Introspection via Self Debugging

Russell Harmon
reh5586@cs.rit.edu

Rochester Institute of Technology
Computer Science

December 12, 2013

Overview

- Debuggers
- Introspection
- Introspection + Debugging
- Ruminate
  - Use
  - Internals
- Limitations
- Future Work

Debugger

... a computer program that is used to test and debug other programs -Wikipedia

Debugger

> *... a computer program that is used to test and debug other programs -Wikipedia*

- LLDB - LLVM debugger
- GDB - GNU debugger
- WinDBG - Windows debugger
- DBX - Sun Studio debugger
- LDB - Intel debugger

Debugging Example

```c
struct foo {
    char *str;
};
int main() {
    struct foo bar;
    bar.str = "Hello World!";
    asm("int $0x3");
}
```

Debugging Example

```
(lldb) run
* thread #1: tid = 0x1c03, 0x0000000100000f65
  a.out‘main + 21 at struct.c:9, stop reason =
  EXC_BREAKPOINT (code=EXC_I386_BPT, subcode=0x0)
    frame #0: 0x0000000100000f65 a.out‘main + 21
    at struct.c:9
   6       struct foo bar;
   7       bar.str = "Hello World!";
   8       asm("int $0x3");
-> 9     }
```

Debugging Example

```
(lldb) run
* thread #1: tid = 0x1c03, 0x0000000100000f65
  a.out`main + 21 at struct.c:9, stop reason =
  EXC_BREAKPOINT (code=EXC_I386_BPT, subcode=0x0)
    frame #0: 0x0000000100000f65 a.out`main + 21
    at struct.c:9
   6       struct foo bar;
   7       bar.str = "Hello World!";
   8       asm("int $0x3");
-> 9    }
(lldb) print bar
(foo) $0 = {
  (char *) str = 0x0000000100000f67 "Hello World!"
}
```

Debugging Example

```
(lldb) print bar
(foo) $0 = {
  (char *) str = 0x0000000100000f67 "Hello World!"
}
```

Debugging Example

```
(lldb) print bar
(foo) $0 = {
  (char *) str = 0x0000000100000f67 "Hello World!"
}
```

```c
struct foo {
    char *str;
};
int main() {
    struct foo bar;
    bar.str = "Hello World!";
    asm("int $0x3");
}
```

Introspection

> *... the ability for a program to examine the type or properties of an object at runtime. -Wikipedia*

- Interpreted languages have reflection / introspection.
- introspection $\subset$ reflection
- Java: `this.getClass().getCanonicalName()`
- Ruby: `self.class.name`

Introspection + Debugging

```c
struct foo {
    char *str;
};
int main() {
    struct foo bar;
    bar.str = "Hello World!";
    printf("%s\n", gettype(bar)->name);
    printf("%s\n", gettype(bar.str)->name);
}
```

Introspection + Debugging

```
struct foo {
    char *str;
};
int main() {
    struct foo bar;
    bar.str = "Hello World!";
    printf("%s\n", gettype(bar)->name);
    printf("%s\n", gettype(bar.str)->name);
}
```

```
foo
char *
```

Use Cases

- `abort_with_stacktrace();`
- easy data structure traversal
- access to third party library internals

Use Cases

- `abort_with_stacktrace();`
- easy data structure traversal
- access to third party library internals

Use Cases

- `abort_with_stacktrace();`
- easy data structure traversal
- access to third party library internals

Ruminate

- Introspective library for C
- Data introspection (structs, unions, enums, functions, primitives)
- Stack introspection

Ruminate

- Introspective library for C
- Data introspection (structs, unions, enums, functions, primitives)
- Stack introspection

Ruminate

- Introspective library for C
- Data introspection (structs, unions, enums, functions, primitives)
- Stack introspection

Ruminate

Simple Use

```c
// Includes go here
struct foo {
    char *str;
};
int main( int argc, char *argv[] ) {
    struct foo bar;
    bar.str = "Hello World!";
    ruminate_init(argv[0], NULL);
    printf("%s\n", r_string_bytes(r_type_name(
      ruminate_get_type(bar, NULL), NULL)));
    printf("%s\n", r_string_bytes(r_type_name(
      ruminate_get_type(bar.str, NULL), NULL)));
}
```

Ruminate

Simple Use

```c
// Includes go here
struct foo {
    char *str;
};
int main( int argc, char *argv[] ) {
    struct foo bar;
    bar.str = "Hello World!";
    ruminate_init(argv[0], NULL);
    printf("%s\n", r_string_bytes(r_type_name(
      ruminate_get_type(bar, NULL), NULL)));
    printf("%s\n", r_string_bytes(r_type_name(
      ruminate_get_type(bar.str, NULL), NULL)));
}
```

```
foo
char *
```

Ruminate

Simple Use

A rundown of the ruminate functions used

- `ruminate_init` initializes Ruminate
- `ruminate_get_type` retrieves an `RType` representing a type
- `r_type_name` retrieves an `RString` containing the name of an `RType`
- `r_string_bytes` retrieves a **`char *`** containing the value of an `RString`

Ruminate

Ok, but that's <u>really</u> simple...

Ruminate

Type Traversal

```c
// Includes go here
struct foo { char *str; };
int main( int argc, char *argv[] ) {
    struct foo bar;
    bar.str = "Hello World!";
    ruminate_init(argv[0], NULL);
    RType *type = ruminate_get_type(bar, NULL);
    RAggregateType *agg_type = (RAggregateType *) type;
    RAggregateMember *type_member =
        r_aggregate_type_member_by_name(agg_type, "str", NULL);
    RType *member_type = r_type_member_type(
            (RTypeMember *) type_member, NULL);
    RString *member_name = r_type_name(member_type, NULL);
    const char *member_name_str = r_string_bytes(member_name);
    printf("%s\n", member_name_str);
}
```

Ruminate

Type Traversal

```c
// Includes go here
struct foo { char *str; };
int main( int argc, char *argv[] ) {
    struct foo bar;
    bar.str = "Hello World!";
    ruminate_init(argv[0], NULL);
    RType *type = ruminate_get_type(bar, NULL);
    RAggregateType *agg_type = (RAggregateType *) type;
    RAggregateMember *type_member =
        r_aggregate_type_member_by_name(agg_type, "str", NULL);
    RType *member_type = r_type_member_type(
            (RTypeMember *) type_member, NULL);
    RString *member_name = r_type_name(member_type, NULL);
    const char *member_name_str = r_string_bytes(member_name);
    printf("%s\n", member_name_str);
}
```

```
char *
```

Ruminate

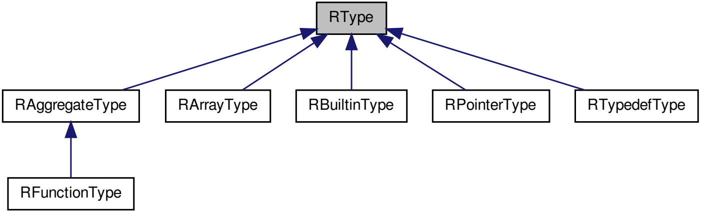
Type Traversal

Only a few new functions

- `r_aggregate_type_member_by_name` gets a member of an aggregate type by it's name
- `r_type_member_type` gets the type of an aggregate member

Ruminate

Type Hierarchy



Ruminate

Still too simple?

Ruminate

Still too simple?
Ok, well there's this

Ruminate

Complex Example

```c
#include <ruminate.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef char *string_t;

struct foo {
    string_t str;
    int i;
};

void print_data( RType *type, const void *data ) {
    switch( r_type_id(type, NULL) ) {
        case R_TYPE_TYPEDEF:
            if( strcmp(r_string_bytes(r_type_name(type, NULL)), "string_t") == 0 )
                printf("(string_t) \"%s\"\n", *((const string_t *) data));
            break;
        case R_TYPE_BUILTIN:
            switch( r_builtin_type_id((RBuiltinType *) type, NULL) ) {
                case R_BUILTIN_TYPE_INT:
                    printf("(int) %d\n", *((const int *) data));
                    break;
            }
            break;
        case R_TYPE_AGGREGATE: {
            RAggregateType *agg = (RAggregateType *) type;
            if( r_aggregate_type_id(agg, NULL) == R_AGGREGATE_TYPE_STRUCT ) {
                printf("(%s) {\n", r_string_bytes(r_type_name(type, NULL)));
                for( size_t i = 0; i < r_aggregate_type_nmembers(agg, NULL); i++ ) {
                    RAggregateMember *memb = r_aggregate_type_member_at(agg, i, NULL);
                    printf("\t.%s = ", r_string_bytes(r_aggregate_member_name(memb, NULL)));
                    RTypeMember *tmemb = (RTypeMember *) memb;
                    off_t offset = r_type_member_offset(tmemb, NULL);
                    print_data(r_type_member_type(tmemb, NULL), data + offset);
                }
                printf("}\n");
            }
            break;
        }
    }
}

int main( int argc, char *argv[] ) {
    (void) argc;
    ruminate_init(argv[0], NULL);
    struct foo bar = {
        .str = "Hello World!",
        .i = 6666
    };
    print_data(ruminate_get_type(bar, NULL), &bar);
}
```

Ruminate

But that's too large to fit.

Ruminate

Oh wait

Ruminate

Oh wait
Lets not forget

Ruminate

Oh wait
Lets not forget
You can do stack introspection too

Ruminate

Stack Introspection

```c
// Includes go here
void print_backtrace() {
    RFrameList *frames = ruminate_backtrace(NULL);
    size_t frames_len = r_frame_list_size(frames, NULL);
    for( size_t i = 0; i < frames_len; i++ )
        printf("%s\n", r_string_bytes(r_frame_function_name(
                       r_frame_list_at(frames, i, NULL), NULL)));
}
void bar( int i ) { if( i < 2 ) bar(i + 1); else print_backtrace(); }
void foo() { bar(0); }
int main( int argc, char *argv[] ) {
    ruminate_init(argv[0], NULL);
    foo();
}
```

Ruminate

Stack Introspection

```c
// Includes go here
void print_backtrace() {
    RFrameList *frames = ruminate_backtrace(NULL);
    size_t frames_len = r_frame_list_size(frames, NULL);
    for( size_t i = 0; i < frames_len; i++ )
        printf("%s\n", r_string_bytes(r_frame_function_name(
                           r_frame_list_at(frames, i, NULL), NULL)));
}
void bar( int i ) { if( i < 2 ) bar(i + 1); else print_backtrace(); }
void foo() { bar(0); }
int main( int argc, char *argv[] ) {
    ruminate_init(argv[0], NULL);
    foo();
}
```

```
ruminate_hit_breakpoint
ruminate_backtrace
print_backtrace
bar
bar
bar
foo
main
__libc_start_main
```

Ruminate

Typed Memory Allocator

Building on this, you can do typed dynamic memory allocation.

Ruminate

Typed Memory Allocator

Building on this, you can do typed dynamic memory allocation.

```c
// Includes go here
int main( int argc, char *argv[] ) {
    ruminate_init(argv[0], NULL);
    const char *src_str = "Hello World!";
    size_t src_str_len = strlen(src_str) + 1;
    void *str = r_mem_malloc_sized(
        char *, src_str_len, NULL
    );
    memcpy(str, src_str, src_str_len);
    printf("(%s) \"%s\"\n", r_string_bytes(r_type_name(
        r_mem_type(str),
        NULL
    )), str);
}
```

```
(char *) "Hello World!"
```

Ruminate

JSON

and do transparent json conversion

Ruminate

JSON

and do transparent json conversion

both unidirectionally

```c
// Includes go here
struct MyStruct {
    int i;
    enum MyEnum { MY_ENUM_VALUE_1, MY_ENUM_VALUE_2 } e;
};
int main( int argc, char *argv[] ) {
    ruminate_init(argv[0], NULL);
    struct MyStruct foo = { 1, MY_ENUM_VALUE_2 };
    json_dumpf(
        json_serialize(NULL, ruminate_get_type(foo, NULL),
                             &foo, NULL),
        stdout, 0
    );
}
```

```
{"i": 1, "e": 1}
```

Ruminate

JSON

and bidirectionally

```c
// Includes go here
struct MyStruct {
    int i;
    enum MyEnum { MY_ENUM_VALUE_1, MY_ENUM_VALUE_2 } e;
};
int main( int argc, char *argv[] ) {
    ruminate_init(argv[0], NULL);
    struct MyStruct foo = { 1, MY_ENUM_VALUE_2 };
    JsonState *js = json_state_new();
    json_state_set_flags(js, JSON_FLAG_INVERTABLE);
    json_dumpf(
        json_serialize(js, ruminate_get_type(foo, NULL),
                       &foo, NULL),
        stdout, 0
    );
}
```


```
{"value": {"i": 1, "e": 1}, "type": "MyStruct"}
```

Implementation

Lets dive in

Implementation

RType

```
struct RType {
    RTypeId id;
    gint refcnt;
    RString *name;
    RuminateBackend::TypePrx type;
    RuminateBackend::TypeId type_id;
    struct {
        size_t value;
        bool initialized;
    } size;
};
```

Implementation

RType

```
struct RType {
    RTypeId id;
    gint refcnt;
    RString *name;
    RuminateBackend::TypePrx type;
    RuminateBackend::TypeId type_id;
    struct {
        size_t value;
        bool initialized;
    } size;
};
```

Aside from caching, `RType` is just a proxy backed by `RuminateBackend::TypePrx`

RPC Contract

- Simple, minimal RPC interface
- Built off the `Type` interface
- Should be simple enough to port to other debuggers

RPC Contract

- Simple, minimal RPC interface
- Built off the Type interface
- Should be simple enough to port to other debuggers

```
interface Type {
  TypeId getId();
  Type *getPointeeType();
  Type *getPointerType();
  Type *getCanonicalType();
  Type *getReturnType();
  Type *getArrayMemberType();
  idempotent long getArrayLength();
  TypeMemberList getMembers( optional(1) long tid );
  string getName();
  idempotent long getSize();
  idempotent bool isSigned();
  idempotent bool isUnsigned();
};
```

Implementation

RType

`RType` and it's decendants:

- type safe
- caching
- reference counted
- wrapper around `Type`

Implementation

Debugger Server

Manipulate LLDB to implement the RPC contract

Implementation

Debugger Server

Manipulate LLDB to implement the RPC contract

Some interesting bits

- Enums
- Signals
- Arrays

Implementation
LLDB

- DWARF
- ELF, Mach-O
- Name demangling
- Stack traversal

Implementation

Enums

LLDB's public API did not support enum introspection. Support was added.

Implementation

Enums

LLDB's public API did not support enum introspection. Support was added.

Enum members are interesting

Implementation

Enums

LLDB's public API did not support enum introspection. Support was added.

Enum members are interesting

the only type in C whose value is part of the type

```
enum MyEnum {
    MY_ENUM_1,
    MY_ENUM_2
};
```

Implementation

Signals

> *While being traced, the tracee will stop each time a signal is delivered -ptrace(2)*

LLDB default behavior is to stop on (most) signals and wait for user input...

Implementation

Signals

> *While being traced, the tracee will stop each time a signal is delivered -ptrace(2)*

LLDB default behavior is to stop on (most) signals and wait for user input...
there is no user

Implementation

Signal Deadlock

If a signal occurrs, LLDB will wait for the debugee to provide instructions. The debugee will remain stopped.

Implementation

Signal Deadlock

If a signal occurrs, LLDB will wait for the debugee to provide instructions. The debugee will remain stopped.
change LLDB's signal disposition to ignore signals
No support for this in LLDB. Support was added.

Implementation

Arrays

LLDB does not model arrays in it's pure type representation.

Implementation

Arrays

LLDB does not model arrays in it's pure type representation.
Arrays are accessed by value only.

Implementation

Arrays

LLDB does not model arrays in it's pure type representation.
Arrays are accessed by <u>value</u> only.

```
char a[sizeof((*((int (*)[4]) NULL))[0])];
__typeof__(&a) ap = &a;
(int (*)[4]) ap
```

Implementation

Typed Memory

Typed memory is implemented by padding the beginning of allocated memory with a pointer to the type.

Implementation

JSON

Conversion to JSON is simple

```
if the type is a struct
  generate a JSON object
  for each subtype
    recurse
else if the type is an enum
  generate a JSON integer
else if the type is an array
  generate a JSON array
  recurse
else if the type is a pointer
  recurse using the dereferenced pointer
else
  generate a JSON primitive
```

Note that unions and strings are not handled.

Implementation

JSON

Conversion from json is similar

```
retrieve the name of the type from the top-level json object
lookup the type in the current program
allocate typed memory
proc repeat
  if the type is a struct
    retrieve the JSON object representing this value
    for each subtype
      call repeat
  else if the type is an enum
    insert the value of the JSON int
  else if the type is an array
    retrieve the JSON array representing this value
    call repeat
  else if the type is a pointer
    allocate typed memory
    call repeat
  else
    insert the value from the JSON primitive
```

Limitations

- strings are not detectable
- the current type of a union is not detectable
- slow
- debugging symbols are required

Limitations

Strings

strings are not detectable

Limitations

Strings

strings are not detectable

In C, a string has type **`char *`**. That is, it's type is "pointer to byte".

With no additional information it is <u>unknowable</u> if this **`char *`** type is

- a valid `NULL` terminated string
- a pointer to a single character
- a pointer to multiple characters which is non `NULL` terminated
- a pointer to one or more bytes which are not valid characters in the current encoding

Limitations

Unions

The current type of a union is not detectable

Limitations

Unions

The current type of a union is not detectable

```
1  union foo { int a; char b; }
2  int main( int argc ) {
3      union foo f;
4      if( argc > 2 ) {
5          f.a = 1;
6      } else {
7          f.b = 'c';
8      }
9      // !!
10 }
```

- The compiler cannot know the type of `f` at line 9 due to the indeterminate nature of `argc`
- The debugger cannot know the type of `f` at line 9 because DWARF does not (and cannot) contain enough information for the debugger to determine the current type of `f`

Limitations

Slow

Ruminate imposes significant overhead.

```c
// Includes go here
int main( int argc, char *argv[] ) {
    ruminate_init(argv[0], NULL);
    const char *src_str = "Hello World!";
    size_t src_str_len = strlen(src_str) + 1;
    void *str = r_mem_malloc_sized(
        char *, src_str_len, NULL
    );
    memcpy(str, src_str, src_str_len);
    printf("(%s) \"%s\"\n", r_string_bytes(r_type_name(
        r_mem_type(str),
        NULL
    )), str);
}
```

Takes 1.6 seconds to run.

Limitations

Debugging Symbols

Debugging symbols are required

Limitations

Debugging Symbols

Debugging symbols are required
Not a limitation

Limitations

Debugging Symbols

Debugging symbols are required

Not a limitation

DWARF debugging symbols are made up of two components:

- line number information
- type information

all of which is available in every other language that has introspection.

Limitations

Debugging Symbols

Debugging symbols are required
Not a limitation
DWARF debugging symbols are made up of two components:

- line number information
- type information

all of which is available in every other language that has introspection.
Debugging symbols are only required in modules being introspected.

Future Work

slow is a problem

Future Work

slow is a problem
eliminate the debugger controller

Future Work

slow is a problem
eliminate the debugger controller
embed the debugger inside the debugee

Future Work

slow is a problem
eliminate the debugger controller
embed the debugger inside the debugee
how?

Future Work

what makes a debugger?

Future Work

what makes a debugger?

- remote process controller
- remote memory inspector
- executable parser (ELF, Mach-O, etc)
- symbol parser
- stack traverser

Future Work

what does Ruminate need?

Future Work

what does Ruminate need?

- ~~remote process controller~~
- ~~remote memory inspector~~
- executable parser (ELF, Mach-O, etc)
- symbol parser
- stack traverser

Future Work

how?

Future Work

how?

split LLDB's core into several components

- symbol parser
- stack traverser

useful for more than just Ruminate

Thanks