Jove

Dynamic Analysis via Decompilation/Recompilation

Anthony Eden

Overview

- Goals
  - Faithful decompile/recompile of the program
  - Utilize compiler tools for dynamic analysis on binaries
  - Limit complexity on emulations
- Challenges
  - Very difficult to find code. Static approaches are not able to find 100% of the code
  - Accurate decompile/recompile is challenging
  - Reasonable analysis time
  - Efficient runtime

Jove Approach

- Dynamically monitor the program to find basic blocks
- Iterate over inputs to find an initial code base
- Decompile code back to LLVM IR using QEMU TCG emulation
  - Enables multiple instruction sets
  - Emulation code can be used as the basis for the LLVM IR
- Recompile with desired instrumentation (e.g., DFSan)
  - Include traps at each indirect jump to discover any new code
- When new code is found
  - Augment the code base
  - Recompile
  - Rerun on the same inputs
  - Works well with our use case

Jove-init / Jove-add

- Process the binary and each of the libraries used by the binary
- Recursively explores all of the entry points for each
- Builds an initial list of basic blocks
- Stores the result in a .jv file for the executable

Jove-bootstrap

- Adds traps to each indirect jump in the current basic block list
- Each time a trap is triggered with new code
  - Recursively follow the new entry point
  - Add the new basic blocks to the code base
  - Continue execution.
- Results are stored back to the .jv file. Process supports multiple runs with different inputs.
- Supports incomplete/corrupted ELF files (which are common)

Jove-llvm - Conversion from binary to LLVM-IR

- Based on QEMU translation to TCG
- TCG translation to LLVM-IR
- Creates an ELF file that contains the recompiled code and all of the original ELF file contents.
- The LLVM-IR interacts with an emulated stack
  - Largely matches the original program’s native stack (parameters, local variables, etc)
  - Return pointers are on the native stack
  - Registers in the binary are local variables in the recompiled version.

Jove-llvm - Addresses

- Addresses of code/data are original addresses (perhaps with an offset)
- An address-translation-table maps original code addresses to recompiled code addresses
- At each indirect jump/call the address is looked up in address-translation-table

Jove-llvm - native interactions with recompiled code.

- Some code is not recompiled (e.g., dynamic linker)
- Recompiled code calls native (jove-thunk)
  - Real stack pointer is saved, then set to be the value of the emulated stack pointer.
  - Code is called
  - Upon return, emulated stack pointer is set to the real stack pointer. Real stack pointer is then restored to saved value.
  - Any calling-convention-specific transformations on the stack pointer are reflected in the emulated stack pointer after the call returns.
- Native code calls recompiled code (jove-inverse-thunk)
  - Exported symbols from recompiled code are set to cause a segfault
  - In the signal handler, emulated stack pointer is saved; then set to be the value of the real stack pointer. Real stack pointer is set to a newly allocated stack
  - Code is called
  - Upon return, real stack pointer is set to emulated stack pointer; emulated stack pointer is restored to saved value; newly allocated stack is freed.

System State Optimization

- Original TCG IR uses a global system state structure
- Jove uses a local system state structure
  - Allows much better code generation
- QEMU helper functions
  - Called from TCG
  - Interact with the global system state
  - Jove analyzes these functions to determine their system state interactions
  - Changes to the system state are reflected back in the caller
  - Approach allows helper functions to be inlined as well.

IR Example

```
typedef long complex_part_t;

struct complex_t {
  complex_part_t real;
  complex_part_t imag; };

struct complex_t cn_add(struct complex_t a,
                        struct complex_t b) {
  struct complex_t c;
  c.real = a.real + b.real;
  c.imag = a.imag + b.imag;
  return c;
}
```

Decompile/Recompile IR Example

```
; Function Attrs: noinline norecurse nounwind readnone uwtable
define dso_local { i64, i64 } @cn_add(i64 %0, i64 %1, i64 %2, i64 %3) local_unnamed_addr #4 {
  %5 = add nsw i64 %2, %0
  %6 = add nsw i64 %3, %1
  %7 = insertvalue { i64, i64 } undef, i64 %5, 0
  %8 = insertvalue { i64, i64 } %7, i64 %6, 1
  ret { i64, i64 } %8
}

define internal {i64, i64, i64} @j201ab0(i64 %rdi, i64 %rsi, i64 %rdx, i64 %rcx, i64 %rsp) #12 {
  "0x201ab0":
  %0 = add i64 %rsp, 8
  %1 = add i64 %rcx, %rsi
  %2 = add i64 %rdx, %rdi
  % returning rax = insertvalue {i64, i64, i64} undef, i64 %2, 0
  % returning rdx = insertvalue { i64, i64, i64 } % returning rax , i64 %1, 1
  % returning rsp  = insertvalue {i64, i64, i64}  %_returning_rdx_, i64 %0, 2
  ret { i64, i64, i64 } %_returning_rsp_
}
```

Jove-llvm details

- Thread Pointer Offset (TPOFF) relocations can’t be expressed in LLVM constant expressions and are thus computed at runtime.
- Support for GNU `ifuncs`
- GLIBC’s `__libc_early_init`
- On X386 translate call `dword ptr gs:[16]` as a syscall
- PC relative expressions (not directly expressible in LLVM-IR)
- Support “twirl” - calling the next instruction to get the program counter
- Delay slots on MIPs (unlike jumps, traps don’t execute delay slots)

Code Extractor

- The QEMU emulator has a lot of helper functions that are called at runtime
- Those same helper functions are used from the recompiled code
- The helper functions are intertwined with QEMU
  - Difficult to pull out exactly what we want
  - QEMU is much too big to include as a whole
- Extracts a function by name
  - Creates a minimal stand-alone source file for the function without includes
  - Contains any macros or type/structure definitions that are referenced
  - Contains any functions called by the original function
  - Nothing else is included.
- Utilizes a Clang plugin to extract the source information

Helper Example

```
uint32_t HELPER(glue(PFX,add16))(uint32_t a,
                uint32_t b GE_ARG)
{
    uint32_t res = 0;
    DECLARE_GE;

    ADD16(a, b, 0);
    ADD16(a >> 16, b >> 16, 1);
    SET_GE;
    return res;
}
```

```
#define xglue(x, y) x ## y

#define glue(x, y) xglue(x, y)

#include <stdint.h>

#define HELPER(name) glue(helper_, name)

#define ADD16(a, b, n) do { \
    uint32_t sum; \
    sum = (uint32_t)(uint16_t)(a) + (uint32_t)(uint16_t)(b); \
    RESULT(sum, n, 16); \
    if ((sum >> 16) == 1) \
        ge |= 3 << (n * 2); \
    } while(0)

#define PFX u

#define GE_ARG , void *gep

#define DECLARE_GE uint32_t ge = 0

#define SET_GE *(uint32_t *)gep = ge

#define RESULT(val, n, width) \
    res |= ((uint32_t)(glue(glue(uint,width),_t))(val)) << (n *
width)
```

Handling New Code (jove-loop and jove-run)

- New code paths may be executed
  - New inputs
  - Same inputs may still have non-deterministic results
- At each indirect jump/call
  - Address is looked up in the address-translation table
  - New addresses will terminate the program
- The new address is captured and new basic blocks are recursively traversed
- The program is recompiled with the addition of the new blocks
- The program is run again over the same inputs
  - Command lines, environment variables, pipes, etc.
- Process is repeated until the program does not encounter any new code.

Modes of operation

- Library mode
  - The executable and almost all libraries are recompiled.
  - Each library has a recompiled version
  - Dynamic linker is always native
- Executable only mode
  - Only the executable itself is recompiled
  - All libraries are left as native code
  - Advantages
    - Simpler problem (library code is often more complex and hand optimized)
    - No need to manage multiple library versions
  - Disadvantages
    - Critical code is sometimes in libraries
    - Instrumentation code (e.g., DFSan) is only applied to the executable. Taint may be lost or more complex summaries have to be used.

Status

- Supports 386, X86-64, MIPS, and ARM64
- Single threaded programs
- Reasonable sized programs working correctly (eg, nginx, Trendnet and Netgear httpd)