The rev.ng binary analysis framework

This presentation is available at:

rev.ng/presentation

All the demos are available at:

github.com/revng/demos

rev.ng Labs

1. 10 people
2. Partly in Milan, Italy, partly in rest of Europe
3. Compiler engineers/security researchers
4. We worked with

Qualcomm HUAWEI

Outline

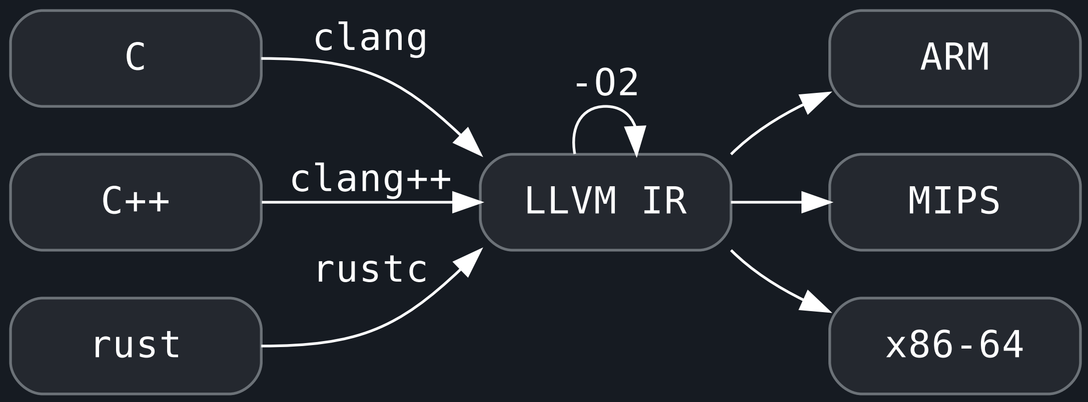
1. rev.ng design overview
2. **Demo**: how to interact with rev.ng
3. Let's put our hands into LLVM IR
4. **Demo**: automatic bug finding
5. Automated `struct` recovery
6. The state of rev.ng

rev.ng design overview

rev.ng is an open source
binary analysis framework
and interactive decompiler
for native code

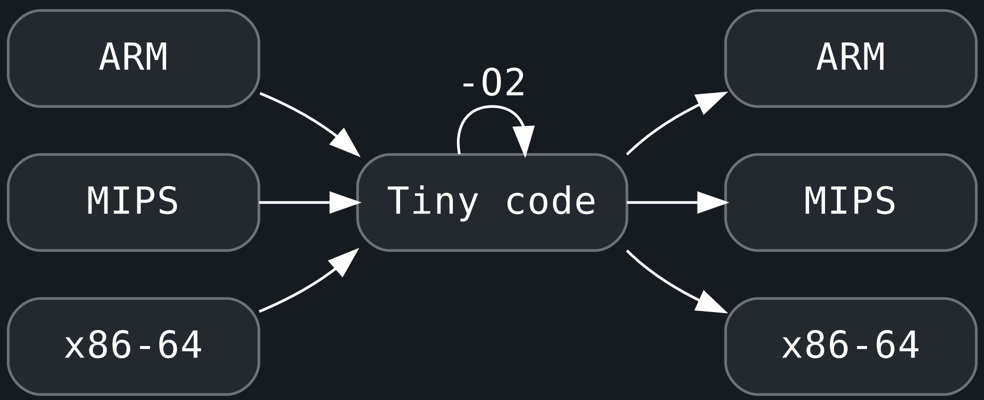
This is a compiler

LLVM



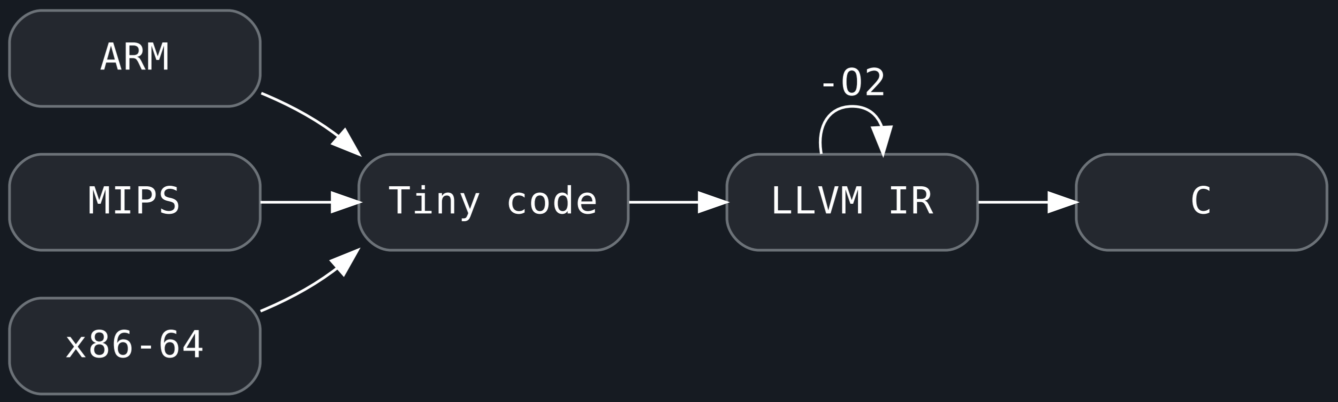
This is a dynamic binary translator

QEMU



This is a decompiler

rev.ng



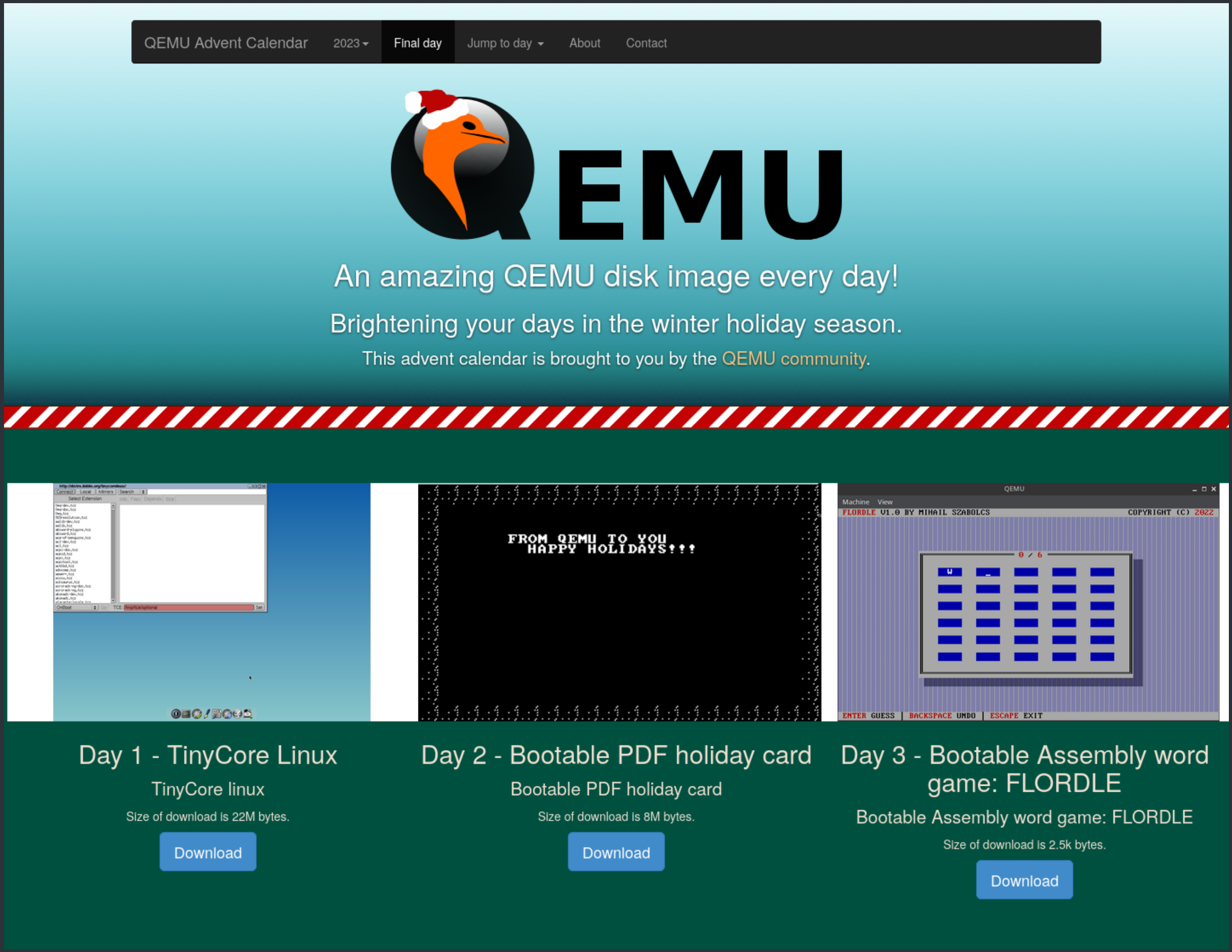
Using QEMU as a lifter

We use QEMU but

we do not run any code

We just use it as a lifter

Writing an accurate lifter is hard



QEMU Advent Calendar

Supporting many architecture is hard

QEMU supports:

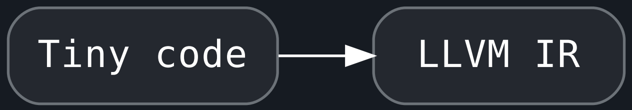
Alpha, ARM/AArch64, CRIS, HPPA, i386/x86-64, Hexagon, LatticeMico32, 68K, MicroBlaze, MIPS, Moxie, Nios2, OpenRISC, PowerPC, RISC-V, SH4, Sparc, s390x, TileGX, TriCore, Unicore32, Xtensa

QEMU is good for:

- accuracy
- supporting many architectures
- lifting
- dynamic binary translation

QEMU is bad for:

- anything else


Tiny code
LLVM IR

Here be dragons

LLVM

- Enable us to focus on building a decompiler, not a compiler framework
- Well known and big community
- Well defined semantics
- Many tools build on top of it (KLEE, Phasar, ...)
- High performance (C++)

A note on symbolic execution

We do not use symbolic execution in the pipeline.
However, we deem it appropriate for bug hunting!

A note on fuzzing

We've done it.

We're no longer focused on it.

There are a lot of effective alternative approaches.

How do I interact with rev.ng?

The model

- basically rev.ng's project file
- a YAML document
- contains everything the user can customize

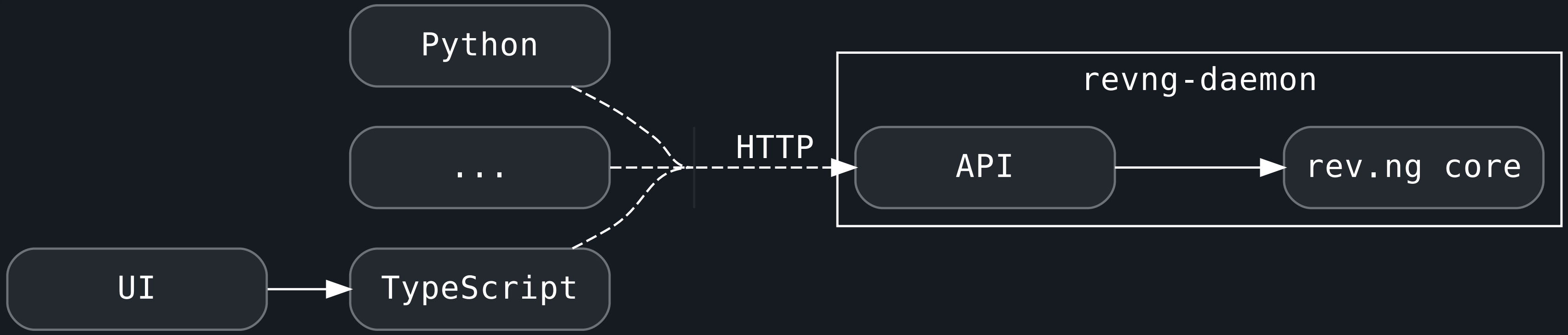
Example

```yaml
Architecture: x86_64
DefaultABI: SystemV_x86_64
Segments:
  - StartOffset: 0
    FileSize: 7
    StartAddress: "0x400000:Generic64"
Functions:
  - Entry: "0x400000:Code_x86_64"
Types:
  - Kind: StructType
    ID: 1
    Size: 8
    Fields:
      - Offset: 4
        Type: ...
```

Interaction option #1: the CLI

```
revng \
  artifact \
  --analyze \
  decompile-to-single-file \
  /bin/df
```

Interaction option #2: the daemon



tl;dr: users stay out-of-process

They can:

- use any language having YAML + HTTP/`system`
- use any version of the language they want
- use a different version of LLVM
- be on a different machine
- crash independently from rev.ng

No more

hex-rays

Products › Solutions Partners Shop Support › Company ›

IDAPython and Python 3

As of now (IDA version 7.3), IDA ships with an IDAPython plugin, that is compiled against, and compatible with Python 2.7.

The problem: Python 2.x end-of-life

The Python authors have decided that Python 3 has been available for long enough, to drop support for Python 2.x.

That effectively means that since Python 2.x will be unmaintained, it will gradually disappear from the landscape.

Moving IDAPython to Python 3

Work has begun (in fact, work is even finished) here at Hex-Rays to make IDAPython compilable, and compatible with Python 3.

Wrappers

In practice, we make things easier for Python and TypeScript users with wrappers.

users stay out-of-process

developers stay in-process

They have to buy into our dev stack (docs)

Demo time!

Try it!

tl;dr there are only two types of actions:

1. Request an artifact (LLVM IR, valid C, …)
2. Run an analysis/make changes to the model

It’s IR time!

Example program

```
long myfunction(long value) {
  long result = value;
  result = result * 2;
  return result;
}
```

Disassembly

```
myfunction:
  push    rbp
  mov     rbp,rsp
  mov     QWORD PTR [rbp-0x8],rdi
  mov     rax,QWORD PTR [rbp-0x8]
  mov     QWORD PTR [rbp-0x10],rax
  mov     rax,QWORD PTR [rbp-0x10]
  shl     rax,0x1
  mov     QWORD PTR [rbp-0x10],rax
  mov     rax,QWORD PTR [rbp-0x10]
  pop     rbp
  ret
```

15.3

```
define void @local_myfunction() {
  call @newpc(ptr nonnull @"revng.const.0x401130:Code_x86_64", i64 1, i32 1, i32 0,
ptr null, ptr null)
  %0 = load i64, ptr @rbp
  %1 = load i64, ptr @rsp
  %2 = add i64 %1, -8
  %3 = inttoptr i64 %2 to ptr
  store i64 %0, ptr %3
  store i64 %2, ptr @rsp
  call @newpc(ptr nonnull @"revng.const.0x401131:Code_x86_64", i64 3, i32 0, i32 0,
ptr null, ptr null)
  %4 = load i64, ptr @rsp
  store i64 %4, ptr @rbp
  call @newpc(ptr nonnull @"revng.const.0x401134:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %5 = load i64, ptr @rbp
  %6 = add i64 %5, -8
  %7 = load i64, ptr @rdi
  %8 = inttoptr i64 %6 to ptr
  store i64 %7, ptr %8
  call @newpc(ptr nonnull @"revng.const.0x401138:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %9 = load i64, ptr @rbp
  %10 = add i64 %9, -8
  %11 = inttoptr i64 %10 to ptr
  %12 = load i64, ptr %11
  store i64 %12, ptr @rax
  call @newpc(ptr nonnull @"revng.const.0x40113c:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %13 = load i64, ptr @rbp
  %14 = add i64 %13, -16
  %15 = load i64, ptr @rax
  %16 = inttoptr i64 %14 to ptr
  store i64 %15, ptr %16
  call @newpc(ptr nonnull @"revng.const.0x401140:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %17 = load i64, ptr @rbp
  %18 = add i64 %17, -16
  %19 = inttoptr i64 %18 to ptr
  %20 = load i64, ptr %19
  store i64 %20, ptr @rax
  call @newpc(ptr nonnull @"revng.const.0x401144:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %21 = load i64, ptr @rax
  %22 = shl i64 %21, 1
  store i64 %22, ptr @rax
  store i64 %22, ptr @cc_dst
  call @newpc(ptr nonnull @"revng.const.0x401148:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %23 = load i64, ptr @rbp
  %24 = add i64 %23, -16
  %25 = load i64, ptr @rax
  %26 = inttoptr i64 %24 to ptr
  store i64 %25, ptr %26
  call @newpc(ptr nonnull @"revng.const.0x40114c:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %27 = load i64, ptr @rbp
  %28 = add i64 %27, -16
  %29 = inttoptr i64 %28 to ptr
  %30 = load i64, ptr %29
  store i64 %30, ptr @rax
  call @newpc(ptr nonnull @"revng.const.0x401150:Code_x86_64", i64 1, i32 0, i32 0,
ptr null, ptr null)
  %31 = load i64, ptr @rsp
  %32 = inttoptr i64 %31 to ptr
  %33 = load i64, ptr %32
  %34 = add i64 %31, 8
  store i64 %34, ptr @rsp
  store i64 %33, ptr @rbp
  call @newpc(ptr nonnull @"revng.const.0x401151:Code_x86_64", i64 1, i32 0, i32 0,
ptr null, ptr null)
  %35 = load i64, ptr @rsp
  %36 = inttoptr i64 %35 to ptr
  %37 = load i64, ptr %36
  %38 = add i64 %35, 8
  store i64 %38, ptr @rsp
  store i64 %37, ptr @pc
  ret void
}
```

```
define i64 @local_myfunction(i64 %rdi_x86_64) {
  %0 = call i64 @_init_rbp()
  call @newpc(ptr nonnull @"revng.const.0x401130:Code_x86_64", i64 1, i32 1, i32 0,
ptr null, ptr null)
  %1 = load i64, ptr @rsp
  %2 = add i64 %1, -8
  %3 = inttoptr i64 %2 to ptr
  store i64 %0, ptr %3
  store i64 %2, ptr @rsp
  call @newpc(ptr nonnull @"revng.const.0x401131:Code_x86_64", i64 3, i32 0, i32 0,
ptr null, ptr null)
  %4 = load i64, ptr @rsp
  call @newpc(ptr nonnull @"revng.const.0x401134:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %5 = add i64 %4, -8
  %6 = inttoptr i64 %5 to ptr
  store i64 %rdi_x86_64, ptr %6
  call @newpc(ptr nonnull @"revng.const.0x401138:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %7 = load i64, ptr %6
  call @newpc(ptr nonnull @"revng.const.0x40113c:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %8 = add i64 %4, -16
  %9 = inttoptr i64 %8 to ptr
  store i64 %7, ptr %9
  call @newpc(ptr nonnull @"revng.const.0x401140:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %10 = load i64, ptr %9
  call @newpc(ptr nonnull @"revng.const.0x401144:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %11 = shl i64 %10, 1
  call @newpc(ptr nonnull @"revng.const.0x401148:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  store i64 %11, ptr %9
  call @newpc(ptr nonnull @"revng.const.0x40114c:Code_x86_64", i64 4, i32 0, i32 0,
ptr null, ptr null)
  %12 = load i64, ptr %9
  call @newpc(ptr nonnull @"revng.const.0x401150:Code_x86_64", i64 1, i32 0, i32 0,
ptr null, ptr null)
  %13 = load i64, ptr @rsp
  %14 = add i64 %13, 8
  store i64 %14, ptr @rsp
  call @newpc(ptr nonnull @"revng.const.0x401151:Code_x86_64", i64 1, i32 0, i32 0,
ptr null, ptr null)
  %15 = load i64, ptr @rsp
  %16 = add i64 %15, 8
  store i64 %16, ptr @rsp
  ret i64 %12
}
```

```
define i64 @local_myfunction(i64 %0) {
  %1 = call i64 @revng_stack_frame(i64 24)
  %2 = call i64 @AddressOf(ptr nonnull
@revng.const.c10d6afb753dc601da714646784a7e4040e86f7b, i64 %1)
  %3 = add i64 %2, 8
  %4 = inttoptr i64 %3 to ptr
  %5 = call ptr @stack_offset(ptr %4, i64 -16, i64 -7)
  store i64 %0, ptr %5
  %6 = inttoptr i64 %2 to ptr
  %7 = shl i64 %0, 1
  %8 = call ptr @stack_offset(ptr %6, i64 -24, i64 -15)
  store i64 %7, ptr %8
  ret i64 %7
}
```

```
define i64 @local_myfunction(i64 %0) {
  %1 = shl i64 %0, 1
  ret i64 %1
}
```

A couple of demos

1. Write a small taint analysis (try it!)
2. Collect some information about loops (try it!)

-ENOTIME

You can check them out at

`github.com/revng/demos`

Things you can do in LLVM

- Graph theory
  - Build the dominator tree
  - Identify strongly connected components
  - Perform visits (depth first, topological…)
- Manipulate functions
  - Inline
  - Outline
  - Specialize

Let’s find some bugs!

- Use `revng` for automated bug finding in binaries
- With static analysis tools for LLVM IR or C Code
- Demo on LLVM IR: KLEE
- We’ve got offline demos on C with
  - Clang Static Analyzer
  - CodeQL

What is KLEE

- Symbolic execution engine for LLVM
- Works on vanilla LLVM IR
- Has a builtin set of states known as bugs:
  - `assert` fail
  - various classes of memory bugs
- Regular use from source code

Demo time!

Try it!

Bug-finding on decompiled C?

- Take the same input program
- Decompile it with `revng`
- Feed it into source-level static analysis tools
  - Clang Static Analyzer
  - CodeQL
- Profit!

Clang static analyzer (try it!)

Original C

```c
void my_free(void *p) {
    free(p);
}

int do_stuff(int condition) {
  int *p = malloc(sizeof(int));
  if (condition > 4)
    my_free(p);
  *p = 3;
  int result = *p;
  my_free(p);
  return result;
}
```

Clang Static Analyzer Report

```
_ABI(SystemV_x86_64)
generic64_t do_stuff(generic64_t _argument0) {
  void *_var_0;
  _var_0 = malloc_((size_t) 4);
          2 ← Calling 'malloc_' →
          4 ← Returned allocated memory →
  if ((int32_t) (generic32_t) _argument0 > (int32_t) 2) {
          5 ← Assuming '_argument0' is > 2 →
          6 ← Taking true branch →
    my_free((generic32_t *) _var_0);
          7 ← Calling 'my_free' →
          11 ← Returning; memory was released via 1st parameter →
  }
  *(generic32_t *) _var_0 = 3;
          12 ← Use of memory after it is freed
  my_free((generic32_t *) _var_0);
  return 3;
}
```

CodeQL (try it!)

Original C

```
int do_stuff(int condition) {
  int *p = malloc(sizeof(int));
  if (condition > 4)
    free(p);
  *p = 3;
  int result = *p;
  free(p);
  return result;
}
```

CodeQL Report

```
_ABI(SystemV_x86_64)
generic64_t do_stuff(generic64_t _argument0) {
  void *_var_0;
  _var_0 = malloc((size_t) 4);
  if ((int32_t) (generic32_t) _argument0 > (int32_t) 4) {
    // BUG 1: call to free
    // BUG 2: call to free
    free((generic32_t *) _var_0); // line 69
  }
  // BUG 1: Potential use after free
  // An allocated memory block is used after it has been freed.
  // Behavior in such cases is undefined and can cause memory corruption.
  // Memory may have been previously freed by call to free at line 69.
  *(generic32_t *) _var_0 = 3;
  // BUG 2: Potential double free
  // Behavior in such cases is undefined and can cause memory corruption.
  // Memory may already have been freed by call to free at line 69.
  free((generic32_t *) _var_0);
  return 3;
}
```

Automated bug-finding with rev.ng

- `revng` output quality plays well with tooling
- Different levels of abstraction: LLVM IR or C
- LLVM: crossroad of project for static analysis
- C: has a huge base of analysis tools

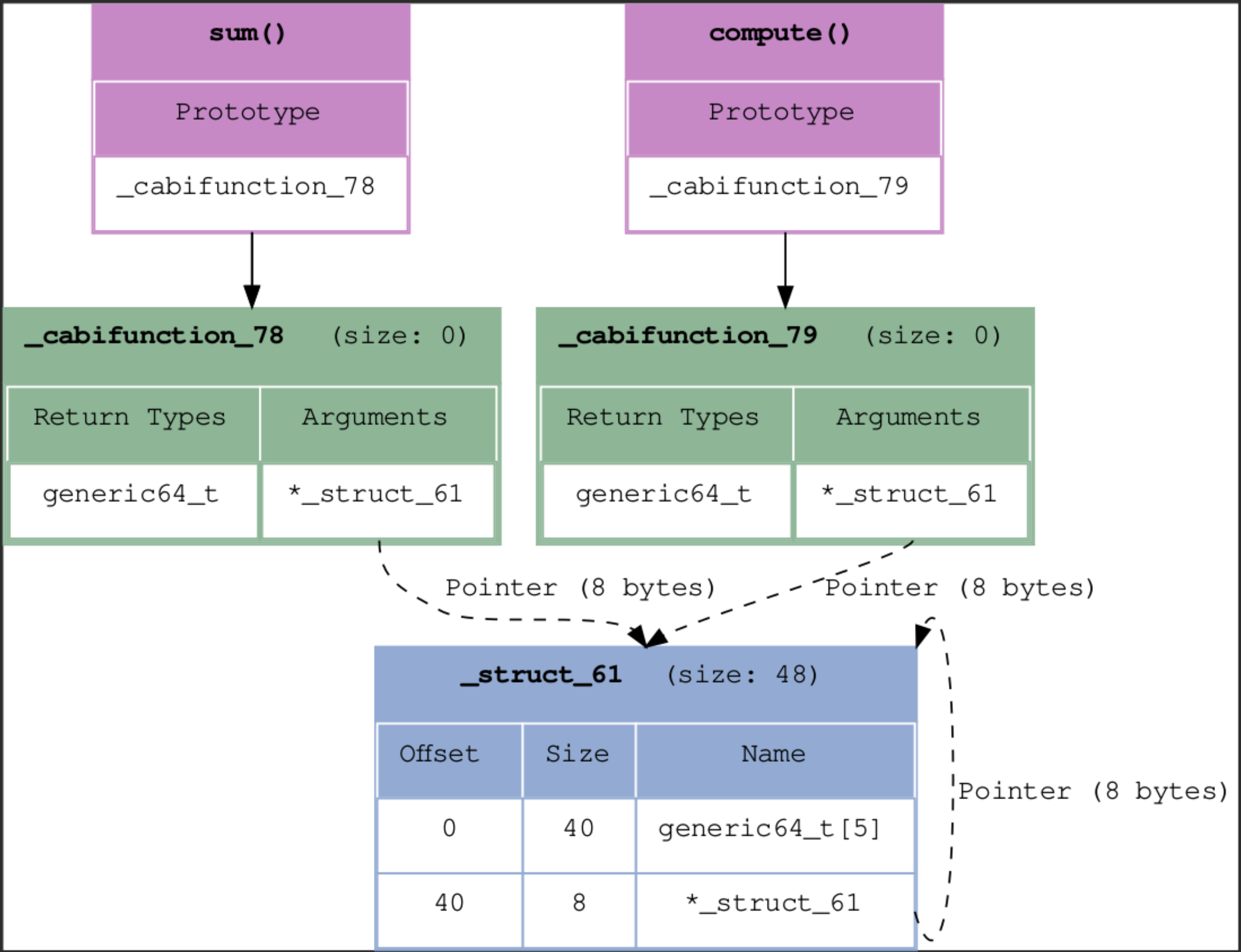
Automated type recovery

No type recovery

```
generic64_t sum(generic64_t _argument0) {
  generic64_t _var_0 = 0, _var_1 = 0;
  do {
    _var_1 = _var_1 + *(generic64_t *) ((_var_0 << 3) + _argument0);
    _var_0 = _var_0 + 1;
  } while (_var_0 != 5);
  return _var_1;
}

generic64_t compute(generic64_t _argument0) {
  generic64_t _var_0 = _argument0, _var_1 = 0;
  generic64_t _var_2;
  do {
    gen_var_2 = sum(_var_0);
    _var_1 = _var_1 + _var_2;
    _var_0 = *(generic64_t *) (_var_0 + 40);
  } while (_var_0);
  return _var_1;
}
```

With automated type recovery

```c
typedef struct _PACKED _struct_61 {
  generic64_t _offset_0[5];
  _struct_61 *_offset_40;
} _struct_61;

generic64_t sum(_struct_61 *_argument0) {
  generic64_t _var_0 = 0, _var_1 = 0
  do {
    _var_0 = _var_0 + _argument0->_offset_0[_var_1];
    _var_1 = _var_1 + 1;
  } while (_var_1 != 5);
  return _var_0;
}

generic64_t compute(_struct_61 *_argument0) {
  _struct_61 *_var_0 = _argument0
  generic64_t _var_1 = 0, _var_2;
  do {
    _var_2 = sum(_var_0);
    _var_1 = _var_1 + _var_2;
    _var_0 = _var_0->_offset_40;
  } while (_var_0);
  return _var_1;
}
```


sum()
Prototype
_cabifunction_78
compute()
Prototype
_cabifunction_79
_cabifunction_78 (size: 0)
Return Types
Arguments
generic64_t
*_struct_61
_cabifunction_79 (size: 0)
Return Types
Arguments
generic64_t
*_struct_61
Pointer (8 bytes)
Pointer (8 bytes)
_struct_61 (size: 48)
Offset
Size
Name
0
40
generic64_t[5]
40
8
*_struct_61
Pointer (8 bytes)

The status of rev.ng

Supported CPUs

- x86
- x86-64
- ARM
- AArch64
- MIPS
- SystemZ

Importers

- ELF
- DWARF
- PE/COFF
- CodeView (.pdb)
- Mach-O
- IDA Pro

Where does rev.ng run?

- Daemon
  - Linux x86-64 natively
  - macOS via Docker
  - Windows via WSL
- Clients can run anywhere

Recently, we released the pipeline as open source.

We’re now focusing

on robustness and performance.

Performance

We currently produce IR for GCC in 18 minutes.

- New argument detection analysis: 2.1x
- Reduce invalidation: ~1.7x (10min expected)
- Other low effort fixes: ~2x (5min expected)

Goal: be on par with Ghidra and IDA

1min 30sec and 40sec

Next up: mass testing

- Decompile all binaries on:
  - Ubuntu x86-64
  - Windows x86-64
  - Android AArch64
- Focus on:
  - optimizing performance of bottlenecks
  - squashing bugs

In short, rev.ng

- is FLOSS
- is declarative in user interactions
- has a modern design
- uses a “standard” IR and emits valid C
- interacts with existing tools
- has a nice (commercial) UI

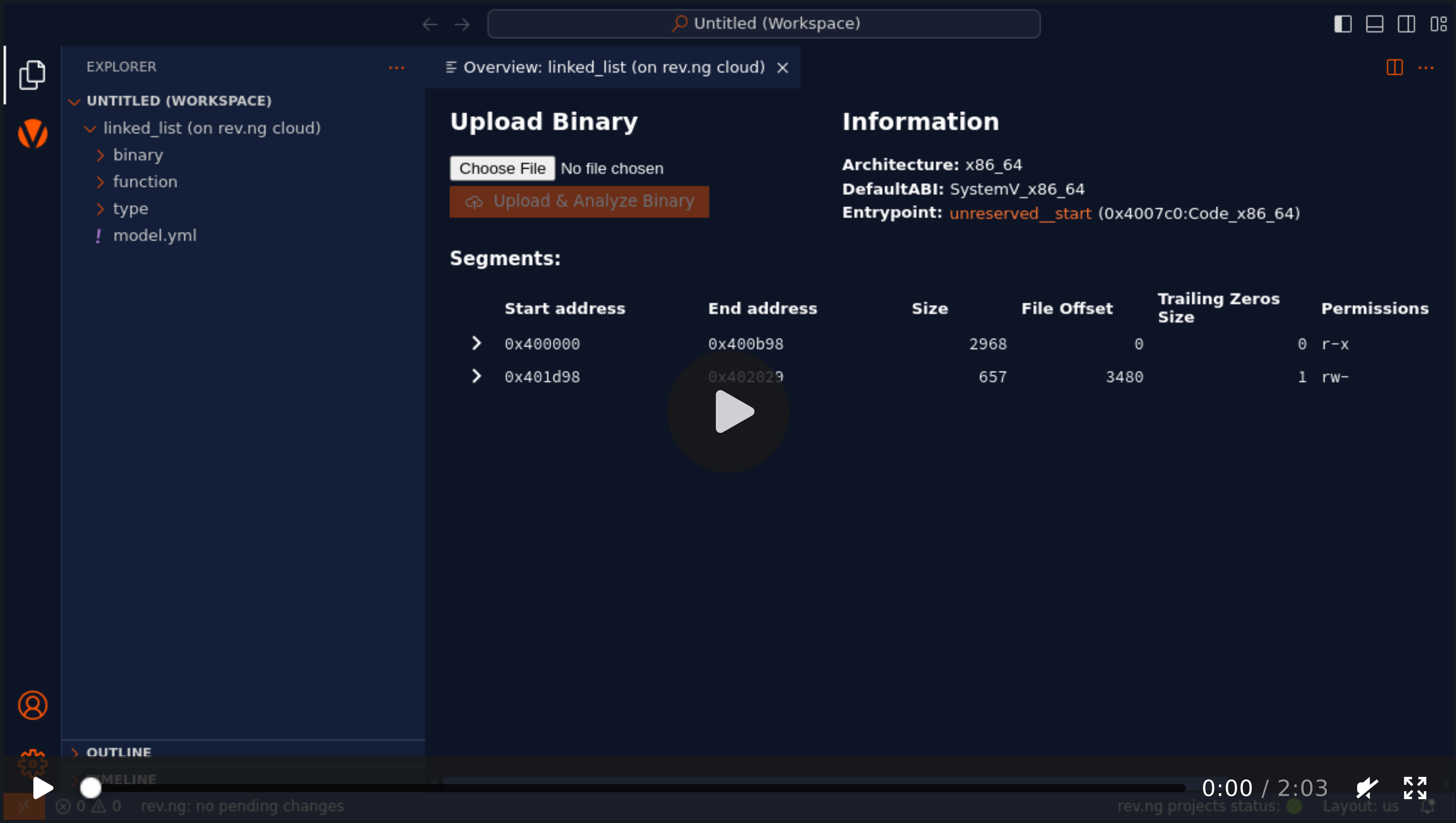
rev.ng UI

- Based on VSCode, mostly a plugin
- Connects to `revng-daemon`
- Runs as a standalone app or in the cloud
- Collaboration just works out of the box
- Also, hub.rev.ng (think GitHub for reversers)
- Cloud version will be free for public projects

Final note

We’re hiring and we do consulting

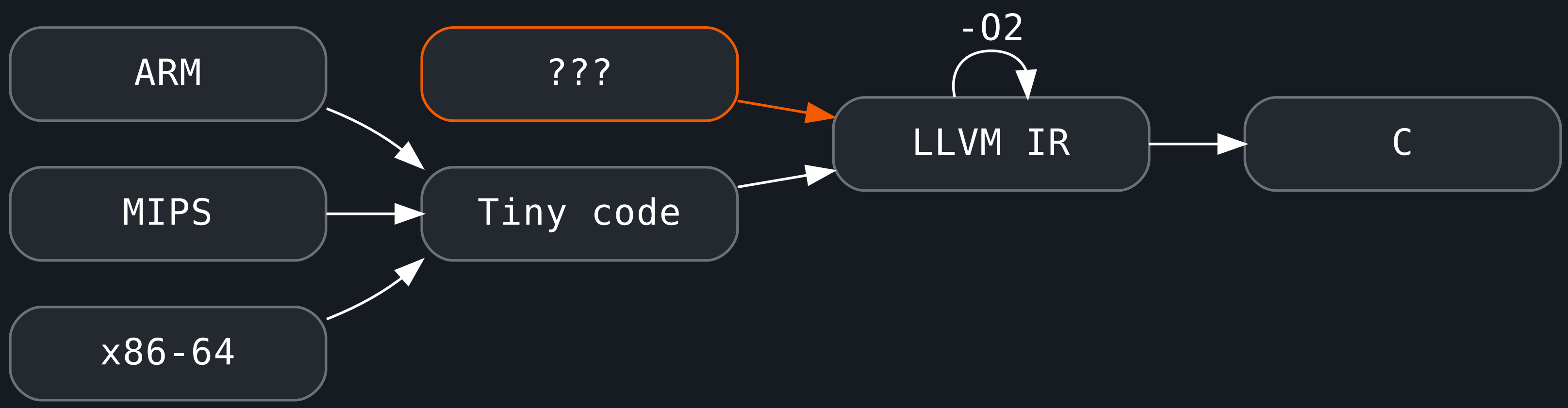
Questions?



Untitled (Workspace)
EXPLORER
UNTITLED (WORKSPACE)
linked_list (on rev.ng cloud)
binary
function
type
model.yml
OUTLINE
Overview: linked_list (on rev.ng cloud)
Upload Binary
Choose File No file chosen
Upload & Analyze Binary
Information
Architecture: x86_64
DefaultABI: SystemV_x86_64
Entrypoint: unreserved__start (0x4007c0:Code_x86_64)
Segments:
Start address
End address
Size
File Offset
Trailing Zeros Size
Permissions
0x400000 0x400b98 2968 0 0 r-x
0x401d98 657 3480 1 rw-
0 0 rev.ng: no pending changes
rev.ng projects status:
0:00 / 2:03

Backup slides

non-QEMU frontends?



Idea

```c
typedef struct {
  uint32_t rax;
  uint32_t rdi;
  // ...
} CPUState;

void add(CPUState *state) {
  state->rax = state->rax + state->rdi;
}
```

WIP, particularly interesting for WebAssembly