFA` are defined in term of the stack pointer `rsp`, so tracking instructions that modify the stack pointer is enough. In general, DWARF expressions can involve arbitrary registers and can read arbitrary memory locations, so a complete host instruction set semantics might be needed. In Section 3.2 we will investigate the real-world uses of DWARF instructions but for now, the key remark we make is that all the information stored in the unwind table can be recomputed from the semantics of the assembly code and the calling conventions.

In other words, the unwind table captures an *abstract, symbolic execution* of the assembler:

- the *abstract state* is represented by the addresses where the return address and callee-saved registers have been stored by each call-frame, expressed in terms of symbolic registers or memory locations;

```
short a,b,g;  long c;
char d;  int e, f;

void main() {
  for(; f; f++)
    for(; e; e++)
      for(; c; c++) {
        g = a%b;
        for(; d <= 1; d++);
      }
}
```

```
0 <main>:          CFA      ra
1   push %rbx      rsp+8    c-8
2   ...            rsp+16   c-8
3   ...            ...
4   pop %rbx       rsp+16   c-8
5   retq           rsp+16   c-8
```

Fig. 3. An incorrect `.eh_frame` table generated by `clang`

- the *abstract semantics* of the `call` (or `return`) instruction pushes (or pops) a call-frame on the abstract state; the abstract semantics of all other instructions traces the effects of the instruction semantics on the expressions used by top-level call-frame.

From this point of view, the unwind table is *redundant* with respect to the program text. This paves the way to validate the unwind tables against the program text, or to synthesise them from the program text.

2.1 Validation of `.eh_frame` Tables

The observation that the unwind table is redundant with respect to the program text suggests a natural approach to dynamically validating its correctness along one execution path, which we explore in this section.

It is indeed possible to execute a program one instruction at a time, tracking where the return address and callee-saved registers have been stored on the stack. At the same time, before each instruction, the `.eh_frame` row for the instruction can be evaluated on the current machine state, and the values computed from the unwind table can be compared with the concrete address being tracked. If a mismatch is detected for any of the columns, an error is reported.

As an example, let us consider the C program on the left in Figure 3. An excerpt of the assembly code and `.eh_frame` table generated by the `clang` compiler (version 3.8) is shown on the right. Suppose that we have access to the machine state and suppose that before executing the first instruction the stack pointer points to address `0xFFFF1000`, and in turn that the return address has been stored at `0xFFFF1000`. We can verify that the return address computation in the `.eh_frame` table is correct: `0xFFFF1000+8-8` is equal to `0xFFFF1000`. Similarly, after executing the first instruction, we access the machine state and get that `rsp` stores `0xFFFF0FF8`: the table is again correct because `0xFFFF0FF8+16-8` is `0xFFFF1000`.

Consider now the unwind entry for the `retq` instruction. The machine state confirms that after the previous `pop` instruction, the register `rsp` stores `0xFFFF1000`. However the table row for this instruction is *incorrect*: `0xFFFF1000+16-8` is different from the address where the return address is stored, namely `0xFFFF1000`.

We track the stack addresses where the return addresses have been saved by keeping a stack of all the return address addresses, called *shadow stack*. Whenever we observe a `call` instruction being executed, we push the current value of the stack pointer on the shadow stack. Similarly, upon execution of a `ret` instruction, one entry is popped (and discarded) from the shadow stack.

Implementation. This strategy requires being able to execute an assembly program one step at a time, while accessing the machine state. There are several alternatives for this task, of varying robustness and efficiency. We have opted to build upon the `gdb` debugger: the debugger gives us

Algorithm 1 Pseudocode for the validation algorithm

```
function MAIN()
    abstract_state = {}
    eh_frame_table = parse_eh_table(file)
    gdb_advance_to_main(file)
    while True do
        (ip, instr) = gdb_get_current_instruction()          ▹ validation of eh_table at ip
        regs = gdb_get_registers()
        eh_entry = get_eh(eh_frame_table, ip)
        if abstract_state.top() != compute_ra(eh_entry, regs) then
            error('Inconsistent table at '+ip)
        if instr == 'call' then                              ▹ update of abstract state
            abstract_state.push(regs['rsp'] - 8);
        else if instr == 'ret' then
            abstract_state.pop()
            if empty(abstract_state) then
                exit('Validation succesful')
```

access to the machine state, can execute a binary one assembly instruction at a time, and can be scripted using Python. Our tool thus is implemented as a Python script that implements literally the dynamic analysis described above, sketched in Algorithm 1. For ELF and DWARF parsing we rely on the `pyelftools` library [Bendersky 2019] to which we contributed a parser for the `.eh_frame` table, since only the rarely-used `.debug_frame` table was initially supported. This approach is simple and robust, and can validate arbitrarily complex control flow or DWARF unwind tables. Additionally it is easily portable to multiple architectures: in addition to supporting the x86_64 architecture and calling conventions, we have experimental support for the IBM Power architecture. The tool can process about 3000 assembly instructions per second, measured by dividing the overall running time by the number of assembly instructions executed, easily obtained via `gdb`. Validation speed is independent of the program being verified, the overhead coming mostly from `gdb` step-by-step execution and `pyelftools` decoding of `.eh_frame` tables. Although the overhead is important, our implementation is fast enough for batch testing as done by C fuzzers or library test-suites.

At work. We have put our validation tool at work to test both the binaries generated by C mainstream compilers and the binary of the `glibc` library.

By leveraging the CSmith fuzzer [Yang et al. 2011] and CReduce test-case reducer [Regehr et al. 2012], we have performed random testing of the `.eh_frame` table generation of the `gcc` and `clang` C compilers. The program we reported in Figure 3 was indeed found with fuzzing: the compiler forgot to insert a `.cfi_*` directive to reflect the update to the stack pointer due to restoring register `rbx`. If an interrupt occurs between the `pop` and `ret` instructions, the unwind information might be wrong, despite using `-fasynchronous-unwind-tables` which is supposed to be exact at every instruction. We have filed a report to `clang` developers (https://bugs.llvm.org/show_bug.cgi?id=13161#c2); the last version of the compiler does not miscompile the `.eh_frame` table anymore, suggesting that the issue has been resolved.

Our previous example contradicts the common belief that the unwinder does not fail on C/C++ compiled code. Hand-written assembly code raises extra challenges. We focus on the `glibc` library, that often relies on inline assembly and comes with an extensive test suite. To illustrate the complexity of unwinding correctly hand-written assembly we report on the excerpt in Figure 4 of

```
#define LLL_STUB_UNWIND_INFO_START \
        ".section       .eh_frame,\"a\",@progbits\n"        \
"5:\t"  ".long  7f-6f   # Length of CIE\n"                  \
"6:\t"  ".long  0x0     # CIE Identifier Tag\n\t"           \
        ".byte  0x1     # CIE Version\n\t"                  \
        ".ascii \"zR\\0\"       # CIE Augmentation\n\t"     \
        ".uleb128 0x1   # CIE Code Alignment Factor\n\t"    \
        ".sleb128 -4    # CIE Data Alignment Factor\n\t"    \
        ".byte  0x8     # CIE RA Column\n\t"                \
        ".uleb128 0x1   # Augmentation size\n\t"            \
        ".byte  0x1b    # FDE Encoding (pcrel sdata4)\n\t"  \
        ".byte  0xc     # DW_CFA_def_cfa\n\t"               \
        ".uleb128 0x4\n\t"                                  \
        ".uleb128 0x0\n\t"                                  \
        ".align 4\n"                                        \
"7:\t"  ".long  17f-8f  # FDE Length\n"                     \
"8:\t"  ".long  8b-5b   # FDE CIE offset\n\t"               \
        ".long  1b-.    # FDE initial location\n\t"         \
        ".long  4b-1b   # FDE address range\n\t"            \
        ".uleb128 0x0   # Augmentation size\n\t"            \
        ".byte  0x16    # DW_CFA_val_expression\n\t"        \
        ".uleb128 0x8\n\t"                                  \
        ".uleb128 10f-9f\n"                                 \
"9:\t"  ".byte  0x78    # DW_OP_breg8\n\t"                  \
        ".sleb128 3b-1b\n"
```

```
/* Unwind info for
   1: lea ..., ...
   0: movl ..., ...
   2: call ...
   3: jmp 18f
   4:
   snippet.  */
#define LLL_STUB_UNWIND_INFO_4 \
LLL_STUB_UNWIND_INFO_START                                          \
"10:\t" ".byte  0x40 + (0b-1b) # DW_CFA_advance_loc\n\t"  \
        ".byte  0x16    # DW_CFA_val_expression\n\t"      \
        ".uleb128 0x8\n\t"                                \
        ".uleb128 20f-19f\n"                              \
"19:\t" ".byte  0x78    # DW_OP_breg8\n\t"                \
        ".sleb128 3b-0b\n"                                \
"20:\t" ".byte  0x40 + (2b-0b) # DW_CFA_advance_loc\n\t"  \
LLL_STUB_UNWIND_INFO_END

#define LLL_STUB_UNWIND_INFO_END \
        ".byte  0x16    # DW_CFA_val_expression\n\t"      \
        ".uleb128 0x8\n\t"                                \
        ".uleb128 12f-11f\n"                              \
        ...
```

Fig. 4. Hand-written `.eh_frame` bytecode in `glibc`'s `lowlevellock.h`

the `lowlevellock.h` file. This file is part of the implementation of futexes (fast user-space mutexes), and the low-level thread synchronisation is implemented in inline assembly. The figure reports the intricate, *hand-written*, DWARF `.eh_frame` bytecode corresponding to the low-level lock functions. Despite great care by the authors, we identified an off-by-one error in one of the return-address offsets; this error causes, for instance, `gdb` to duplicate one `_L_lock_19` call in the backtrace:

```
Breakpoint 2, 0x0000000000406c2c in _L_lock_19 ()
(gdb) bt
  #0 0x0000000000406c2c in _L_lock_19 ()
  #1 0x0000000000406c3f in _L_lock_19 ()
  #2 0x00000000004069c6 in abort ()
  #3 0x0000000000401017 in main () at foo1.c:4
```

Most other program analysis tools, rather than a relatively harmless duplicate frame, would crash upon following a bad pointer in an unwind table.

2.2 Synthesis of `.eh_frame` Tables

In an ideal world, no programmer would have to hand-write DWARF unwind bytecode: this should be inferred by the compiler for the inline assembly snippets, as it is for the rest of the C code. This is the starting point for this section: we will design and implement a tool that synthesises DWARF unwind tables directly from the assembly program text.

We have seen that there is a close relationship between assembly instructions and updates to the lines of the unwind table. We also observe that the unwind data itself is structured in a collection of Frame Description Entries (FDEs), each FDE providing the unwind information for a given function in the original source code: ABI calling conventions ensure that unwind tables are compositional. Therefore our synthesis algorithm only needs to analyse one function body at a time.

Let us initially consider a function for which the compiler *can statically compute* the stack size at any instruction. In this case, before each assembly instruction, the `CFA` can always be expressed as a static offset from the stack pointer register. Additionally, suppose that we can reliably build the control flow graph for the assembly code of the function.

Let us consider the running example in Figure 5, and focus on the entry basic block. Calling conventions ensure that before the first instruction the address of the return address is stored at `rsp`. Therefore we can synthesise the standard `CFA = rsp+8; ra = c-8` entry. We can then analyse all the following instructions in the basic block, one at a time, checking if they modify the stack pointer. If

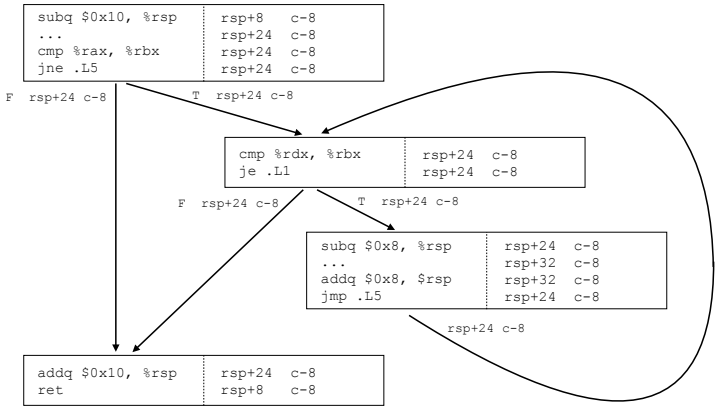

Fig. 5. Relationship between CFG and unwind tables

```
int main() {
  int z_max = read_integer();

  for(int z=0; z < z_max; z++) {
    int x[z];
    ...do something with x...
  }
}
```

LOC	CFA	rbp	ra
00400526	rsp+8	u	c-8
00400527	rsp+16	c-16	c-8
0040052a	rbp+16	c-16	c-8
00400537	rbp+16	c-16	c-8
004005cf	rsp+8	c-16	c-8

Fig. 6. Dynamically stack-allocated data-structures

they do, we update the `CFA` offset accordingly. In the running example, the `CFA` is updated from `rsp+8` to `rsp+24` after the stack allocation performed by the first instruction of the basic block, and is left unchanged afterward. Having reached the end of the basic block, we perform a *forward analysis* and we propagate the unwind row after the last instruction to all the following basic blocks, following the control flow arrows. A key property is that when propagating the unwind entry along control flow graph arrows there is *no risk of conflict* with previously propagated entries. To understand why, let us consider the loop in our running example. Since we have assumed that at any moment the compiler must be able to reference stack-allocated variables with statically computed offsets from `rsp`, the assembly must have been generated so that these offsets are *invariant at merge points* of the control flow graph. In turn, no conflict can arise when merging the propagated unwind lines, and in presence of loops computing a fixpoint is not necessary. The forward analysis iterates the process of computing the unwind information for a basic block, and propagating the last entry to the successor blocks, until unwind information is synthesised for all the basic blocks.

Suppose now that semantics of the code implies that stack offsets *cannot be statically computed*. This happens in presence of variable-size data structures dynamically allocated on the stack, as in the example reported in Figure 6. In this case it turns out that compilers systematically fall back to using a base-pointer register even if the `-fomit-frame-pointer` option has been specified, as revealed by a quick glance to the unwind table (or to the assembly).

The good consequence is that unwind rows remain invariant at merge points even when the size of the stack cannot be statically determined by the compiler. However the synthesis algorithm

must thus distinguish when the register `rbp` is used as base-register from the cases where `rbp` is used as a general purpose register.

A similar difficulty arises for synthesising the columns for callee-saved registers: the analysis must detect which instructions save and restore each callee-saved register on the stack. Since synthesised unwind rows may refer to the callee-saved register `rbp`, we have implemented synthesis of unwind information for the `rbp` register.

Implementation. We have implemented our synthesis tool on top of BAP, the binary analysis platform developed at CMU [Brumley et al. 2011]. BAP decompiles a binary program into a RISC-like instruction language, and provides both an API to manipulate the intermediate representation and a set of tools to ease the construction of program analysis tools.

Algorithm 2 Pseudocode for the synthesis algorithm

```
function MAIN()
    eh_frame = {}; visited_blks = {}
    last_row = 'cfa = rsp+8; rbp = u; ra = c-8'
    for all functions f in the binary do
        entry_blk, cfg = BAP_build_cfg(f)                          ▷ build the CFG for f
        success = DFS(entry_blk, cfg, last_row, {}, eh_frame, 'rsp-mode')  ▷ rsp-mode synthesis
        if not success then                                        ▷ fallback: rbp-mode synthesis
            eh_frame = {}; visited_blks = {}
            success = DFS(entry_blk, cfg, last_row, eh_frame, 'rbp-mode')
            if not success then
                error('synthesis failed')
    return eh_frame

function DFS(blk, cfg, last_row, eh_frame, mode)
    if not consistent(last_row, eh_frame(first_ip(blk))) then
        error('inconsistency at '+first_ip(blk))
    if not in(blk, visited_blks) then
        for all ip in blk do                                       ▷ synthesise eh_frame for blk
            instr = instruction_at(ip, blk)
            eh_frame(ip) = last_row
            success, last_row = synthesize_row (ip, instr, last_row, mode)   ▷ see Heuristics
            if not success then
                return false
        add(blk, visited_blk)
        for all s_blk in successors(blk, cfg) do                   ▷ continue DFS visit of the CFG
            success = DFS(s_blk, cfg, last_row, eh_frame, mode)
            if not success then
                return false
    return true
```

The pseudo-code for the synthesis algorithm is shown in Algorithm 2. The behaviour of the function *synthesize_row* is described in the *Euristic* section below. Given a binary we rely on BAP to decompile it and build the control-flow graph of each function. Then, to process a function, we perform the forward analysis described above and synthesize the table for the function. As a sanity

check, if different rows are propagated to a merge point the tool aborts and reports an error. Once the table is synthesised, it is encoded using simple DWARF bytecode instructions, analogous to those used by `gcc` or `clang`.

Heuristics. The first key choice when synthesising the unwind row for a given IP is guessing if the register `rbp` is being used as base pointer or as general purpose register. Rather than attempting to identify instruction patterns, we use a two-pass approach. Our algorithm first attempts to synthesise a table indexing the `CFA` exclusively with the register `rsp`. We call this state `rsp`-index mode. If synthesis in `rsp`-index mode fails, then the process is started again, allowing both `rsp` and `rbp` registers to be used index the `CFA`. In practice this method is successful because if the first pass succeeds, then a correct `rsp`-based `CFA` indexing was found; if not, then the source code must contain variable-size stack-allocated data structures, and as we have seen the compiler itself opted to use `rbp` as a base pointer.

In detail, when `CFA` is expressed as an offset of the `rsp` register (both in `rsp`-indexing and `rbp`-indexing mode), then it must be updated whenever an instruction modifies the `rsp` register. Such instructions are represented in the BAP internal representation as `rsp <- EXPR`. We perform a simple syntactic analysis of `EXPR`: if `EXPR` is of the form `rsp + offset`, then the `CFA` is updated with the specified offset. Otherwise, the analysis is said to *lose track* and it is restarted in `rbp`-index mode if it was in `rsp`-index mode, and aborted otherwise.

When the algorithm is in `rbp`-index mode and CFA is set as an offset of `rbp`, by definition the assembly code is relying on a stable frame pointer in `rbp`, then the offset should not affected by assembly instructions. However, in `rbp`-index mode, the synthesis must detect when the CFA indexing switches from register `rsp` to register `rbp` and vice-versa. The easy part is to switch indexing from `rsp` to `rbp` whenever `rsp` is saved into `rbp` (which includes the usual instruction `mov %rsp, %rbp`). This is reverted when the content of `rbp` is overwritten. Although it is non-trivial to decide which offset should be used when switching back to register `rsp`, in our experiments this occurs only in the epilogue of functions. We thus assume that upon restore, `CFA = rsp+8`.

In addition to CFA, the synthesis algorithm must update the callee-saved register columns, which we have implemented only for the `rbp` register. This is done by tracking the program points where:

- `rbp` is undefined and an instruction saves `rbp` to the stack,
- `rbp` is defined and an instruction overwrites `rbp` with the data initially saved on the stack. For this, we perform a preliminary backward analysis to identify the set of all the IR terms that correspond to reading or popping the callee-saved register `rbp` from the stack.

One last remark: at merge points of the CFG, all the rows propagated by incoming edges must be merged into a *consistent* row at each merge point. In a first approximation *consistent* can be defined as *equal*; this works perfectly on all our `gcc` experiments. However the consistency relation can be safely weakened by allowing rows with `rbp` undefined on one side and defined on the other to be merged. In this case the merge results in an undefined `rbp` entry — allowing a information loss. Although this implies that in the general case a fixpoint computation is necessary, we have observed such *lossy* merge only in rare cases of programs compiled with `clang`, and only for the exit block of the analysed function. We thus enable the lossy merge only for terminal blocks, just before the `retq` instruction, and our implementation still computes the fixpoint in one pass.

Evaluation. Apart from some hand-written programs designed to test particular features, most of the testing was done using random C programs generated with Csmith (with default options), compiled with either `gcc` (version 8.2.1) or `clang` (version 7.0.1), each with either `-O0 -fomit-frame-pointer`, `-O1` or `-O2`, yielding a total of six testing setups. For each setup, 100 random programs were generated and compiled, then their `.eh_frame` and `.eh_frame_hdr` sections were removed. We used

our tool to synthesise unwinding data for these stripped binaries. An automated tool took care of comparing the compiler-generated and synthesised unwinding data, reporting the cases in which the data were not equivalent.

In this setup, all three levels of optimisation of `gcc` passed all the 300 tests. For `clang`, 295 tests passed while 5 tests required manual investigation. One binary compiled at `-O0` has dead code inside a function. Our algorithm cannot synthesise unwind tables for dead code because the forward dataflow analysis cannot guess initial values. This is not an issue as unwinding will never be called from a location in dead code. In one case, compiled with `-O1`, `clang` generates code for which no correct `.eh_frame` entry exists while handling abort paths. This is arguably a bug in `clang` linearisation algorithm and we will report to clang developers. The two observed failures at `-O2` and the other failure at `-O1` are instead cases in which synthesis is performed correctly, but `clang` generates incorrect tables. These will be reported to the `clang` developers.

Hand-written tests checked that our tool synthesises correct tables for C code including inline assembly that artificially move the stack pointer or the base pointer in a consistent way – that is, such that there exists a valid table – compiled with `gcc`.

Finally we have tested synthesis on larger binaries. We considered a statically linked `gzip` application, compiled with `gcc` and default configure options; the overall binary size is 1.2 MB, and the `.text` section is 698 KB. Statically linking has the effect of including portions of the C library in the binary, and `glibc` heavily uses inline assembly. The synthesis tool generates a correct unwind table, except for the function `x2nrealloc`. This is due to a bug in BAP: a function call to this function never returns, but BAP control-flow graph construction adds a fall-through branch to some other address. As a result, a merge error is generated where no merge should actually happen. This bug has been reported to BAP developers. Similarly, synthesis is performed correctly on a `sqlite3` binary compiled with `gcc` and default configure options.

On a 3.1 GHz MacBook Pro with 16GB of RAM, synthesising the `.eh_frame` table for the `gzip` binary takes 43.5 (± 1) seconds (average on 10 runs), of which 39.5 (± 1) are needed by BAP to build the whole-program control flow graph, and 3.8 (± 0.1) seconds are actually required by our tool to synthesise the table.

3 SPEEDING-UP DWARF-BASED STACK UNWINDING

The design of DWARF `.eh_frame` tables aims at minimising the space occupied by the tables on disk, at the price of efficiency. Stack-unwinding is often thought of as a debugging procedure: when something behaves unexpectedly, the programmer opens a debugger and explores the stack. In this scenario, the time spent unwinding the stack is irrelevant. Yet stack unwinding can be performance-critical: think of sampling-based profilers repeatedly performing stack unwinding to reconstruct the call graph of a program, or C++ exception handling walking the stack to call finalizers for stack-allocated objects and finding suitable catch-blocks. For such applications, it might be desirable to find a different time/space trade-off, storing more information on disk but enabling faster unwinding.

In this section we explore a strategy to precompile the `.eh_frame` section of a binary to executable code directly responsible for unwinding one stack frame. Rather than generating directly assembly code, we actually translate DWARF bytecode to C code later compiled by `gcc` in `-O2` mode, to benefit from `gcc` optimisations. We show one example in Figure 7.

The generated code consists of a single function, called `_eh_elf`, taking as arguments the current instruction pointer and the current memory context describing the values stored in processor registers. The `_eh_elf` function then returns a new memory context containing the values the registers after unwinding the current frame. Unwind contexts are defined in Listing 4: the values of processor registers are stored with type `uintptr_t`, while `flags` is a 8-bits value, indicating for each

```
[...] 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 1. DWARF bytecode

```
[...] FDE [...]  pc=615..65a
   LOC     CFA      ra
0000615    rsp+8    c-8
0000619    rsp+48   c-8
0000659    rsp+8    c-8
```

Listing 2. Interpreted table

```
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 3. Bytecode compiled to C code

```
typedef struct {
    uint8_t flags;
    uintptr_t rip, rsp,
        rbp, rbx;
} unwind_context_t;
```

Listing 4. Unwinding context

Fig. 7. Sample compilation of DWARF bytecode to C code

register whether it is present or not in this context, plus an error bit useful to report errors due e.g. to unsupported instructions. Only registers `rip`, `rbp`, `rsp` and `rbx` are tracked by the contexts: in Section 3.2 we provide evidence that these are sufficient to perform stack-unwinding reliably (surprisingly `rbx` is used a few times in `glibc` and other less common libraries to hold the `CFA` address in common functions).

Observe that the `_eh_elf` function is compositional and it can be called multiple times in a row to unwind the stack, as in the pseudocode below:

```
while(!unwinding_done()) {
    current_context = _eh_elf(current_context[RA], current_context);
    do_something_with_context(current_context);
}
```

The C code for case `0x615 ... 0x618` in Listing 3 is compiled from the first two instructions of Listing 1. In Section 2 we assumed we had a reliable interpreter of DWARF bytecode and we were thus working directly with the rows and columns of the interpreted unwind table. Our ahead-of-time DWARF unwind compiler works instead directly on DWARF instructions and expressions. The DWARF standard is written in English prose and it only defines an informal semantics for instructions `DW_CFA_def_cfa: r7 (rsp)ofs 8` or `DW_CFA_offset: r16 (rip)at cfa-8`: the former sets the `CFA` of the current row at `rsp+8`, the latter sets the offset of the register `rip` (that is of the `ra` column) at `CFA-8`. The C code we generate is nothing more than a precise formalisation of the DWARF semantics. The full details are tedious, so we omit them here but we report the complete semantics and translation functions in Appendix A.

Implementation details. We store the compiled unwinding code in separate shared object files, which we call `eh_elfs`: whenever the unwinder needs the `_eh_elf` function, it dynamically links its

```c
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;
}
```

Fig. 8. Optimised C code for Listing 1

`eh_elfs`. For this the unwinder must first acquire a *memory map* from the OS: a table listing the ELF files loaded in memory with their memory segment. Once this map is acquired, to unwind from a given IP, the unwinder identifies the memory segment from which it comes, deduces the source ELF file, and deduces the corresponding `eh_elfs` file. Having separate files enables generating `eh_elfs` for system shared libraries without modifying the shared libraries themselves. A similar approach is taken by MacOS for all the debug information, which are stored as separate files from the executable. Additionally, packaging the `eh_elfs` files separately in future environment production would enable users interested in faster unwinding to install them on demand.

Naive generation of `eh_elfs` generates code which is roughly 7 times bigger than the original `.eh_frame`. Several space optimisations, such as filtering out empty FDEs and merging together equivalent rows, were made in order to shrink the size of the `eh_elfs`. Replacing the `switch/case` construct with an explicit `if/else` tree implementing a binary search on the instruction pointer intervals was extremely effective in reducing the binary size. We also *outline* the code whenever possible: identical "switch cases" bodies are moved outside of the binary search tree, identified with a label and invoked using a `goto`. When applied to the `eh_elfs` code for `libc`, it turns out that of a total of 20827 rows, only 302 (1.5 %) unique rows remain after the outlining.

For flexibility of use, a setting of our compiler 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.

3.1 Benchmarks and Evaluation

To provide relevant benchmarks of the `eh_elfs`' performance, hundreds or thousands of stack unwindings must be sampled, because a single frame unwinding with regular DWARF takes the order of magnitude of 10 μs. At the same time, to mimic real-world unwind calls, unwind points should be distributed across the program; we should not unwind repeatedly from the same IP in the same function. The `perf` sampling-based profiler is an ideal candidate for providing such benchmarks; when invoked to reconstruct the call-graph, `perf` stops the traced program at regular

intervals, unwinds the stack recording the current nested function calls, and integrates the sampled data in the end. As such it easily unwinds a running program thousands of times in a few seconds.

We run `perf`-benchmarking on several standard Unix binaries, including `gzip`, `find`, `sqlite3`, and the `python` runtime executing a simple program. Many of these spend most of the time in few functions, thus providing an ideal situation for comparing against `libunwind` caching. This allows us to evaluate a worst-case speedup for our ahead-of-time compilation strategy. Additionally, to stress-test the robustness of strategy, we run `perf`-benchmarking on `hackbench` [Zhang 2008]. This small program is designed to stress-test and benchmark the Linux scheduler by spawning processes or threads that communicate with each other: it generates large stack activity, and is additionally linked against the `pthread` library.

Interfacing `eh_elfs` with `perf` required adding `eh_elfs` support to `libunwind`, a widely used unwinder library. This required passing explicitly the memory map of the process to a library initialisation function; apart from this, the modified version of `libunwind` produced is entirely compatible with the vanilla version.

Measured time performance. The following evaluation was made on 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 to never rely on swap space.

Benchmarking of `eh_elfs` against the vanilla `libunwind` yielded the results in Table 1 (standard deviation is reported in parentheses). The sharp difference between cached and uncached `libunwind` confirms that our experimental setup does not unwind at totally different locations every single time, and thus was not biased in this direction, since caching proves efficient.

Table 1. Time benchmarking

Binary	Unwind method	Frames unwound	Total unwind time (μs)	Average time per frame (ns)	Unwind errors	Time ratio
gzip	`eh_elfs`	331523	25930 (2)	78	379	1
	`libunwind`, cached	331523	403292 (27)	1217	379	15.6
	`libunwind`, uncached	331523	2197296 (50)	6635	379	84.7
sqlite3	`eh_elfs`	42807	4423 (9)	135	5361	1
	`libunwind`, cached	42807	69688 (40)	1861	5361	13.7
	`libunwind`, uncached	42807	383832 (78)	10250	5361	75.6
find	`eh_elfs`	18157	2232 (5)	127	409	1
	`libunwind`, cached	18157	41469 (36)	2336	409	18.5
	`libunwind`, uncached	18157	143813 (99)	2336	409	99.0
python3.7	`eh_elfs`	203273	18201 (2)	97	9020	1
	`libunwind`, cached	203273	249145 (13)	1282	9020	13.2
	`libunwind`, uncached	203273	1930438 (195)	9937	9020	102.3
hackbench	`eh_elfs`	152297	12941 (5)	84	1	1
	`libunwind`, cached	152297	316907 (110)	2076	1	24.6
	`libunwind`, uncached	152297	982697 (120)	6439	1	76.3

The compilation time of `eh_elfs` is reasonable. On our test machine, without using multiple cores to compile, all the shared objects needed to run `hackbench` – that is, `hackbench`, `libc`, `ld` and `libpthread` – are compiled in an overall time of 25.28 seconds.

The unwinding errors are due to truncated stack records. Since `perf` dumps the last n bytes of the call stack (for a given n), and only keeps those for later unwinding, large stacks lead to lost information when analysing the results. Our implementation reports the same errors of the vanilla `libunwind` implementation.

Space overhead. We measure the space taken by the `eh_elfs` and compare it against the size of `.eh_frame` tables for `hackbench` and the libraries on which it depends. Outlining proves effective. We note that `hackbench` has a significantly bigger growth than the other shared objects. This is because `hackbench` has a much smaller `.eh_frame` and the outlined data is reused only a few times, compared to e.g. `glibc`, in which the outlined data is heavily reused.

Table 2. `eh_elfs` space usage

Shared object	Original program size	Original `.eh_frame`	Generated `eh_elf .text`	% of original program size	Growth factor
ld-2.29.so	141.0 KiB	9.8 KiB	33.2 KiB	23.58	3.40
libc-2.29.so	1.4 MiB	128.6 KiB	368.8 KiB	25.47	2.87
libdl-2.29.so	3.6 KiB	896.0 B	3.6 KiB	98.73	4.09
libm-2.29.so	1.2 MiB	36.9 KiB	109.1 KiB	9.19	2.96
libncursesw.so.6.1	282.3 KiB	35.3 KiB	103.5 KiB	36.67	2.93
libpthread-2.29.so	60.0 KiB	11.4 KiB	34.1 KiB	56.79	2.98
libpython3.7.so.1.0	2.0 MiB	260.7 KiB	623.9 KiB	29.85	2.39
libreadline.so.8.0	163.4 KiB	25.5 KiB	78.6 KiB	48.11	3.08
libutil-2.29.so	2.4 KiB	592 B	4.0 KiB	165.03	6.92
libz.so.1.2.11	69.0 KiB	6.1 KiB	20.7 KiB	30.02	3.37
find	146.5 KiB	21.3 KiB	68.3 KiB	46.63	3.21
find + libs	2.9 MiB	196.6 KiB	577.2 KiB	19.75	2.94
python3.7	393.0 B	160.0 B	1.4 KiB	355.98	8.33
python3.7 + libs	4.8 MiB	449.0 KiB	1.1 MiB	23.77	2.61
gzip	66.2 KiB	5.1 KiB	10.9 KiB	16.48	2.13
gzip + libs	1.6 MiB	143.5 KiB	413.1 KiB	24.96	2.88
hackbench	2.9 KiB	568.0 B	3.2 KiB	107.99	5.74
hackbench + libs	1.6 MiB	150.4 KiB	439.4 KiB	26.60	2.92
sqlite	1.1 MiB	121.7 KiB	382.8 KiB	34.68	3.14
sqlite + libs	4.4 MiB	376.2 KiB	1.1 MiB	25.32	3.00

3.2 Instruction Coverage

We conclude by an extensive investigation of which DWARF instructions should be implemented to have meaningful results, as well as to assess the instruction coverage of our `eh_elfs` compiler. For this we 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.

Table 3 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 registers supported by `eh_elfs`). The second row analyses all the columns that were encountered, no matter whether supported or not in `eh_elfs`.

Table 3. Instructions coverage statistics

	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 4. Instruction type 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 4 analyzes the proportion of the DWARF instructions used to update 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 covers the majority of registers. The data gathered (omitted in the tables) also suggests that only few expressions are common: more than 80 % of expressions are covered only by supporting two basic constructs.

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

4 RELATED WORK

We are not the first ones to have remarked the poor reliability of DWARF debug information, but most of the testing of DWARF tables is done with tools that lack a principled design.

For instance, LLVM has a stack-unwinding stress-test suite, briefly described in https://reviews.llvm.org/D10454, that steps through the code line by line and then tests unwinding from each instruction. We have seen that testing unwinding is slow, in particular it is much slower than validating a DWARF entry as done by our validator tool. This test-suite is reported to to need minutes to run even over simple code, and is not part of the default testing of LLVM.

Similar in principle but aimed at to provide quantitative values that indicate the quality of the debugging experience, the DExTer (Debugging Experience Tester) by Sony, available from https://github.com/SNSystems/dexter, drives an external debugger, running on small test programs, and collects information on the behaviour at each debugger step. The statistics reported at the end of the execution do not identify precisely bugs in the tables.

He et al. [2018] apply machine learning to synthesise debug information. Their tool takes a stripped binary and rebuilds meaningful variable names and types for values and functions

stored in registers or memory, using probabilistic models learned from non-stripped binaries. Stripping a binary does not remove the `.eh_frame`: the tool thus synthesises only the `.debug_line` and `.debug_types` tables, skipping the `.eh_frame` which are needed for unwinding. In this context machine learning enables debugging with some accuracy on binaries that would be very hard to debug otherwise, but it is unclear if machine learning could synthesise perfectly accurate unwinding tables, that are needed to ensure that the unwinder never follows a bad pointer.

Synthesis of DWARF frame information can be seen as reverse engineering of binary code, for which various other techniques have been demonstrated. Most similar are perhaps techniques for decompiling to high-level code [Cifuentes 1994], which must analyse binary code to extract patterns of control flow. However it does not focus on techniques for recovering stack frame sizes.

The work most closely related to ours is the ORC project, part of the Linux kernel. ORC is briefly described by [Corbet 2017] and stems from a previous project, called `objtool`, that statically verifies that the frame pointer is correctly updated along all the execution paths of an x86_64 binary. Building backtraces with a frame pointer requires that all functions which call other functions must first create a stack frame and update the frame pointer; otherwise the caller of the first function will be skipped on a stack trace. This is routinely done by the compiler when the `-fno -omit-frame-pointer` option is used, but for inline assembly the frame setup instructions have to be written by hand. Since the Linux kernel relies extensively on inline assembly, `objtool` attempts to verify that programmers do not accidentally break the frame pointer discipline. Building on `objtool`, Linux developers proposed a new, simple, binary format for unwind tables, called ORC, that can be edited using the internal `objtool` metadata. These include the size of the stack at each instruction or how to compute the base of the stack frame. At the time of writing ORC tables have the same expressiveness of the `.eh_frame` tables inferred by our synthesis tool. However they require an ad-hoc unwinder and, contrarily to our DWARF-based approach, they cannot be easily extended to support richer expressions. Since `objtool` is intimately bound to the Linux kernel, its analysis is tailored for control structures and coding conventions used in the kernel; for instance only indirect jumps arising from compilation of switch statements are supported, and there are symbol-table related restrictions on targets of call instructions. Performance-wise, the parsing ORC tables is much simpler than DWARF tables, and ORC-based unwinding is reported to be faster that DWARF-based unwinding; the ORC documentation mentions a factor of 20, but the methodology used to measure this speedup is not documented. As `eh_elfs`, ORC tables are stored off-band in the `.orc_unwind` and `.orc_unwind_ip` tables; ORC tables are reported to take about 50% more RAM space than DWARF-based `.eh_frame` tables.

Although DWARF informations can be seen as analogue to *stack maps* studied in the garbage collector literature, stack maps are used to identify accurately stack-allocated pointers. Often garbage collectors avoid to explicitly walk the stack either by allocating all the roots in an explicit pointer stack [Siebert 2001], or by keeping an explicit lists across stack frames [Henderson 2002]. Other approaches, as the lazy pointer stacks by Baker et al. [2009], identify all the stack allocated pointers by walking the stack at a GC safe point, and require reliable unwind tables.

Our technique for compilation of DWARF is complementary to a similar technique described by Kell [2015]. This also performs translation into a more efficiently queryable form, but of a different part of the DWARF information: it computes the layout of locals within stack frames, using the `.debug_info` section, whereas the present work computes frame sizes and saved register locations using the `.eh_frame` section. However, to the best of our knowledge, with the exclusion of the aggressive caching of the interpreted `.eh_frame` tables performed by unwinders as `libunwind`, there have been no previous attempts at speeding up DWARF-based unwinding.

5 FUTURE DIRECTIONS

In this paper we have designed and implemented techniques to validate or synthesise `.eh_frame` unwind tables, and to speed-up DWARF-based unwinding. Additionally we hope that this paper has illustrated some inner working of DWARF debug information, a central component of our computing infrastructure mostly ignored by the research community. Apart for the academic contribution, our long term aim is to integrate these tools in the wide low-level software ecosystem. For this some engineering work might be needed, for instance a faster validator might be implemented by instrumenting the QEMU machine emulator rather than relying on `gdb`, and tools might be ported to other architectures or ABIs. We thus conclude this paper with the following summary of our findings, highlighting the directions we are currently investigating.

- We showed that the `.eh_frame` validation tool, combined with the CSmith fuzzer, proved effective to identify bugs in the LLVM x86_64 backend. We also realised some preliminary testing of the `glibc` test suite, identifying one issue in subtle inline assembly code. Our aim is to make this testing part of the standard test-suites both of compilers and of code relying extensively on inline assembly code, as it improves on the rudimentary testing strategies used now;
- We showed that synthesis of `.eh_frame` tables is possible and effective. This synthesis strategy is of interest to both language implementors and debugger developers. Synthesis might be integrated in the inline-assembly support of mainstream compilers, so that correct `.eh_frame` tables can be generated without requiring the error prone task of specifying unwinding information in inline-assembly snippets. Synthesis might also be invoked on demand by debuggers and program analysis tools to analyse binaries that lack unwind tables, most notably JITted code.
- Speeding up DWARF based unwinding is valuable for the wider audience of tools and libraries that need to efficiently unwinding the stack (including profilers or language runtimes). Discussion is currently ongoing with developers of the `perf` profiler to enable part of the profiler call-stack analysis to be done online, with the benefit that the final `perf.data` file would not contain any sensitive information anymore.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their valuable feedback, John Regehr for suggestions at an early stage of the project, and Nhat Minh Lê for debugging a nasty interaction between CReduce and `gdb`. This project has received funding from ONR, award 503353, and Google, Faculty Research Award, 2018.

REFERENCES

J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. 2009. Accurate Garbage Collection in Uncooperative Environments Revisited. *Concurr. Comput. : Pract. Exper.* 21, 12 (Aug. 2009), 1572–1606. https://doi.org/10.1002/cpe.v21:12

Eli Bendersky. 2019. pyelftools. https://github.com/eliben/pyelftools

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* 463–469. https://doi.org/10.1007/978-3-642-22110-1_37

Cristina Cifuentes. 1994. *Reverse compilation techniques.* Ph.D. Dissertation. Queensland University of Technology. https://eprints.qut.edu.au/36820/

Jonathan Corbet. 2017. The ORCs are coming. *LWN.net* (2017). https://lwn.net/Articles/728339/

DWARF. 2017. *DWARF Debugging Information Format version 5.* DWARF Debugging Information Format Committee. http://dwarfstd.org

Anonymous Google Engineer. 2018. Personal communication.

Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18).* ACM, New York, NY, USA, 1667–1680. https://doi.org/10.1145/3243734.3243866

Fergus Henderson. 2002. Accurate Garbage Collection in an Uncooperative Environment. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM '02)*. ACM, New York, NY, USA, 150–156. https://doi.org/10.1145/512429.512449

Stephen Kell. 2015. Towards a Dynamic Object Model Within Unix Processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 224–239. https://doi.org/10.1145/2814228.2814238

James Oakley and Sergey Bratus. 2011. Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code. In *5th USENIX Workshop on Offensive Technologies, WOOT'11, August 8, 2011, San Francisco, CA, USA, Proceedings*. 91–102. http://static.usenix.org/event/woot11/tech/final_files/Oakley.pdf

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

Fridtjof Siebert. 2001. Constant-Time Root Scanning for Deterministic Garbage Collection. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*. Springer-Verlag, London, UK, UK, 304–318. http://dl.acm.org/citation.cfm?id=647477.727769

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. 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*. 283–294. https://doi.org/10.1145/1993498.1993532

Yanmin Zhang. 2008. Hackbench. https://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c

A DWARF UNWIND INSTRUCTION SEMANTICS

In this appendix we sketch a formal semantics for a subset of the DWARF 5 instructions. Since the standard [DWARF 2017] is written in English prose, we first recall the expected behaviour of DWARF instructions as provided by the standard, and later we formalise their semantics. We omit the translation of DWARF expressions: the standard defines a rich language but only a limited subset is used in practice, as discussed in Section 3.2.

The semantics of DWARF instructions is defined by translation to the C language, via an intermediate language. DWARF instructions can read the whole memory or register, and are executed for some instruction pointer. The C function representing their semantics thus takes as parameters an array of the current registers' values and an IP address, and returns another array of registers values, which represents the evaluated DWARF row.

Informal semantics of DWARF instructions. Below we report the DWARF instructions used for CFI description, descriptions have been reworded for brevity and clarity. Since we abstract from the underlying file format here, we omit variations differing only on the number of bytes of their operand, e.g. `advance_loc1` vs. `advance_loc2`.

- `set_loc(loc)`: start a new table row from address *loc*;
- `advance_loc(delta)`: start a new table row at address $prev_loc + delta$;
- `def_cfa(reg, offset)`: sets this row's CFA at ($\%reg + offset$);
- `def_cfa_register(reg)`: sets CFA at ($\%reg + prev_offset$);
- `def_cfa_offset(offset)`: sets CFA at ($\%prev_reg + offset$);
- `def_cfa_expression(expr)`: sets CFA as the result of *expr*;
- `undefined(reg)`: sets the register %reg as undefined in this row;
- `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;
- `offset(reg, offset)`: the value of the register %reg is stored in memory at the address $CFA + offset$;
- `val_offset(reg, offset)`: the value of the register %reg is the value $CFA + offset$;

- `register(reg, model)`: the register %reg has, in this row, the value of *%model*;
- `expression(reg, expr)`: the value of %reg is stored in memory at the address defined by *expr*;
- `val_expression(reg, expr)`: %reg has the value of *expr*;
- `restore(reg)`: %reg has the same value as in this FDE's preamble (CIE) in this row. We do not support this instruction, as it requires considerable boilerplate to differentiate CIE (preamble) and FDE (body) instructions, and is not used in practice (see Section 3.2);
- `remember_state()`: push the state of all the registers of this row on a state-saving stack;
- `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;
- `nop()`: do nothing (padding).

Intermediary language $\mathcal{I}$. A first pass translates DWARF instructions into the intermediate language $\mathcal{I}$. This language abstracts away the different instructions and defines the semantics as interpreted bytecode tables. The grammar is defined below:

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

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 $\%reg$ maps to a number. We use x86_64 names for convenience, although in DWARF registers are merely identifiers, that is $\%reg \in \mathbb{R}$.

A value can be undefined, stored at memory address x, or a value x. In this case x is a simple expression of the form $\%reg + \mathit{offset}$. The CFA is considered as any 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 leave open the possibility to extend the language with DWARF expressions support as Expr .

Target language: a C function body. The target language of these semantics is a C function expected 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 stack 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 `glibc`. In this case, unwinding the second frame requires loading the corresponding DWARF information. To simplify the presentation the function we use here is slightly simplified (in particular error reporting is omitted) with respect to `_eh_frame`, and is defined below:

```
#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:

```
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 – i.e. binary file – for two frames, we can safely unwind two frames this way:

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

Translation 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 with respect to 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.

At the same time we need to keep track of the state-saving stack relied upon by DWARF, which is kept in subscript. We define $[\![\bullet]\!]^{\mathcal{I}}_{s}(\bullet) : \text{DWARF} \times \text{FDE} \to \text{FDE}$, for s a stack of Row (e.g. $s \in \mathbb{S} := \text{Row}^*$, in Table 5. Implicitly, $[\![\bullet]\!]^{\mathcal{I}} := [\![\bullet]\!]^{\mathcal{I}}_{\varepsilon}$

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

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

We also 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 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, e.g. $F \xrightarrow{\text{CFA}} \text{Val}\,(\%reg + \text{off})$. 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. 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.

Table 5. Definition of $[\![\bullet]\!]_s^{\mathcal{I}}(\bullet)$

$$\begin{aligned}
[\![\varepsilon]\!]_s^{\mathcal{I}}(F) &:= F \\
[\![\texttt{set_loc(loc)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\,(F \cdot (loc, F[-1].row)) \\
[\![\texttt{adv_loc(delta)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\,(F \cdot (F[-1].addr + delta, F[-1].row)) \\
[\![\texttt{def_cfa(reg, offset)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\left(F \xleftarrow{\text{CFA}} \text{Val}\,(\%reg + offset)\right) \\
[\![\texttt{def_cfa_register(reg)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= \text{let F} \xrightarrow{\text{CFA}} \text{Val}\,(\%oldreg + oldoffset) \text{ in} \\
&\qquad [\![d]\!]_s^{\mathcal{I}}\left(F \xleftarrow{\text{CFA}} \text{Val}\,(\%reg + oldoffset)\right) \\
[\![\texttt{def_cfa_offset(offset)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= \text{let F} \xrightarrow{\text{CFA}} \text{Val}\,(\%oldreg + oldoffset) \text{ in} \\
&\qquad [\![d]\!]_s^{\mathcal{I}}\left(F \xleftarrow{\text{CFA}} \text{Val}\,(\%oldreg + offset)\right) \\
[\![\texttt{undefined(reg)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\left(F \xleftarrow{\%\text{reg}} \bot\right) \\
[\![\texttt{same_value(reg)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= \text{Val}\,(\%reg) \\
[\![\texttt{offset(reg, offset)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\left(F \xleftarrow{\text{reg}} \text{Addr}\,(CFA + offset)\right) \\
[\![\texttt{val_offset(reg, offset)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\left(F \xleftarrow{\text{reg}} \text{Val}\,(CFA + offset)\right) \\
[\![\texttt{register(reg, model)} \cdot d]\!]_s^{\mathcal{I}}(F) &:= \text{let } F \xrightarrow{\text{model}} r \text{ in } [\![d]\!]_s^{\mathcal{I}}\left(F \xleftarrow{\text{reg}} r\right) \\
[\![\texttt{remember_state()} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_{s\,\cdot\,(F[-1].row)}^{\mathcal{I}}(F) \\
[\![\texttt{restore_state()} \cdot d]\!]_{s\,\cdot\,t}^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\,(F\,[0 \ldots |F| - 2] \cdot (F[-1].addr, t)) \\
[\![\texttt{nop()} \cdot d]\!]_s^{\mathcal{I}}(F) &:= [\![d]\!]_s^{\mathcal{I}}\,(F)
\end{aligned}$$

Translation from $\mathcal{I}$ to C. The translation from $\mathcal{I}$ to C if defined by $[\![\bullet]\!]^C$: DWARF $\rightarrow$ C, 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

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

The generated C code constitutes a correct reference implementation. The compiler additionally performs the optimisations described in Section 3.