Growing the DWARF tougher: synthesis, validation and compilation

Théophile Bastian

Based on work done with
Francesco Zappa Nardelli, **Stephen Kell**, **Simon Ser**

ENS Paris, INRIA

Slides: `https://tobast.fr/files/oracle18.pdf`

1. DWARF and stack unwinding data
2. Unwinding data validation
3. Unwinding data synthesis from binaries
4. Unwinding data compilation

DWARF and stack unwinding data

We often use stack unwinding!

```
Program received signal SIGSEGV.
0x54625 in fct_b at segfault.c:5
5           printf("%l\n", *b);
```

We often use stack unwinding!

```
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
```

We often use stack unwinding!

```
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);
```

We often use stack unwinding!

```
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);

 (gdb) print a
$1 = 84
```

We often use stack unwinding!

```
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);

 (gdb) print a
$1 = 84
```

How does it work?!

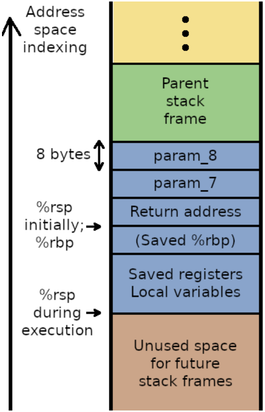
We often use stack unwinding!

```
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);

 (gdb) print a
$1 = 84
```

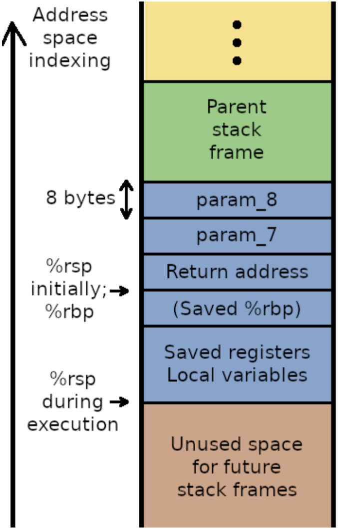


How does it work?!

Call stack and registers

How do we get the grandparent RA?

Isn't it as trivial as **pop()**?

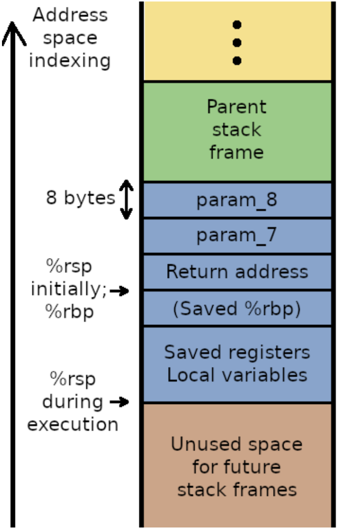


Call stack and registers

How do we get the grandparent RA?

Isn't it as trivial as **pop()**?

We only have %rsp and %rip.



DWARF unwinding data

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	u	u	u	c-32	c-24	c-16	c-8
0084958	rsp+40	u	u	c-40	c-32	c-24	c-16	c-8
0084959	rsp+48	u	c-48	c-40	c-32	c-24	c-16	c-8
008495a	rsp+56	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084962	rsp+64	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a19	rsp+56	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a1d	rsp+48	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a1e	rsp+40	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a20	rsp+32	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a22	rsp+24	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a24	rsp+16	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a26	rsp+8	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a30	rsp+64	c-56	c-48	c-40	c-32	c-24	c-16	c-8

DWARF unwinding data

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	u	u	u	c-32	c-24	c-16	c-8
0084958	rsp+40	u	u	c-40	c-32	c-24	c-16	c-8
0084959	rsp+48	u	c-48	c-40	c-32	c-24	c-16	c-8
008495a	rsp+56	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084962	rsp+64	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a19	rsp+56	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a1d	rsp+48	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a1e	rsp+40	c-56	c-48	c-40	c-32	c-24	c-16	[illegible]
0084a20	rsp+32	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a22	rsp+24	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0084a24	rsp+16	c-56	c-48	c-40	c-32	c-24	c-16	[illegible]
0084a26	rsp+8	c-56	c-48	c-40	c-32	c-24	c-16	[illegible]
0084a30	rsp+64	c-56	c-48	c-40	c-32	c-24	c-16	c-8

The real DWARF

```
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
  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
  [...]
```

⟶ **constructed on-demand by a Turing-complete bytecode!**

The real DWARF

```
00009b30 48 009b34 FDE cie=0000 pc=0084950..0084b37
  DW_CFA_advance_loc: 2 to 0000000000084952
  DW_CFA_def_cfa_offset: 16
  DW_CF
  DW_CF
  DW_CF
  DW_CF
  DW_CF
  DW_CFA_c
  DW_CFA_c
  DW_CFA_a
  DW_CFA_c
  DW_CFA_offset: r12 (r12) at cfa-40
  DW_CFA_advance_loc: 1 to 0000000000084959
  [...]
```

Complex & slow!

⟶ constructed on-demand by a Turing-complete bytecode!

Why does slow matter?

- After all, we're talking about debugging procedures ran by a human being (slower than the machine).
  . . . or are we?

Why does slow matter?

- After all, we're talking about debugging procedures ran by a human being (slower than the machine).
  ...or are we?

No!

Why does slow matter?

- After all, we're talking about debugging procedures ran by a human being (slower than the machine).
  . . . or are we?

No!

- Pretty much any program analysis tool

Why does slow matter?

- After all, we're talking about debugging procedures ran by a human being (slower than the machine).
  . . . or are we?

No!

- Pretty much any program analysis tool
- Profiling with polling profilers

Why does slow matter?

- After all, we're talking about debugging procedures ran by a human being (slower than the machine).
  . . . or are we?

No!

- Pretty much any program analysis tool
- Profiling with polling profilers
- Exception handling in C++

Debug data is not only for debugging

Difficult to generate

Major concern with DWARF: it is difficult to generate (correctly).

- Hard to generate: each compiler pass must keep it in sync
- Most of it is seldom used (*eg.* unwinding data of dusty code), and thus seldom tested

Yields to

- unreliable DWARF: can cause headaches when debugging
- or not generated at all (eg. OCaml until recently)

$\rightsquigarrow$ **Complex, buggy, untested**

A debugging hell: Linux kernel

> “Sorry, but last time was too f. . . painful. The whole (and only) point of unwinders is to make debugging easy when a bug occurs. But the 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.”
>
> — Linus Torvalds, Kernel mailing list, 2012

A debugging hell: Linux kernel

> "Sorry, but last time was too f. . . painful. The whole (and only) point of unwinders is to make debugging easy when a bug occurs. But the 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."
>
> — Linus Torvalds, Kernel mailing list, 2012

This is where we still are!

Unwinding data validation

Validating an example

```
<foo>:
  push   %r15
  push   %r14
  mov    $0x3,%eax
  push   %r13
  push   %r12
  push   %rbp
  push   %rbx
  sub    $0x68,%rsp
  cmp    $0x1,%edi
  add    $0x68,%rsp
  pop    %rbx
  pop    %rbp
```

Validating an example

```
<foo>:
  push   %r15
  push   %r14
  mov    $0x3,%eax
  push   %r13
  push   %r12
  push   %rbp
  push   %rbx
  sub    $0x68,%rsp
  cmp    $0x1,%edi
  add    $0x68,%rsp
  pop    %rbx
  pop    %rbp
```

CFA	**ra**
rsp+8	c-8
rsp+16	c-8
rsp+24	c-8
rsp+24	c-8
rsp+32	c-8
rsp+40	c-8
rsp+48	c-8
rsp+56	c-8
rsp+160	c-8
rsp+160	c-8
rsp+56	c-8
rsp+48	c-8

Validating an example

<foo>:

		CFA	ra
push	%r15	rsp+8	c-8
push	%r14	rsp+16	c-8
mov	$0x3,%eax	rsp+24	c-8
push	%r13	rsp+24	c-8
push	%r12	rsp+32	c-8
push	%rbp	rsp+40	c-8
push	%rbx	rsp+48	c-8
sub	$0x68,%rsp	rsp+56	c-8
cmp	$0x1,%edi	rsp+160	c-8
add	$0x68,%rsp	rsp+160	c-8
pop	%rbx	rsp+56	c-8
pop	%rbp	rsp+48	c-8

Upon function call, ra = *(%rsp) (ABI)

Validating an example

<foo>:		CFA	ra
push	%r15	rsp+8	c-8
push	%r14	rsp+16	c-8
mov	$0x3,%eax	rsp+24	c-8
push	%r13	rsp+24	c-8
push	%r12	rsp+32	c-8
push	%rbp	rsp+40	c-8
push	%rbx	rsp+48	c-8
sub	$0x68,%rsp	rsp+56	c-8
cmp	$0x1,%edi	rsp+160	c-8
add	$0x68,%rsp	rsp+160	c-8
pop	%rbx	rsp+56	c-8
pop	%rbp	rsp+48	c-8

push decreases %rsp by 8: $\mathrm{ra} = *(\%\mathrm{rsp} + 8)$

Validating an example

<foo>:		CFA	ra
push	%r15	rsp+8	c-8
push	%r14	rsp+16	c-8
mov	$0x3,%eax	rsp+24	c-8
push	%r13	rsp+24	c-8
push	%r12	rsp+32	c-8
push	%rbp	rsp+40	c-8
push	%rbx	rsp+48	c-8
sub	$0x68,%rsp	rsp+56	c-8
cmp	$0x1,%edi	rsp+160	c-8
add	$0x68,%rsp	rsp+160	c-8
pop	%rbx	rsp+56	c-8
pop	%rbp	rsp+48	c-8

and again: $ra = *(\%rsp + 16)$

Validating an example

<foo>:		CFA	ra
push	%r15	rsp+8	c-8
push	%r14	rsp+16	c-8
mov	$0x3,%eax	rsp+24	c-8
push	%r13	rsp+24	c-8
push	%r12	rsp+32	c-8
push	%rbp	rsp+40	c-8
push	%rbx	rsp+48	c-8
sub	$0x68,%rsp	rsp+56	c-8
cmp	$0x1,%edi	rsp+160	c-8
add	$0x68,%rsp	rsp+160	c-8
pop	%rbx	rsp+56	c-8
pop	%rbp	rsp+48	c-8

This mov leaves %rsp untouched: $ra = *(\%rsp + 16)$

Validating an example

```
<foo>:
  push   %r15
  push   %r14
  mov    $0x3,%eax
  push   %r13
  push   %r12
  push   %rbp
  push   %rbx
  sub    $0x68,%rsp
  cmp    $0x1,%edi
  add    $0x68,%rsp
  pop    %rbx
  pop    %rbp
```

CFA	**ra**
rsp+8	c-8
rsp+16	c-8
rsp+24	c-8
rsp+24	c-8
rsp+32	c-8
rsp+40	c-8
rsp+48	c-8
rsp+56	c-8
rsp+160	c-8
rsp+160	c-8
rsp+56	c-8
rsp+48	c-8

The unwinding table can actually be seen as
an abstract interpretation of the code. . .

Validating an example

```
<foo>:
  push   %r15
  push   %r14
  mov    $0x3,%eax
  push   %r13
  push   %r12
  push   %rbp
  push   %rbx
  sub    $0x68,%rsp
  cmp    $0x1,%edi
  add    $0x68,%rsp
  pop    %rbx
  pop    %rbp
```

CFA	**ra**
rsp+8	c-8
rsp+16	c-8
rsp+24	c-8
rsp+24	c-8
rsp+32	c-8
rsp+40	c-8
rsp+48	c-8
rsp+56	c-8
rsp+160	c-8
rsp+160	c-8
rsp+56	c-8
rsp+48	c-8

...and thus, for a given run, be re-computed for verification

Validating an example

<foo>:		CFA	ra
push	%r15	rsp+8	c-8
push	%r14	rsp+16	c-8
mov	$0x3,%eax	rsp+24	c-8
push	%r13	rsp+24	c-8
push	%r12	rsp+32	c-8
push	%rbp	rsp+40	c-8
push	%rbx	rsp+48	c-8
sub	$0x68,%rsp	rsp+56	c-8
cmp	$0x1,%edi	rsp+160	c-8

If, within an execution,

- ra = `*(0xFFFF1098)`
- %rsp = `0xFFFF1000`

We can evaluate both expressions and compare

Dynamic validation

Abstract state

- Stack of actual addresses where return addresses are stored

Dynamic validation

Abstract state

- Stack of actual addresses where return addresses are stored

Abstract instruction semantics

`call` push %rsp on the stack

`ret` pop from the stack

Dynamic validation

Abstract state

- Stack of actual addresses where return addresses are stored

Abstract instruction semantics

`call` push %rsp on the stack

`ret` pop from the stack

Validation of each instruction

- Evaluate the return address provided by DWARF
- Compare it with the value at the top of the stack

In practice: `eh_frame_check`

Strategy implemented and working: `eh_frame_check`

- gdb allows for Python instrumentation

In practice: `eh_frame_check`

Strategy implemented and working: `eh_frame_check`

- gdb allows for Python instrumentation

- Parse ELF and DWARF data (`pyelftools`)
- Run the binary inside gdb
- Pause at each (assembly) step
- Jointly evaluate DWARF data and the abstract stack
- Report upon error

Works, but. . . Python is slow!
A few thousand of ASM instructions/second (good enough)

A real bug!

```c
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++);
            }
}
```

CSmith
+ Creduce
+ eh_frame_check

⇝ **LLVM (3.8) bug!**

A real bug!

```
<foo>:
  4004e0  push    %rbx

          [. . . ]

  40061d  pop     %rbx
  40061e  retq
```

CFA	**ra**
rsp+8	c-8
rsp+16	c-8
rsp+16	c-8
rsp+16	c-8

A real bug!

Abstract state `[0xFFFF1000]`

<foo>:

```
                                    CFA     ra
4004e0   push    %rbx               rsp+8   c-8
                                    rsp+16  c-8
         [. . .]

40061d   pop     %rbx               rsp+16  c-8
40061e   retq                       rsp+16  c-8
```

A real bug!

Abstract state	[0xFFFF1000]
%*rsp*	0xFFFF1000

<foo>:

			CFA	**ra**
4004e0	push	%rbx	rsp+8	c-8
			rsp+16	c-8
	[. . .]			
40061d	pop	%rbx	rsp+16	c-8
40061e	retq		rsp+16	c-8

A real bug!

Abstract state `[0xFFFF1000]`
%*rsp* `0xFFFF1000`

✓

<foo>:

			CFA	ra
4004e0	push	%rbx	rsp+8	c-8
			rsp+16	c-8
	[. . .]			
40061d	pop	%rbx	rsp+16	c-8
40061e	retq		rsp+16	c-8

A real bug!

Abstract state	[0xFFFF1000]
%rsp	0xFFFF0FF8

```
<foo>:                                    CFA     ra
  4004e0  push    %rbx                    rsp+8   c-8
                                          rsp+16  c-8
          [. . . ]

  40061d  pop     %rbx                    rsp+16  c-8
  40061e  retq                            rsp+16  c-8
```

A real bug!

Abstract state	`[0xFFFF1000]`
%rsp	`0xFFFF0FF8`

✓

<foo>:

			CFA	**ra**
4004e0	push	%rbx	rsp+8	c-8
			rsp+16	c-8
	[. . .]			
40061d	pop	%rbx	rsp+16	c-8
40061e	retq		rsp+16	c-8

A real bug!

Abstract state [0xFFFF1000]
%rsp 0xFFFF0FF8

```
<foo>:                                   CFA     ra
  4004e0  push     %rbx                  rsp+8   c-8
                                         rsp+16  c-8
          [. . . ]

  40061d  pop      %rbx                  rsp+16  c-8
  40061e  retq                           rsp+16  c-8
```

A real bug!

Abstract state `[0xFFFF1000]`

%rsp `0xFFFF0FF8`

✓

<foo>:

			CFA	**ra**
4004e0	push	%rbx	rsp+8	c-8
			rsp+16	c-8
	[. . .]			
40061d	pop	%rbx	rsp+16	c-8
40061e	retq		rsp+16	c-8

A real bug!

Abstract state	[0xFFFF1000]
%*rsp*	0xFFFF1000

<foo>:

			CFA	**ra**
4004e0	push	%rbx	rsp+8	c-8
			rsp+16	c-8
	[. . .]			
40061d	pop	%rbx	rsp+16	c-8
40061e	retq		rsp+16	c-8

A real bug!

Abstract state	[0xFFFF1000]	✗
%*rsp*	0xFFFF1000	

<foo>:			**CFA**	**ra**
4004e0	push	%rbx	rsp+8	c-8
			rsp+16	c-8
	[. . .]			
40061d	pop	%rbx	rsp+16	c-8
40061e	retq		rsp+16	c-8

A real bug!

Abstract state	`[0xFFFF1000]`
%rsp	`0xFFFF1000`

✗

<foo>:

			CFA	ra
4004e0	push	%rbx	rsp+8	c-8
			rsp+16	c-8
	[. . .]			
40061d	pop	%rbx	rsp+16	c-8
40061e	retq		rsp+16	c-8

⇝ **LLVM bug #13161**

What for, in the end?

- We can find bugs in compilers

What for, in the end?

- We can find bugs in compilers
- We can validate DWARF tables!

What for, in the end?

- We can find bugs in compilers
- We can validate DWARF tables!
- . . . well, only along one execution path. . .

What for, in the end?

- We can find bugs in compilers
- We can validate DWARF tables!
- . . . well, only along one execution path. . .
- but mostly we are close to a working algorithm to synthesize unwinding data from binaries!

Unwinding data synthesis from binaries

Why would synthesis be useful?

Why would synthesis be useful?

- As said earlier, DWARF is complex

Why would synthesis be useful?

- As said earlier, DWARF is complex
- Some compilers do not generate it: hard to debug & profile.

Why would synthesis be useful?

- As said earlier, DWARF is complex
- Some compilers do not generate it: hard to debug & profile.
- Think of JIT-compiled assembly (eg. JVM)

Why would synthesis be useful?

- As said earlier, DWARF is complex
- Some compilers do not generate it: hard to debug & profile.
- Think of JIT-compiled assembly (eg. JVM)
- …or even hand-written inlined assembly!

Why would synthesis be useful?

- As said earlier, DWARF is complex
- Some compilers do not generate it: hard to debug & profile.
- Think of JIT-compiled assembly (eg. JVM)
- . . . or even hand-written inlined assembly!
  - Painful enough to write for not bothering with DWARF
  - May not even be known by the programmer, breaks gdb
  - May be wrong (remember Linus!)

What have we got so far?

We now want to synthesize unwinding data.

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\mathsf{CFA} = \%rsp - 8 \qquad \mathsf{RA} = \mathsf{CFA} + 8$$

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\mathtt{CFA} = \%rsp - 8 \qquad \mathrm{RA} = \mathtt{CFA} + 8$$

- For each instruction, we know how it changes `CFA`.

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\mathsf{CFA} = \%rsp - 8 \qquad \mathsf{RA} = \mathsf{CFA} + 8$$

- For each instruction, we know how it changes `CFA`.
- We assume `RA` constant wrt. `CFA`.

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\texttt{CFA} = \%rsp - 8 \qquad \texttt{RA} = \texttt{CFA} + 8$$

- For each instruction, we know how it changes `CFA`.
- We assume `RA` constant wrt. `CFA`.
  - ⇝ only `CFA` tracking matters (for unwinding)

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\mathtt{CFA} = \%rsp - 8 \qquad \mathtt{RA} = \mathtt{CFA} + 8$$

- For each instruction, we know how it changes `CFA`.
- We assume `RA` constant wrt. `CFA`.
  - ⇝ only `CFA` tracking matters (for unwinding)
- We had a working strategy for a linear execution

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\mathtt{CFA} = \%rsp - 8 \qquad \mathtt{RA} = \mathtt{CFA} + 8$$

- For each instruction, we know how it changes CFA.
- We assume RA constant wrt. CFA.
  - ⇝ only CFA tracking matters (for unwinding)
- We had a working strategy for a linear execution
- We still have to handle

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\mathtt{CFA} = \%rsp - 8 \qquad \mathtt{RA} = \mathtt{CFA} + 8$$

- For each instruction, we know how it changes CFA.
- We assume RA constant wrt. CFA.
  - ⇝ only CFA tracking matters (for unwinding)
- We had a working strategy for a linear execution
- We still have to handle
  - CFA expression

What have we got so far?

We now want to synthesize unwinding data. That means forgetting the blue part of the previous schemes.

- Upon entering a function, we know (ABI)

$$\mathsf{CFA} = \%rsp - 8 \qquad \mathsf{RA} = \mathsf{CFA} + 8$$

- For each instruction, we know how it changes CFA.
- We assume RA constant wrt. CFA.
  - ⇝ only CFA tracking matters (for unwinding)
- We had a working strategy for a linear execution
- We still have to handle
  - CFA expression
  - control flow graph

CFA expression

Two possibilities:

- Either %rbp is used as base pointer

CFA expression

Two possibilities:

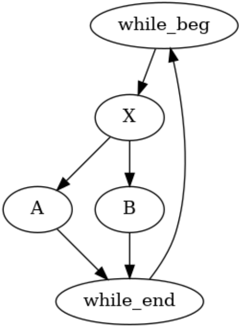
- Either %rbp is used as base pointer
- Or we must track `CFA` wrt. %rsp
    - And update it after each instruction if needed

Control flow graph

```
while(/* ... */) {
    X;
    if(/* ... */) {
        A;
    } else {
        B;
    }
}
```

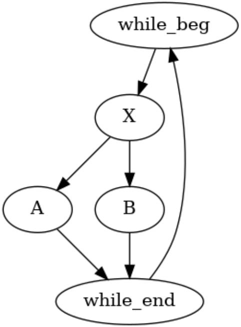
Control flow graph

```
while(/* ... */) {
    X;
    if(/* ... */) {
        A;
    } else {
        B;
    }
}
```



Control flow graph

```
while(/* ... */) {
    X;
    if(/* ... */) {
        A;
    } else {
        B;
    }
}
```



- Upon split (eg. X): nothing special, propagate end state of X to children nodes A and B
- Upon join (eg. `while_end`): check consistency of both input states
    - If tricky, gcc will have used %rbp, even with `-fomit-frame-pointer`.

Trust the compiler to avoid tricky unwinding

```
int z = rand();
for(int x=1; x < z; ++x) {
    int y[x]; // Variable size
    /* do something */
}
```

Trust the compiler to avoid tricky unwinding

```
int z = rand();
for(int x=1; x < z; ++x) {
    int y[x]; // Variable size
    /* do something */
}
```

- At each loop cycle, y is larger and allocated on the stack
- Thus, %rsp is farther from CFA at each cycle: no constant rule $\text{CFA} = \%rsp + k$.
- A complex DWARF expression is possible, but the compiler won't.

Trust the compiler to avoid tricky unwinding

```c
int z = rand();
for(int x=1; x < z; ++x) {
    int y[x]; // Variable size
    /* do something */
}
```

```
$ gcc -O0 -g -c src.c -fomit-frame-pointer
```

LOC	CFA	rbp	ra
000	rsp+8	u	c-8
001	rsp+16	c-16	c-8
004	rbp+16	c-16	c-8
010	rbp+16	c-16	c-8
0ce	rsp+8	c-16	c-8

- At each loop cycle, y is larger and allocated on the stack
- Thus, %rsp is farther from CFA at each cycle: no constant rule $\text{CFA} = \%rsp + k$.
- A complex DWARF expression is possible, but the compiler won't.

Demo time!

Unwinding data compilation

Why compiling?

- Remember that DWARF is slow!
- Bytecode interpreted on the fly to generate the data tables
- Done so for extreme compacity

Why compiling?

- Remember that DWARF is slow!
- Bytecode interpreted on the fly to generate the data tables
- Done so for extreme compacity

- Goal: reasonable time-space trade-off to speed up DWARF
- Tables are now compiled functions returning the requested DAWRF row

Compilation overview

- Compiled to C code
- C code then compiled to native binary (gcc)
  - ⇝ gcc optimisations for free
- Compiled as separate .so files, called `eh_elfs`

- Morally a monolithic switch on IPs
- Each case contains assembly that computes a row of the table

Compilation example: original C, DWARF

```
                                        DWARF
                                        CFA     ra
void fib7() {                 0x615     rsp+8   c-8
    int fibo[8];              0x620     rsp+48  c-8
    fibo[0] = 1;
    fibo[1] = 1;
    for(...)
        ...
    printf("%d\n", fibo[7]);
                              0x659     rsp+8   c-8
}
```

Compilation example: generated C

```c
unwind_context_t _eh_elf(
        unwind_context_t ctx, uintptr_t pc)
{
    unwind_context_t out_ctx;
    switch(pc) {
        ...
        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;
        ...
    }
}
```

Compilation choices

In order to keep the compiler simple and easily testable, the whole DWARF5 instruction set is not supported.

- Focus on x86_64
- Focus on unwinding return address
  $\rightsquigarrow$ *Allows building a backtrace*
  - suitable for perf, not for gdb
  - Only supports unwinding registers: %rip, %rsp, %rbp, %rbx
  - Supports the wide majority ($>$ 99.9%) of instructions used
  - Among 4000 randomly sampled filed, only 24 containing unsupported instructions

Interface: libunwind

- libunwind: *de facto* standard library for unwinding
- Relies on DWARF

- `libunwind-eh_elf`: alternative implementation using `eh_elfs`

⇝ alternative implementation of libunwind, almost plug-and-play for existing projects!

  ⇝ It is easy to use `eh_elfs`: just link against the right library!

Size optimisation: outlining

- Most of the rows boil down to a few common rows.
  - ⇝ ***outline*** **them!**

Size optimisation: outlining

- Most of the rows boil down to a few common rows.
  - ⇝ ***outline* them!**
- On libc, 20 827 rows → 302 outlined (1.5 %)
- Turn the big switch into a binary search if/else tree

Size optimisation: outlining

- Most of the rows boil down to a few common rows.
  ⇝ ***outline* them!**
- On libc, 20 827 rows → 302 outlined (1.5 %)
- Turn the big switch into a binary search if/else tree

⇝ only **2.5 times bigger than DWARF**

Example with outlining

```
unwind_context_t _eh_elf(
        unwind_context_t ctx, uintptr_t pc)
{
    unwind_context_t out_ctx;
    if(pc < 0x619) { ...  }
    else {
        if(pc < 0x659) { // IP=0x619 ... 0x658
            goto _factor_1;
        }
        ...
    }

    _factor_1:
    out_ctx.rsp = ctx.rsp + (48);
    out_ctx.rip = *((uintptr_t*)(out_ctx.rsp + (-8)));
    out_ctx.flags = 3u;

    ...

    return out_ctx;
}
```

Benchmarking requirements

1. Thousands of samples (single unwind: 10 μs)
2. Interesting enough program to unwind: nested functions, complex FDEs
3. Mitigate caching: don't always unwind from the *same* point
4. Yet be fair: don't always unwind from totally different places
5. Distribute evenly: if possible, also from within libraries

perf instrumentation

perf is a state-of-the-art polling profiler for Linux.

- used to get readings of the time spent in each function
- works by regularly stopping the program, unwinding its stack, then aggregating the gathered data

perf instrumentation

perf is a state-of-the-art polling profiler for Linux.

- used to get readings of the time spent in each function
- works by regularly stopping the program, unwinding its stack, then aggregating the gathered data

Instrumenting perf matches all the requirements!

- Plug eh_`elfs` into perf: use eh_`elfs` instead of DWARF to unwind the stack
- Implement unwinding performance counters inside perf

- Use perf on hackbench, a kernel stress-test program
  - Small program
  - Lots of calls
  - Relies on libc, libpthread

Time performance

Unwinding method	**Frames unwound**	**Tot. time (μs)**	**Avg. time / frame (ns)**	**Time ratio**
`eh_elfs`	23506	14837	631	1
`libunwind`, cached	27058	441601	16320	25.9
`libunwind`, uncached	27058	671292	24809	39.3

Space performance

Object	**% of binary size**	**Growth factor**
libc	21.88	2.41
libpthread	43.71	2.19
ld	22.09	2.97
hackbench	93.87	4.99
Total	22.81	2.44

Conclusion

"Sorry, but last time was too f… painful. The whole (and only) point of unwinders is to make debugging easy when a bug occurs. But the 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."

— Linus Torvalds, Kernel mailing list, 2012

"Sorry, but last time was too f... painful. The whole (and only) point of unwinders is to make debugging easy when a bug occurs. But the 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."

"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."

— Linus Torvalds, Kernel mailing list, 2012

"Sorry, but last time was too f. . . painful. The whole (and only) point of unwinders is to make debugging easy when a bug occurs. But the 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."

"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."

— Linus Torvalds, Kernel mailing list, 2012

Give us a few months: we will make Linus reconsider ;)

Keep Breathing

That's The Key

Slides: `https://tobast.fr/files/oracle18.pdf`