Binary Code Analysis: Techniques, Tools, and Applications

Lecture 3: Dynamic Taint Analysis

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

Outline

Outline

1. Basic Concepts
2. Taint Analysis Design
3. Taint Analysis Implementation
4. Summary

Data Flow Analysis

Definition

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.

Data Flow Analysis

Definition

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program.

Basic Approaches

A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint.

Dynamic data flow tracking (DFT)

What is it?

Tagging and **tracking** "interesting" data as they propagate during program execution

Dynamic data flow tracking (DFT)

What is it?

Tagging and **tracking** "interesting" data as they propagate during program execution

- Extremely popular research topic (also known as information flow tracking)
  - Analyzing malware behavior [Portokalidis Eurosys'06]
  - Hardening software against zero-day attacks [Bosman RAID'11, Qin MICRO'06, Newsome NDSS'05]
  - Detecting and preventing information leaks [Zhu SIGOPS'11, Enck OSDI'10]
  - Debugging software misconfigurations [Attariyan OSDI'10]

A large body of research in DFT

Architectural classification

- Integrated into full system emulators and virtual machine monitors [Ho Eurosys'06, Portokalidis Eurosys'06, Myers POPL'99]
- Retrofitted into unmodified binaries using dynamic binary instrumentation (DBI) [Qin et al. MICRO'06]
- Added to source codebases using source-to-source code transformations [Xu et al. USENIX Sec'06]
- Implemented in hardware [Venkataramani HPCA'08, Crandall MICRO'04, Suh ASPLOS'04]

Attempts for Flexible DFT

- TaintCheck [Newsome and Song, NDSS'05] → 20x overhead even for small utilities
- LIFT [Qin et al. Micro'06] → no multithreading support
- Dytan [Clause et al. ISSTA'07] → attempts to define a generic and reusable DFT framework, but incurs a slowdown of more than 30x
- Minemu [Boseman et al, RAID'11] → only 32-bit binaries
- Libdft [Kemerlis et al., VEE'12] → faster (1.14 to 10x slowdown), and reusable, applicable to commodity hardware and software

DFT

Formalisms

- Many aliases
  - Data flow tracking (DFT)
  - Information flow tracking (IFT)
  - Dynamic taint analysis (DTA)
  - ...

DFT

Formalisms

- Many aliases
  - Data flow tracking (DFT)
  - Information flow tracking (IFT)
  - Dynamic taint analysis (DTA)
  - ...

Definition

The process of **accurately** tracking the flow of **selected** data throughout the execution of a program or system

DFT

Explicit vs. implicit data flows

```
1: unsigned char csum = 0;
2:
3: bcount = read(fd, data, 1024);
4: while(bcount-- > 0)
5:       csum ^= *data++;
6:
7:  write(fd, &csum, 1);
```

(a) Data flow dependency

```
1: int authorized = 0;
2:
3: bcount = read(fd, pass, 12);
4: MD5(pass, 12, phash);
5: if (strcmp(phash, stored_hash) == 0)
6:     authorized = 1;
7: return authorized;
```

(b) Control flow dependecy

Figure: Examples of code with explicit and implicit data dependencies

Outline

Basic Components in a DFT

Three key components

1. **Taint Sources**: program, or memory locations, where data of interest enter the system and subsequently get tagged

Basic Components in a DFT

Three key components

1. **Taint Sources**: program, or memory locations, where data of interest enter the system and subsequently get tagged
2. **Taint Tracking**: process of propagating data tags according to program semantics

Basic Components in a DFT

Three key components

1. **Taint Sources**: program, or memory locations, where data of interest enter the system and subsequently get tagged
2. **Taint Tracking**: process of propagating data tags according to program semantics
3. **Taint Sinks**: program, or memory locations, where checks for tagged data can be made

Data Flow Facts (e.g., taint record)

Definition

Data flow facts concerns the information about the data flow of interest. For instance, it could be the liveness of the variables, could be the reach definitions, and could be the taint of certain information.

Where to store the data flow facts?

Shadow Memory

Shadow Memory

Shadow memory describes a computer science technique in which potentially every byte used by a program during its execution has a shadow byte or bytes.

Shadow Memory

Shadow Memory

Shadow memory describes a computer science technique in which potentially every byte used by a program during its execution has a shadow byte or bytes.

These shadow bytes are typically invisible to the original program and are used to record information about the original piece of data.

Shadow Memory

Shadow Memory

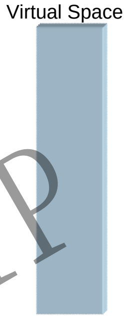
Shadow memory describes a computer science technique in which potentially every byte used by a program during its execution has a shadow byte or bytes.

These shadow bytes are typically invisible to the original program and are used to record information about the original piece of data.

The program is typically kept unaware of the existence of shadow memory by using a dynamic binary translator/instrumentor, which, among other things, may translate the original programs memory read and write operations into operations that do the original read and write and also update the shadow memory as necessary.

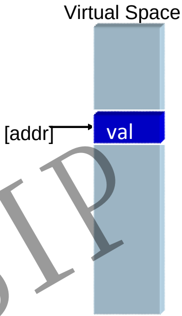
Store Abstract State

- Shadow memory
  – We need a mapping
    - Addr → Abstract State
    - Register → Abstract



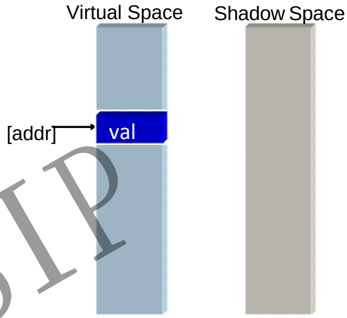
Store Abstract State

- Shadow memory
  – We need a mapping
  - Addr → Abstract State
  - Register → Abstract



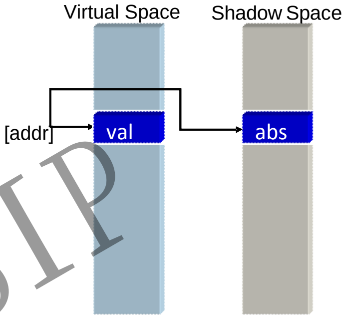
Store Abstract State

- Shadow memory
  – We need a mapping
    - Addr → Abstract State
    - Register → Abstract



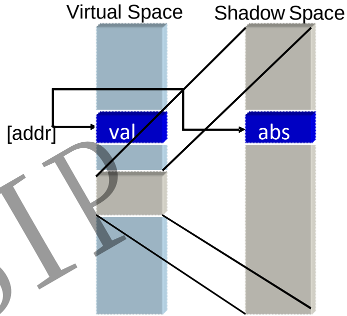
Store Abstract State

- Shadow memory
  – We need a mapping
    - Addr → Abstract State
    - Register → Abstract



Store Abstract State

- Shadow memory
  – We need a mapping
    - Addr → Abstract State
    - Register → Abstract

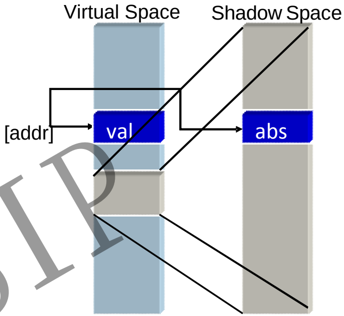


Store Abstract State

```
typedef
   struct {
      UChar abits[65536];
}   SecMap;

static SecMap* primary_map[65536];
static SecMap  default_map;

static void init_shadow_memory (void)
{
  for (i = 0; i < 65536; i++)
      default_map.abits[i] = 0;
  for (i = 0; i < 65536; i++)
      primary_map[i] = &default_map;
}
static SecMap* alloc_secondary_map()
{
   map=malloc(sizeof(SecMap));
   for (i = 0; i < 65536; i++)
          map->abits[i] = 0;
   return map;
}
void Accessible (addr)
{
   if (primary_map[(addr) >> 16]
       == default_map)
       primary_map[(addr) >> 16] =
          alloc_secondary_map(caller);
}
```



An Example of Using Taint Analysis

Goal

Recover the data structure type information, from program binary code

Basic Techniques

Collecting the type constraints, and solve and unify them statically or dynamically (during the program execution).

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118	●	N/A

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118	●	N/A
eax	●	

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		N/A
eax		

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118	●	N/A
eax	●	
0x4(%esp)	●	

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,    %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		N/A
eax		
0x4(%esp)		
0x8049128		N/A

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		N/A
eax		
0x4(%esp)		
0x8049128		N/A
(%esp)		

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		N/A
eax		
0x4(%esp)		
0x8049128		N/A
(%esp)		

(esp+4) → char*
(esp) → char*
strcpy(char*, char*)

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		N/A
eax		
0x4(%esp)		char*
0x8049128		N/A
(%esp)		char*

(esp+4) → char*
(esp) → char*
strcpy(char*, char*)

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,    %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		char*
eax		char*
0x4(%esp)		char*
0x8049128		char*
(%esp)		char*

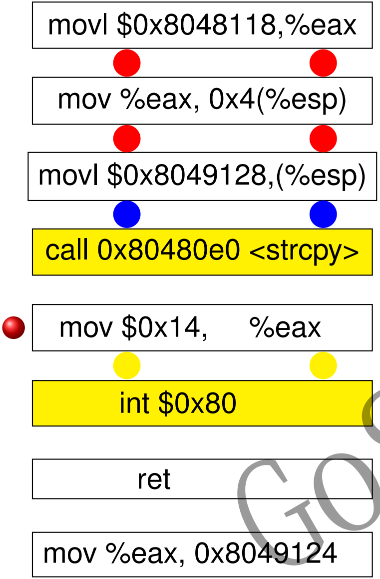
(esp+4) → char*
(esp) → char*
strcpy(char*, char*)

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		char*
eax		char*
0x4(%esp)		char*
0x8049128		char*
(%esp)		char*

Example: Data Flow Based Type Resolution



Mem,Reg	Tag	Type
0x8048118		char*
eax		imm_t
0x4(%esp)		char*
0x8049128		char*
(%esp)		char*

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,    %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		char*
eax		imm_t
0x4(%esp)		char*
0x8049128		char*
(%esp)		char*

getpid   eax → pid_t

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118	red	char*
eax	green	pid_t
0x4(%esp)	red	char*
0x8049128	blue	char*
(%esp)	blue	char*

getpid eax → pid_t

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,     %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118	red	char*
eax	green	pid_t
0x4(%esp)	red	char*
0x8049128	blue	char*
(%esp)	blue	char*

Example: Data Flow Based Type Resolution

```
movl $0x8048118,%eax
mov %eax, 0x4(%esp)
movl $0x8049128,(%esp)
call 0x80480e0 <strcpy>
mov $0x14,    %eax
int $0x80
ret
mov %eax, 0x8049124
```

Mem,Reg	Tag	Type
0x8048118		char*
eax		pid_t
0x4(%esp)		char*
0x8049128		char*
(%esp)		char*
0x8049124		pid_t

Outline

Using Valgrind

```
UCodeBlock* SK_(instrument)(UCodeBlock* cb_in, Addr orig_addr)
{
   UCodeBlock* cb;
   ...
   switch (u_in->opcode) {
       case LOAD:
              VG_(ccall_RR_R) (cb, (Addr) HELPER_bdd_load ,
                     u_in->val1, SHADOW(u_in->val1), SHADOW(u_in->val2), 2);
              break;
                ...
    }
}
bdd HELPER_bdd_load(Addr a, bdd addr_bdd)
{
       bdd mem_bdd = get_ii_vbytes4_ALIGNED(a);
       bdd_allsat (mem_bdd,allsatPrintHandler);
       return mem_bdd;
}
...
```

Using QEMU

```
static target_ulong disas_insn(DisasContext *s, target_ulong pc_start)
{
        /* inc, dec, and other misc arith */
    case 0x40 ... 0x47: /* inc Gv */
        ot = dflag ? OT_LONG : OT_WORD;
        gen_inc(s, ot, OR_EAX + (b & 7), 1);
        break;
    case 0x48 ... 0x4f: /* dec Gv */
        ot = dflag ? OT_LONG : OT_WORD;
        gen_inc(s, ot, OR_EAX + (b & 7), -1);
        break;
    case 0x134: /* sysenter */
        gen_helper_sysenter();
        break;
}
static void gen_inc(DisasContext *s1, int ot, int d, int c)
{
    if (d != OR_TMP0)
        gen_op_mov_TN_reg(ot, 0, d);
    else
        gen_op_ld_T0_A0(ot + s1->mem_index);
}
void helper_sysenter(void)
{
    ESP = env->sysenter_esp;
    EIP = env->sysenter_eip;
}
...
```

Using PIN

```
main()
{
          ...
     INS_AddInstrumentFunction(SetupDataflow, 0);
     setup_inst_hook();
     ...
}
void SetupDataflow(INS ins, void *v)
{
  xed_iclass_t opcode = (xed_iclass_t) INS_Opcode(ins);
  (*instrument_functions[opcode])(ins, v);
}
void setup_hook(){
    for(int i = 0; i < XED_ICLASS_LAST; i++) {
        instrument_functions[i] = &UnimplementedInstruction;
    }
   instrument_functions[XED_ICLASS_ADD] = &Instrument_ADD;
}
static void Instrument_MOV(INS ins, void *v)
{
//1. R -> R | M
if(INS_OperandIsReg(ins, 1)) {
    INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(GetRegTag),
           IARG_ADDRINT, INS_OperandReg(ins, 1),
           IARG_PTR, &reg_tag_src,
           IARG_END);
}
```

Pin-based `libdft`

Goal

Shared library for customized DFT

Allow the creation of "meta-tools" that transparently employ DFT

Pin-based `libdft`

Goal

Shared library for customized DFT

Allow the creation of "meta-tools" that transparently employ DFT

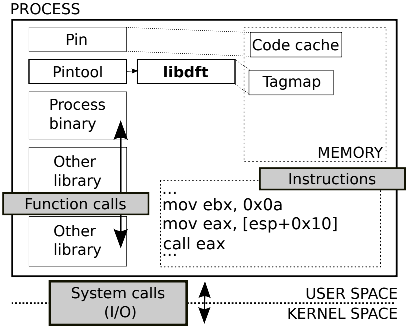


Figure: Putting it altogether: Pin, libdft, process

Usage

`libdft` in a nutshell

1. Pin loads itself, `libdft`, and a `libdft`-enabled tool into the same address space with a process
2. Before commencing or resuming execution, the `libdft`-tool defines the data sources and sinks by tapping arbitrary points of interest
3. User-defined callbacks drive the DFT process by tagging and untagging data, or checking and enforcing data use

Challenges

Achieving low overhead is hard

- Size & structure of the analysis routines (i.e., DFT logic) matters
- Complex analysis code → excessive register spilling
- Certain types of instructions should be avoided altogether (e.g., test-and-branch, EFLAGS modifiers)

libdft

Prototype implementation

- libdft has been implemented using Pin v2.9
- Currently supports only x86 Linux binaries
- Consists of three main components (Figure 2)
  1. Tagmap
  2. Tracker
  3. I/O interface
- $\sim$ 5000 LOC in C/C++

libdft
Architecture

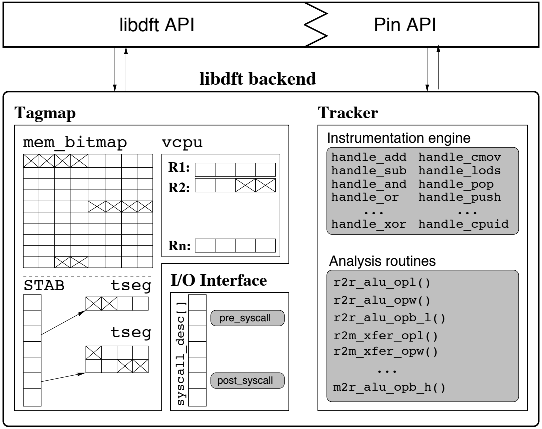


Figure: The architecture of libdft

libdft

Tagmap

- Stores the tags for every process
- Major impact on the overall performance → DFT logic constantly operates on data tags
- Tag format
  - Tagging granularity → byte
  - Tag size → {1,8}-bit
- Register tags
  - Per thread vcpu structure
  - 8 general purpose registers (GPRs)
- Memory tags
  - Per process mem bitmap, or STAB and tseg structures
  - 1 bit/byte for every byte of addressable memory

`libdft`

Tracker

- Instruments a program for retrofitting the DFT logic
- **Instrumentation Engine**
  - Invoked once for each sequence of instructions
  - Handles the elaborate logic of discovering data dependencies → allows for compact and fast analysis code
  - Inspects the instructions of a program
  - Determines the analysis routines that should be injected before each instruction
  - Allows for customization (`libdft` API)
- **Analysis Routines**
  - Invoked every time a specific instruction is executed
  - Contain code that implements the DFT logic
  - Clear, assert, and propagate tags

`libdft`

I/O Interface

- Handles the kernel $\leftrightarrow$ process data
- `pre_syscall`/`post_syscall` $\rightarrow$ **instrumentation stubs**
- `syscall_desc[]` $\rightarrow$ syscall **meta-information table**
- The stubs are invoked upon every system call entry/exit
- Allows the user to register callback functions (libdft API)
- The default behavior of the `post_syscall` stub is to untag the data being written/returned by the system call

Advantages

- Enables the *customization* of libdft by using I/O system calls as data sources and sinks arbitrarily
- *Eliminates* tag leaks by considering that some system calls write specific data to user-provided buffers

libdft

Optimizations

- `fast_vcpu` Uses a scratch-register to store a pointer to the `vcpu` structure of each thread
- `fast_rep` Avoids recomputing the effective address (EA) on each repetition in `REP`-prefixed instructions
- `huge_tlb` Uses huge pages for `mem_bitmap` and `STAB` to minimize TLB poisoning
- `tagmap_col` Collapses `tseg` structures that correspond to write-protected memory regions to a single constant segment

`libdft`-DTA

Taint analysis made easy

- libdft offers a small and elegant API for *transparently* incorporating DFT into running applications $\rightarrow$ can we use it in order to enforce security policies?
- Built a full-fledged DTA tool in $\sim$ 450 LOC that protects against *code injection* attacks (*e.g.*, stack smashing, heap corruption) *memory overwrite* attacks (*e.g.*,`return-to-libc`, format string) *etc.*
- $+7\%$ additional runtime overhead
- Tested with real exploits

Dynamically retrofit DTA capabilities into running applications
$\rightarrow$ Binary inline reference monitor

Outline

Summary

Key steps in designing DFT/DTA

1. Designing shadow memory
2. Instrument each instruction
3. Generate or propagate data flow facts
4. Query data flow facts

An example of DFT/DTA: `libdft`

- **Fast** (highly optimized Tracker)
  - branch-less tag propagation
  - single assignment tagmap updates
  - inlined DFT logic
- **Reusable**(API)
  - customizable propagation policy
  - assignment of data sources and sinks at arbitrary points of interest
- Applicable to **commodity hardware and software**
  - multiprocess and multithreading support
  - no modifications to the binaries or the underlying OS
- `www.cs.columbia.edu/~vpk/research/libdft/`

Pin DBI

- `libdft` relies on Pin [Luk PLDI'05] for instrumenting and analyzing the target process
- **Instrumentation** → what analysis routines should be inserted where
- **Analysis routines** → code that is dynamically injected into the application and augments its execution
- Pin uses a *JIT* compiler for combining the original code, `libdft`, and the code of a `libdft`-tool
- "Jitted" code is placed into a code cache for avoiding re-translation

Binary Code Analysis: Techniques, Tools, and Applications

Lecture 4: Symbolic Execution

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

Outline

Outline

1. Background
2. Enabling Techniques
   - SAT Solving
   - Data Flow Tracking
3. Applications
   - Software Testing
   - Whitebox Fuzzing
   - Program Understanding/Reverse Engineering
4. Summary

Symbolic Execution

Symbolic execution (also symbolic evaluation) is a means of analyzing a program to determine what inputs cause each part of a program to execute.

"An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch."

https://en.wikipedia.org/wiki/Symbolic_execution

Symbolic Execution

- "Symbolic execution and program testing", [King 1976]
- Analysis of programs with unspecified inputs
  - Execute a program on symbolic inputs
- Symbolic states represent sets of concrete states
- **Insight**: code can generate its own test cases

Example

```
y = read()
y = 2 * y
if (y == 12)
   fail ()
printf ("OK")
```

Assume the goal of the analysis is to determine what inputs cause the "fail()" statement to execute. The analyzer uses a constraint solver to determine what values of input y make ’2 * y == 12’ true, and thus determines that ’6’ is the answer.

Concolic Execution: better scalability

Combine **concrete** and **symbolic execution**:

- Concrete + Symbolic = Concolic

- Use concrete execution over a concrete input to guide symbolic execution
- Concrete execution helps Symbolic execution to simplify complex and unmanageable symbolic expressions (by replacing symbolic values by concrete values)

Tools

Tool	Arch/Lang	url	Available?
KLEE	LLVM	http://klee.github.io/	yes
FuzzBALL	VineIL/native	http://bitblaze.cs.berkeley.edu/fuzzball.html	yes
JPF	java	http://babelfish.arc.nasa.gov/trac/jpf	yes
jCUTE	java	https://github.com/osl/jcute	yes
janala2	java	https://github.com/ksen007/janala2	yes
KeY	java	http://www.key-project.org/	yes
S2E	llvm/qemu/x86	http://dslab.epfl.ch/proj/s2e	yes
Pathgrind	native 32bit valgrind based	https://github.com/codelion/pathgrind	yes
Mayhem	binary	http://forallsecure.com/mayhem.html	no
Otter	C	https://bitbucket.org/khooyp/otter/overview	yes
SymDroid	Dalvik bytecode	http://www.cs.umd.edu/ jfoster/papers/symdroid.pdf	no
Rubyx	Ruby	http://www.cs.umd.edu/ avik/papers/ssarorwa.pdf	no
Pex	.NET Framework	http://research.microsoft.com/en-us/projects/pex/	no
Jalangi	JavaScript	https://github.com/SRA-SiliconValley/jalangi	yes
Kite	llvm	http://www.cs.ubc.ca/labs/isd/Projects/Kite/	yes
pysymemu	amd64/native	https://github.com/feliam/pysymemu/	yes
Triton	x86-64	http://triton.quarkslab.com	yes
angr	libVEX based	http://angr.io/	yes

Source: `https://en.wikipedia.org/wiki/Symbolic_execution`

Outline

1. Background
2. Enabling Techniques
   - SAT Solving
   - Data Flow Tracking
3. Applications
   - Software Testing
   - Whitebox Fuzzing
   - Program Understanding/Reverse Engineering
4. Summary

Outline

SAT Problem

SAT

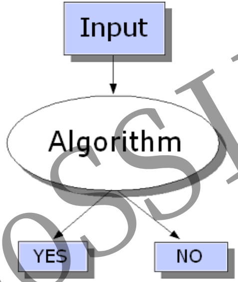
In computer science, satisfiability (often written in all capitals or abbreviated SAT) is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE.

In complexity theory, the satisfiability problem (SAT) is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses.
The question is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true?

Decision Problem

Definition

In computability theory and computational complexity theory, a decision problem is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters



Basic Concepts

Literal

A literal p is a variable x or its negation $\neg x$.

Clause

A clause C is a disjunction of literals: $x_1 \vee x_2 \vee x_3$

CNF

A CNF is a conjunction of clauses:
$(x2 \vee x41 \vee x15) \wedge (x6 \vee x2) \wedge (x31 \vee x41 \vee x6 \vee x156)$

SAT is a NP-complete problem

SAT Problem

The SAT-problem is:

1. Find a boolean assignment
2. such that each clause has a true literal

First problem shown to be NP-complete (1971)

An Example with Z3Py

```
#!/usr/bin/env python

# Copyright (c) Microsoft 2015

from z3 import *

x = Real('x')
y = Real('y')
s = Solver()
s.add(x + y > 5, x > 1, y > 1)
print(s.check())
print(s.model())

m = s.model()
for d in m.decls():
    print "%s = %s" % (d.name(), m[d])
```

An Example with Z3Py

```
#!/usr/bin/env python

# Copyright (c) Microsoft 2015

from z3 import *

x = Real('x')
y = Real('y')
s = Solver()
s.add(x + y > 5, x > 1, y > 1)
print(s.check())
print(s.model())

m = s.model()
for d in m.decls():
    print "%s = %s" % (d.name(), m[d])
```

```
~/z3/examples/python$ ./example.py
sat
[y = 4, x = 2]
y = 4
x = 2
```

A mini symbolic execution engine

```
#!/usr/bin/env python
# Copyright (c) 2015 Xi Wang
from mc import *

def test_me(x, y):
  z = 2 * x
  if z == y:
    if y == x + 10:
      assert False

x = BitVec("x", 32)
y = BitVec("y", 32)
test_me(x, y)
```

```
zlin@zlin-desktop:~/mini-mc$ ./t.py
[27486] assume (2*x == y)
[27487] assume not (2*x == y)
[27487] exit
[27486] assume (y == x + 10)
[27488] assume not (y == x + 10)
[27488] exit
[27486] Traceback (most recent call last):
  File "./t.py", line 15, in <module>
    test_me(x, y)
  File "./t.py", line 11, in test_me
    assert False
AssertionError: x = 10, y = 20
[27486] exit
```

A mini symbolic execution engine

```python
#!/usr/bin/env python
# Copyright (c) 2015 Xi Wang
from mc import *

def test_me(x, y):
  z = 2 * x
  if z == y:
    if y == x + 10:
      assert False

x = BitVec("x", 32)
y = BitVec("y", 32)
test_me(x, y)
```

```
zlin@zlin-desktop:~/mini-mc$ ./t.py
[27486] assume (2*x == y)
[27487] assume not (2*x == y)
[27487] exit
[27486] assume (y == x + 10)
[27488] assume not (y == x + 10)
[27488] exit
[27486] Traceback (most recent call last):
  File "./t.py", line 15, in <module>
    test_me(x, y)
  File "./t.py", line 11, in test_me
    assert False
AssertionError: x = 10, y = 20
[27486] exit
```

http://kqueue.org/blog/2015/05/26/mini-mc/

Mostly used SMT Solvers

Z3

A high-performance theorem prover being developed at Microsoft Research. Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers.

Yices

An efficient SMT solver that decides the satisfiability of arbitrary formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and lambda expressions

Mostly used SMT Solvers

MiniSmt

MiniSmt is a simple SMT solver for non-linear arithmetic based on MiniSat and Yices

CVC3

CVC3 is an automatic theorem prover for Satisfiability Modulo Theories (SMT) problems. It can be used to prove the validity (or, dually, the satisfiability) of first-order formulas in a large number of built-in logical theories and their combination.

Mostly used SMT Solvers

STP

STP is a constraint solver (also referred to as a decision procedure or automated prover) aimed at solving constraints generated by program analysis tools, theorem provers, automated bug finders, biology, cryptography, intelligent fuzzers and model checkers. STP has been used in many research projects at Stanford, Berkeley, MIT, CMU and other universities.

Outline

Data Flow Tracking (DFT)

DFT is characterized by 3 aspects:

1. **Data sources**: program, or memory locations, where data of interest enter the system and subsequently get tagged

Data Flow Tracking (DFT)

DFT is characterized by 3 aspects:

1. **Data sources**: program, or memory locations, where data of interest enter the system and subsequently get tagged
2. **Data tracking**: process of propagating data tags according to program semantics

Data Flow Tracking (DFT)

DFT is characterized by 3 aspects:

1. **Data sources**: program, or memory locations, where data of interest enter the system and subsequently get tagged
2. **Data tracking**: process of propagating data tags according to program semantics
3. **Data sinks**: program, or memory locations, where checks for tagged data can be made

Data flow tracking tracks the input sources and propagations, and enables the reasoning of program input.

Symbolic Execution

For each path, build a path condition

- Condition on inputs, for the execution to follow that path
- Check path condition satisfiability (SAT-problem), explore only feasible paths
- When execution path diverges, fork, adding constraints on symbolic values
- When we terminate (or crash), use a constraint solver to generate concrete input

Symbolic Execution

For each path, build a path condition

- Condition on inputs, for the execution to follow that path
- Check path condition satisfiability (SAT-problem), explore only feasible paths
- When execution path diverges, fork, adding constraints on symbolic values
- When we terminate (or crash), use a constraint solver to generate concrete input

Symbolic state

- Symbolic values/expressions for variables
- Path condition
- Program counter

Symbolic Execution

input = "\x06\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
a=6	i0	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x06\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
a=6, b=15	i1	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x06\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
a=6, b=15	i1	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x06\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
a=6, b=15	i1	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x06\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
	i1	
a=6, b=15, c=9	c=i0+3	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x06\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
	i1	
	c=i0+3	
a=6, b=15, c=9		i0+3 <= 42

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints

equation: `i0 + 3 > 42`
solution: `i0 = 40`

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
a=40	i0	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
a=40, b=15	i1	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
a=40, b=15	i1	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
a=40, b=15	i1	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
	i1	
a=40, b=15, c=43	c=i0+3	

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
	i1	
	c=i0+3	
a=40, b=15, c=43		i0+3 > 42

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x0f\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
	i1	
	c=i0+3	
		i0+3 > 42
a=40, b=15, c=43		i0 − i1 != 7

In courtesy of Gabriel Campana for this example

Symbolic Execution

```
input = "\x28\x00\x00\x00\x21\x00\x00\x00"
```

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints

equation: `i0 + 3 > 42 && i0 – i1 == 7`
solution: `i0 = 40, i1 = 33`

In courtesy of Gabriel Campana for this example

Symbolic Execution

input = "\x28\x00\x00\x00\x21\x00\x00\x00"

```
void main() {
  int a, b;
  FILE *fp = fopen("input", "r");

  fread(&a, sizeof(int), 1, fp);
  fread(&b, sizeof(int), 1, fp);

  f(a, b);
}

void f(int a, int b) {
  int c = a + 3;

  if (c > 42) {
    if (a - b == 7)
      error();
  }
}
```

concrete state	symbolic state	constraints
	i0	
	i1	
	c=i0+3	
		i0+3 > 42
		i0 − i1 == 7
a=40, b=33, c=43		

In courtesy of Gabriel Campana for this example

Outline

Outline

Directed Automated Random Testing (DART)

1. Automated extraction of program interface from source code (through code parsing by compilers)
2. Generation of test driver for random testing through the interface
3. Dynamic test generation to direct executions along alternative program paths

DART [Godefroid et al., PLDI 2005]
http://research.microsoft.com/en-us/um/people/pg/public_psfiles/pldi2005.pdf

Outline

Software security bugs can be very expensive

1. Cost of each Microsoft Security Bulletin: $Millions
2. Cost due to worms (Slammer, CodeRed, Blaster, etc.): $Billions
3. Many security exploits are initiated via files or packets
   - Ex: MS Windows includes parsers for hundreds of file formats
4. 0-day Vulnerability means money/weapon

Hunting for Security Bugs

Black hat

1. Code inspection (of binaries)
2. Blackbox fuzz testing

Blackbox fuzz testing

1. A form of blackbox random testing [Miller+90]
2. Randomly fuzz (=modify) a well-formed input
3. Grammar-based fuzzing: rules that encode “well-formed”ness + heuristics about how to fuzz (e.g., using probabilistic weights)

Black-box fuzzing has been heavily used in security testing – Simple yet effective: many bugs found this way

Blackbox Fuzzing

Examples

1. Peach, Protos, Spike, Autodafe, etc.

Why so many blackbox fuzzers?

- Because anyone can write (a simple) one in a week-end!
- Conceptually simple, yet effective
- Sophistication is in the “add-on”
  - Test harnesses (e.g., for packet fuzzing)
  - Grammars (for specific input formats)

No principled test generation

- No attempt to cover each state/rule in the grammar
- When probabilities, no global optimization (simply random walks)

Introducing Whitebox Fuzzing

Idea: mix fuzz testing with dynamic test generation

1. Symbolic execution
2. Collect constraints on inputs
3. Negate those, solve with constraint solver, generate new inputs

- Foundation: DART (Directed Automated Random Testing)
- Key extensions: (“Whitebox Fuzzing”), implemented in SAGE [NDSS’08]

Whitebox Fuzzing

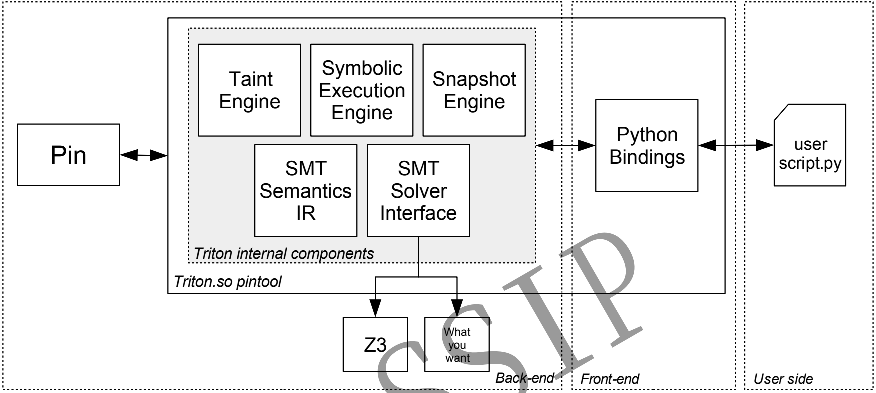
Insight

Use of algebraic expressions to represent the variable values throughout the execution of the program.

Basic Idea

- Symbolically execute the target program on a given input,
- Analyze execution path and extract path conditions
- depending on the input
- Negate each path condition
- Solve constraints and generate new test inputs
- This algorithm is repeated until all executions path are
- (ideally) covered

Internals of Whitebox Fuzzing



http://triton.quarkslab.com/

Internals of Whitebox Fuzzing

1. Dynamic Binary Instrumentation
   - At run-time disassemble instructions, and capture the semantics and constraints
2. Data Flow (Taint) Capturing and Analysis
   - Associate constraint with input
3. Constraint Solving
   - Query and solve the constraint to generate new input
4. System-events, control flow handler (Optional)
   - Run the program with new state

Outline

Introducing Angr

BAP BAT CodeReason
radare2
vivisect Hex-Ray IDA
rdis Valgrind
amoco fuzzgrind gdb
SemTrax
angr
BitBlaze
BARF JARVIS
Jakstab
klee/s2e
insight
PIN QEMU Bindead
Triton
PySysEmu TEMU PEMU
miasm CodeSurfer
paimei

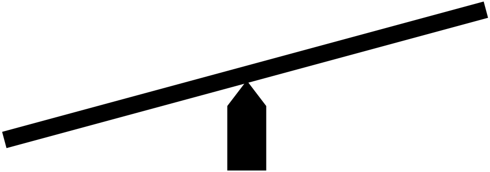
Thanks for the angr authors for providing the following slides

Introducing Angr

- iPython-accessible
- powerful analyses
- versatile
- well-encapsulated
- open and expandable
- architecture "independent"
  - x86, amd64, mips, mips64, arm, aarch64, ppc, ppc64

Thanks for the angr authors for providing the following slides

Design phyilosophy



Powerfulness	Simplicity
Full-featured	Ease of use
Accuracy	Scalability
Abstraction	Loyalty to machine code
Performance	Fast implementation

Quick Overview

```
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import angr
           [angr.init]                |     INFO: Largescale module not availabl
e. Clone from git if needed.

In [2]: p = angr.Project('/bin/echo')
           [cle.generic]              |  WARNING: Unknown reloc type: 37

In [3]: p.
p.arch                p.filename           p.loader
p.entry               p.hook               p.set_sim_procedure
p.factory             p.is_hooked          p.unhook

In [3]: p.factory.
p.factory.analyses          p.factory.path
p.factory.blank_state       p.factory.path_group
p.factory.block             p.factory.sim_block
p.factory.entry_state       p.factory.sim_run
p.factory.full_init_state   p.factory.surveyors

In [3]: p.factory.
```

Fundamentals of angr

angr

- Binary Loader
- Symbolic Execution Engine
- Static Analysis Routines

Binary Loader

CLE Loads ***E****verything*

Binary Loader

Code	Comment
prototype = SimTypeFunction([SimTypeInt()], SimTypeInt()) func = Callable(project, *address*, prototype=prototype) result = func(0x1337)	Call an arbitrary function in a loaded binary

Symbolic Execution Engine

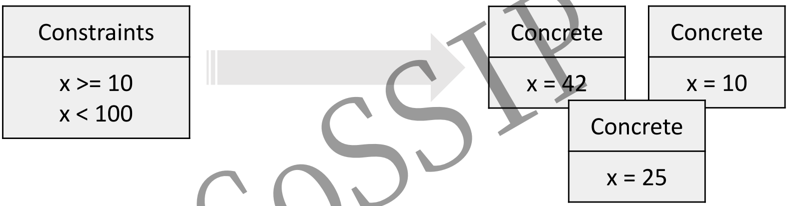
angr

- Binary Loader
- Symbolic Execution Engine
- Static Analysis Routines

Symbolic Execution Engine

constraint translator + SMT solver = symbolic execution engine

```
if (x < 100 && x >= 10) {...}
```



Symbolic Execution Engine

constraint translator ~~+ solver~~ = symbolic execution engine

Simulated environments

Tons of function summaries

Pickling

Independent Constraint Set

Path prioritization

Veritesting (smart path merging)

Floating-point support

Symbolic Execution Engine

Code	Comment

`ex = proj.surveyors.Explorer(`
`        find=some_addrs,`
`        avoid=some_addrs`
`).run()`
	Perform a symbolic exploration

`ex = proj.surveyors.Explorer(`
`        find=some_addrs,`
`        avoid=some_addrs,`
`        enable_veritesting=True`
`).run()`
	Symbolic exploration with Veritesting enabled

Symbolic Execution Engine

Pros

- Precise
- No false positives (with correct environment model)
- Produces directly-actionable inputs

Cons

- Not scalable
    - constraint solving is np-complete
    - path explosion

Static Analysis Routines

angr

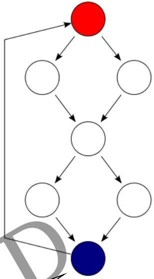
- Binary Loader
- Symbolic Execution Engine
- Static Analysis Routines

Static Analysis Routines

- angr
  - Binary Loader
  - Symbolic Execution Engine
  - Static Analysis Routines
    - Control-Flow Graph
    - Data-Flow Analysis
    - Value-Set Analysis

Static Analysis Routines

- Static Analysis Routines
  - Control-Flow Graph
  - Data-Flow Analysis
  - Value-Set Analysis



rax = **4**[0x0, 0x1024],64
rbx = **4**[0x0, 0x0],64
...
rip = 0x400560

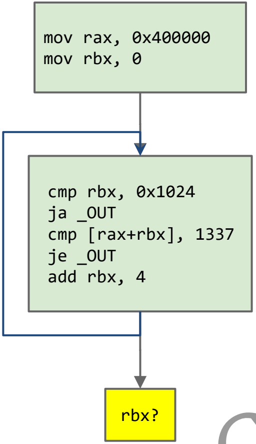
Static Analysis Routines

Code	Comment
`block = project.factory.block(addr)`	Get a block
`block.vex`	Get the VEX IRSB (with IR)
`block.capstone`	Get the capstone block (with instructions/disassembly)
`cfg = project.analyses.CFG()`	Generate a control flow graph
`vfg = project.analyses.VFG()`	Generate a value flow graph (with VSA result)

Static Analysis Routines

- angr
  - Binary Loader
  - Symbolic Execution Engine
  - Static Analysis Routines
    - Control-Flow Graph
    - Data-Flow Analysis
    - Value-Set Analysis

Value Set Analysis



What is rbx in the yellow square?

Symbolic execution: state explosion

Naive static analysis: "anything"

Range analysis: "< 0x1024"

Can we do better?

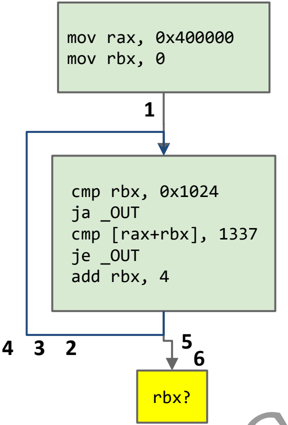
Value Set Analysis

4[0x100, 0x120],32

Stride Low High Size

0x100	0x10c	0x118
0x104	0x110	0x11c
0x108	0x114	0x120

Value Set Analysis



What is rbx in the yellow square?

1. 1[0x0, 0x0],64
2. 4[0x0, 0x4],64
3. 4[0x0, 0x8],64
4. 4[0x0, 0xc],64
5. 4[0x0, ∞],64
6. 4[0x0, 0x1024],64

Outline

Summary

Advantages

1. Symboic execution is promissing in vulnerabiliity discovery, program reverse engineering
2. It can drive the program to run desired path

Research Problems

1. Symbolic execution cannot handle complicated constraint
2. It doesn't provide clues on how to fuzz and get the vulnerability
3. Vulnerable code identification is still needed

Tools

Tool	Arch/Lang	url	Available?
KLEE	LLVM	http://klee.github.io/	yes
FuzzBALL	VineIL/native	http://bitblaze.cs.berkeley.edu/fuzzball.html	yes
JPF	java	http://babelfish.arc.nasa.gov/trac/jpf	yes
jCUTE	java	https://github.com/osl/jcute	yes
janala2	java	https://github.com/ksen007/janala2	yes
KeY	java	http://www.key-project.org/	yes
S2E	llvm/qemu/x86	http://dslab.epfl.ch/proj/s2e	yes
Pathgrind	native 32bit valgrind based	https://github.com/codelion/pathgrind	yes
Mayhem	binary	http://forallsecure.com/mayhem.html	no
Otter	C	https://bitbucket.org/khooyp/otter/overview	yes
SymDroid	Dalvik bytecode	http://www.cs.umd.edu/ jfoster/papers/symdroid.pdf	no
Rubyx	Ruby	http://www.cs.umd.edu/ avik/papers/ssarorwa.pdf	no
Pex	.NET Framework	http://research.microsoft.com/en-us/projects/pex/	no
Jalangi	JavaScript	https://github.com/SRA-SiliconValley/jalangi	yes
Kite	llvm	http://www.cs.ubc.ca/labs/isd/Projects/Kite/	yes
pysymemu	amd64/native	https://github.com/feliam/pysymemu/	yes
Triton	x86-64	http://triton.quarkslab.com	yes
angr	libVEX based	http://angr.io/	yes

Source: `https://en.wikipedia.org/wiki/Symbolic_execution`

Further Reading

- http://en.wikipedia.org/wiki/Fuzz_testing
- http://en.wikipedia.org/wiki/Symbolic_execution
- James C. King,Symbolic execution and program testing, Communications of the ACM, volume 19, number 7, 1976, 385–394
- DART: Directed Automated Random Testing, PLDI 2005
- Automated Whitebox Fuzz Testing, with Levin and Molnar, NDSS 2008
- Grammar-Based Whitebox Fuzzing, PLDI 2008
- Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware (`angr`), NDSS 2015