Machine-Code Analysis and Transformation at GrammaTech

Alexey Loginov
TAPAS 2014
Why Analyze Binaries?

- Necessity: Source code unavailable, e.g., libraries, COTS
- Convenience: Even if available, often infeasible to configure source build to match platform or release of interest
- Breadth of application:
  - Works for applications written in any compiled language(s)
  - But needs approach to multiple Instruction Set Architectures
- Fidelity: may actually be better than source-code analysis
  - **WYSINWYX**: What You See Is Not What You eXecute
  - Binaries reveal platform-specific choices of compiler
  - Binary analysis can use real libraries, not hand-written models
Why Analyze Binaries?

- Fidelity example 1
  - An optimizing compiler can introduce a dependable covert channel in the binary that is not apparent in the source code

```c
void f() {
    char password[N];
    ...
    memset(password, '\0', len);
    return;
}
```

- Fidelity example 2
  - An optimizing compiler can undo your security precautions

```c
_f:
    push ebp
    mov ebp, esp
    push ecx ; allocates 4 bytes in frame
    ... ; “Uninitialized” local gets value of ecx
```
Model-Extraction Challenges

- Subtasks for intermediate representation (IR) recovery
  - Distinguishing between code and data
  - Distinguishing between scalar and symbolic values (addresses)
  - Identifying dynamic libraries that may be loaded
  - Identifying procedure boundaries
  - Identifying variables, their sizes, and their types
  - Identifying parameters, their sizes, and their types
  - Resolving indirect jumps
  - Resolving indirect calls
  - Building inter-procedural control-flow graph (CFG)
  - Deriving useful reverse-engineering meta-data, e.g., effects on heap, file system, etc.
Disassembly Challenges Boiled Down

- Basic questions
  - What code is executed?
  - What data is accessed?

- It’s about boundaries
  - Between code and data
  - Between procedures
  - Between separate pieces of data

- … and symbols
  - Is a given value a scalar or an address?

- … in the presence of indirection
  - Extensive indirection in control flow
  - Extensive indirection in data accesses
Indirection Examples in Control Flow

- Library/function lookup by name
  - LoadLibrary, GetProcAddress

- Library-function calls via thunks, import tables
  - call [__imp__MessageBoxA@16]
  - ... with register-indirect calls as repeated-call optimizations
    - mov eax, [__imp__MessageBoxA@16]
    - ... call eax ... call eax ...

- Virtual-function calls via v-tables

- Switch statements via jump tables
Indirection Examples in Data Accesses

- Formal/local accesses via stack/frame pointers
  - `mov eax,[esp+8]; mov eax,[ebp-4]`
  - Explicit offsets may or may not indicate a data boundary
    - `mov eax,[ebp-4]`: could be the last element of an array

- Strength-reduced computations of offsets into aggregates
  - `lea eax, [eax + eax*2]`
  - `lea ebx, [ebx + eax*4 + 0x12345678]`

- Globals referenced by their address
  - `mov eax, [0x12345678]`
  - Non-zero-based accesses make it difficult to determine layout
Indirection Challenges

- Understanding indirection requires
  - Substantial numeric analysis
    - Value ranges, strides, linear relations
  - Symbol propagation
  - String analysis
Outline

- Analyzing binaries: why and why is it hard?
- Foundations of sound analyses
- Program transformation
- Heuristic analyses
- Stack-object delineation
- A comprehensive approach to security
Core Analyses

- **VSA: Value-Set Analysis**
  - Compute possible contents of variables
    - Both integer and pointer values
  - Compute information about memory accesses
    - Memory locations read, written

- **ASI: Aggregate-Structure Identification**
  - Identify variables and types

- **ARA: Affine-Relation Analysis**
  - Identify induction variables
  - Identify other key linear relations
    - E.g., between stack and frame pointers

- **QFBV: Quantifier-Free Bit-Vectors**
  - Logical representation of instruction semantics
    - Amenable to heuristic and formal reasoning
TSL/ISAL: Automatic Analysis Retargeting

- Separate processor semantics specs from analysis implementations
- Automatically generate platform-specific analysis instantiations
- Benefits:
  - Independence of semantics and analyses
    - Validation of each ISA semantics is separate from static analyses
    - Validation of each static analysis is separate from ISA definitions
  - Consistency. All analyses for given ISA driven off of same definition
  - Completeness. Full analysis generated for all instructions.
TSL: Formal ISA Semantics

- Specifies concrete operational semantics of an ISA
  › Uses small set of platform-independent operations

```plaintext
state interpInstr(instruction I, state S) {
  with(I) {
    MOV(dspOp, srcOp):
      let srcVal = interpOp(S, srcOp) in (updateState(S, dspOp, srcVal)),
    ADD(dstOp, srcOp):
      let dstVal = interpOp(S, dstOp);
      srcVal = interpOp(S, srcOp);
      sum = dstVal + srcVal;
      in (updateState(S, dstOp, sum))
  }
};

INT32 interpOp(state S, operand srcOp) {
  with(S) (State(mem, regs, flags):
    with(srcOp) {
      DirectReg(r): regs(r),
      Indirect(base, disp):
        let base_v = regs(base);
        addr = base_v + disp;
        in (mem(addr)),
      Immediate(i): i
    }
  }
};
```
Validation Infrastructure

Validate model of processor state

Validate abstract domains and solvers

var_20 ∈ [5, 30]
ptr → g
r13 ≤ r11 - #06c

Concrete Semantics

QEMU trace

Abstract Analyses

Disassembly

ISAL spec (decoder)

objdump

stmdb r13!, {r4,r5,r6,r7,r8,r9,r10,r11,lr}
sub r13, r13, #0x490
sub r13, r13, #0xc
str r0, [r13, #0x38]
ldr r0, [r13, #0x4c0]
mov r9, r2
ldr r0, [r0, #0x8]
ldr r2, [r13, #0x4c0]
cmp r0, #0xc
ldr r2, [r2, #0x28]
str r0, [r13, #0x34]
Outline

- Analyzing binaries: why and why is it hard?
- Foundations of sound analyses
- Program transformation
- Heuristic analyses
- Stack-object delineation
- A comprehensive approach to security
Melt-Stir-and-Refreeze Rewriting
Whole program rewriting

- CodeSurfer
  - Melt: Recover high-fidelity Intermediate Representation (IR)
  - Stir: Transform IR based on scripts written in C, C++, Scheme, Python, etc.
  - Refreeze: Emit assembly

- Applications
  - Performance, patching, security (obfuscation, diversification, IP protection)
Sample Transformations

- “Null” transform
  - Not identity: reassembly will result in new placement of data, code
- Converting dynamic linking to static linking
- Abstraction-layer removal (inlining+simplification)
- Simple specialization/partial evaluation
- Dead code removal
- SLX: stack-layout transformation
Case Study: GNU coreutils

- 104 Linux executables
  - Sizes range: 24K-115K
  - Package includes substantial test coverage of each executable
  - Initial melt phase: IDA Pro
    - Stir (“Null” transform) and refreeze break tests for every executable
  - Current melt phase: IDA Pro + GT improvements to IR
    - Stir (many transforms) and refreeze preserve all tested functionality
Melt-Match-and-Refreeze Rewriting
Component Extraction

- **CodeSurfer**
  - *Melt*: Recover high-fidelity Intermediate Representation (IR)
  - *Match*: Pattern match in IR for components of interest
  - *Refreeze*: Emit assembly and high-level metadata for located components

- **Applications**
  - Reuse of components in legacy software
Outline

- Analyzing binaries: why and why is it hard?
- Foundations of sound analyses
- Program transformation
- Heuristic analyses
- Stack-object delineation
- A comprehensive approach to security
CodeSonar/C,C++

- A source-code analyzer that finds serious flaws in software
  - Language Misuse
  - Library Misuse
  - Enforcement of domain-specific rules

- Sample checks
  - Buffer Overrun
  - Null-Pointer Dereference
  - Divide by Zero
  - Uninitialized Variable
  - Free Null Pointer
  - Unreachable Code
  - Dangerous Cast
  - Missing Return Statement
  - Return Pointer to Local
  - Format String Vulnerability
  - Free Non-Heap Variable
  - Use After Free/Close
  - Double Free/Close
  - Memory/Resource Leak
  - Mismatched Array New/Delete
  - Invalid Parameter
  - Unchecked Return Code
  - Race Condition
Buffer Overrun at lexpgn.c:2205
Jump to warning location ↓

No properties have been set. | edit properties
warning details...

Show Events | Options

char *return_append_str(char *dest, const char *s) {
  char *newloc;
  size_t newlen;
  /* This doesn't have buffer overflow vulnerabilities, because
  we always allocate for enough space before appending. */
  if (!dest) {
    newloc = (char *) malloc(strlen(s)+1);
    strcpy(newloc, s);
    return newloc;
  }

Buffer Overrun
This code writes past the end of the buffer pointed to by newloc.
- newloc evaluates to malloc(strlen(s)) + 1 from lexpgn.c:2204.
- strcpy() writes to the byte at an offset that is the length of the string pointed to by s, plus 1 from the beginning of the buffer pointed to by newloc.
  - The offset exceeds the capacity.
  - The length of the string pointed to by s, plus 1 is no less than 1. See related event 8.
  - The capacity of the buffer pointed to by newloc, in bytes, is the length of the string pointed to by s, which is bounded below by 0. See related events 6 and 9.
- The overrun occurs in heap memory.

The issue can occur if the highlighted code executes.
See related events 6, 8, and 9.
Show: All events | Only primary events
Buffer Overrun at lexpgn.c:2205

Jump to warning location ↓

No properties have been set. | edit properties

warning details...

Show Events | Options

return_append_str (/bal0/magagnosrc/src/trunk-swxy/opti/codesonar-tests/regression/swxy/precision/gnuchess-5.07-src.temp/gnuchess-5.07/src/lexpgn.c)

2197
2198
2199
2200
2201
2202
2203
2204
2205

char *return_append_str(char *dest, const char *s) {
    char *newloc;
    size_t newlen;
    /* This doesn't have buffer overflow vulnerabilities, because
       we always allocate for enough space before appending. */
    if (!dest) {
        newloc = (char *) malloc(strlen(s)+1);
        strcpy(newloc, s);
    newloc = (char *) malloc(strlen(s)+1);
    strcpy(newloc, s);
**CodeSonar for Binaries**

*Tool input is source and/or binary*

- Tool for binary code is an extension of source-code tool
- Input is a combination of source and binary

![Diagram](image_url)
Buffer Overrun at gnuchess.lst:39513
Jump to warning location ↓

No properties have been set. | edit properties
warning details...

Categories: LANG.MEM.BO CWE:120 CWE:788

Event 15: malloc() returns the address of a new object.
- This points to the buffer that will be overrun later.

Buffer Overrun
This code writes past the end of the buffer pointed to by strcpy:parameter_1.
- strcpy:parameter_1 evaluates to malloc() + 1 from gnuchess.lst:39509.
- strcpy() writes to the byte at an offset that is the length of the string pointed to by strcpy:parameter_2, plus 1 from the beginning of the buffer pointed to by strcpy:parameter_1.
  - The offset exceeds the capacity.
  - The length of the string pointed to by strcpy:parameter_2, plus 1 evaluates to the length of the string pointed to by s, plus 1, which is bounded below by 1. See related event 23.
  - The capacity of the buffer pointed to by strcpy:parameter_1, in bytes, is the length of the string pointed to by s, which is bounded below by 0. See related events 16 and 22.
- The overrun occurs in heap memory.

The issue can occur if the highlighted code executes.
Event 15: `malloc()` returns the address of a new object.
• This points to the buffer that will be overrun later.

Event 19: `ebx` is set to `eax + 1`, which evaluates to `malloc() + 1` from `malloc()` + 1 from `malloc`.

```c
newloc = (char *) malloc(strlen(s)) + 1;
strcpy(newloc, s);
```

- The offset exceeds the capacity.
- The length of the string pointed to by `strcpy:parameter_2`, plus 1 evaluates to the length of the string pointed to by `s`, plus 1, which is bounded below by 1. See related event 23.
- The capacity of the buffer pointed to by `strcpy:parameter_1`, in bytes, is the length of the string pointed to by `s`, which is bounded below by 0. See related events 16 and 22.
- The overrun occurs in heap memory.

The issue can occur if the highlighted code executes.
Heuristics for Scalability

- Need for scalability/linear cost requires modular analysis
- Examples of helper analyses supporting modular analysis
  - Save-restore pairs and truly-live registers
    - For type inference, parameter identification, …
  - Affine-relations analysis between esp, ebp, etc.
    - For analyzing frame delta (stack height) in the presence of
      - Indirect calls (with varying calling conventions)
      - Stack allocation (alloca)
  - Local value-set analysis
    - For resolving repeated calls via the import table (cl optimization)
Library Models for Precision

- Most analyses rely on models of library functions
  - CodeSurfer models for dependences
  - CodeSonar models for symbolic transformations
  - ARA models for effect on stack pointer
  - ...

- Challenges
  - Writing them and keeping them in sync
  - Identifying calls to library functions!
    - Dynamic linking (easy): import-table information
    - Static linking (harder): IDA Pro’s FLIRT (pattern-matching)
    - Static linking with inlining (hardest)
      - Limited pattern-matching (concentrate on string- and buffer-manipulation)
      - Instruction reordering complicates things
Outline

- Analyzing binaries: why and why is it hard?
- Foundations of sound analyses
- Program transformation
- Heuristic analyses
- Stack-object delineation
- A comprehensive approach to security
A Classic Buffer Overflow

gets(buffer);
A Classic Buffer Overflow

- Can detect in CodeSonar
  › Return-address location is known

- Can protect using:
  › Shadow stacks
  › Padding
  › Activation-record canary / guard
  › Taint propagation
What About Intra-frame Overruns?

```
bool did_auth = false;
char password[12];

gets(password);
if (strcmp(password, "hunter2") == 0)
  did_auth = true;
...
if (did_auth)
  do_something_important();
```
What About Intra-frame Overruns?

- Non-control data attacks:
  - Do not smash stack
  - Overwrite important program data
What About Intra-frame Overruns?

- Non-control data attacks:
  - Do not smash stack
  - Overwrite important program data

- Need to infer stack-object boundaries
CodeSonar Stack-Buffer-Overrun Detection

- Two modes for stack-buffer overruns:
  - Stack-smashing only (false negatives)
  - Intra-frame overruns (false positives, if using all stack-offset bounds)

We want this without this!
Accuracy is Important!

- Too conservative (missed “true” boundaries)
  - Some objects are left unprotected
  - Weakened defenses
  - Static analysis (CodeSonar) misses problems
Accuracy is Important!

- Too aggressive (found “false” boundaries)
  - Transformations (e.g., SLX) may break program
  - Static analysis (CodeSonar) will report false positives
SOD: Source Code

- Object boundaries are directly specified
  - Also applies to binaries with debug info

```c
int get_max_salary(int age)
{
  int max_salary = 0;
  struct Person p;

  while (read_person_data(&p))
  {
    if (p.age == age) {
      if (p.salary >= max_salary) {
        max_salary = p.salary;
      }
    }
  }

  return max_salary;
}
```
**SOD: Stripped Binary**

- No variable declarations
  - Local variables are numeric offsets into the procedure frame

```
080483be <get_max_salary>:
80483be:  push  %ebp
80483bf:  mov  %esp,%ebp
80483c1:  sub  $0x34,%esp
80483c4:  movl  $0x0,-0x4(%ebp)
80483cb:  jmp  80483e3
80483cd:  mov  -0xc(%ebp),%eax
80483d0:  cmp  0x8(%ebp),%eax
80483d3:  jne  80483e3
80483d5:  mov  -0x8(%ebp),%eax
80483d8:  cmp  -0x4(%ebp),%eax
80483db:  jl  80483e3
80483dd:  mov  -0x8(%ebp),%eax
80483e0:  mov  %eax,-0x4(%ebp)
80483e3:  lea  -0x24(%ebp),%eax
80483e6:  mov  %eax,(%esp)
80483e9:  call  80483b4 <read_person_data>
80483ee:  test  %eax,%eax
80483f0:  jne  80483cd
80483f2:  mov  -0x4(%ebp),%eax
80483f5:  leave
80483f6:  ret
```
Existing Techniques

- Reverse-Engineering Heuristics
  - Stack offsets appearing in binary (as in IDA Pro)
    - Generally too aggressive (finds spurious boundaries)

- Existing sound static analyses
  - DIVINE, ASI, VSA
  - Too conservative, not sufficiently scalable
  - Only work for memory-safe programs
    - Will use unsafe memory accesses to infer object bounds

- Dynamic analysis of memory-access patterns
  - Howard, Rewards, Laika
  - Require test suites:
    - That provide good coverage
    - That do not violate memory safety
Existing Techniques

- Reverse-Engineering Heuristics
  - Stack offsets appearing in binary (as in IDA Pro)
    - Generally too aggressive (finds spurious boundaries)

- Existing sound static analyses
  - DIVINE, ASI, VSA
    - Too conservative, not sufficiently scalable
    - Only work for memory-safe programs
    - Will use unsafe memory accesses to infer object bounds

- Dynamic analysis of memory-access patterns
  - Howard, Rewards, Laika
  - Require test suites:
    - That provide good coverage
    - That do not violate memory safety
Different Notions of Soundness

- Sound transformations
  - Preserve all behaviors of program
    - But we want to preserve only memory-safe behaviors
      - Heuristic in nature
  - Must detect a subset of buffer boundaries

- Sound analysis
  - Report all potential problems
  - Must detect a superset of buffer boundaries
Parameter Offset Analysis (POA)

- Observation:
  - Functions often take objects by pointer or reference
  - Field accesses become *(param + C)* where C is a syntactic constant

```c
int get_max_salary(int age) {
    int max_salary = 0;
    struct Person p;

    while (read_person_data(&p)) {
        if (p.age == age) {
            if (p.salary >= max_salary) {
                max_salary = p.salary;
            }
        }
    }
    return max_salary;
}

struct Person {
    char name[24];
    int age;
    int salary;
};

int read_person_data(struct Person * p) {
    ...
    strncpy(p->name, ..., 24);
    p->age = ...;
    p->salary = ...;
    ...
}
```
Parameter Offset Analysis (POA)

- At call-site: determine extent of locals passed as parameters

```c
int get_max_salary(int age)
{
    int max_salary = 0;
    struct Person p;

    while (read_person_data(&p))
    {
        if (p.age == age)
        {
            if (p.salary >= max_salary)
            {
                max_salary = p.salary;
            }
        }
    }
    return max_salary;
}
```
Outline

- Analyzing binaries: why and why is it hard?
- Foundations of sound analyses
- Program transformation
- Heuristic analyses
- Stack-object delineation
- A comprehensive approach to security
PEASOUP Vision

Collaboration with the University of Virginia, Georgia Tech, and Raytheon

**SOUP: Software Of Uncertain Provenance**
- Not malicious, but …
- Not safe from attack!
  - Includes:
    - Doc, PDF viewers
    - Media players
    - Utilities (zip, etc.)
    - Web browsers

**PEASOUP Analyzer**
- Recovers:
  - Structure
  - Vulnerabilities
- Generates prog. variants:
  - Repairs or confines vulnerabilities
  - Alters attack surface
- Checks correctness, strength

**Hardened Executable**
- Immune to many attacks
- On malicious input:
  - Corrected execution
  - Controlled exit
  - Uncontrolled exit
  - *Stops exploits!*

This material is based upon work supported by the United States Air Force under Contract No. FA8650-10-C-7025

© 2014 GrammaTech, Inc. All rights reserved
PEASOUP Vision

Collaboration with the University of Virginia, Georgia Tech, and Raytheon

SOUP: Software of Uncertain Provenance
- Not malicious, but …
- Not safe from attack!
  - Includes:
    - Doc, PDF viewers
    - Media players
    - Utilities (zip, etc.)
    - Web browsers

Notice: no specification is provided

PEASOUP Analyzer
- Recovers:
  - Structure
  - Vulnerabilities
- Checks correctness, strength

Hardened Executable
- Immune to many attacks on malicious input:
  - Corrected execution
  - Controlled exit
  - Uncontrolled exit
  - Stops exploits!
xkcd explanation of Heartbleed

HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).

SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).

HTTP://XKCD.COM/1354/
PEASOUP and Heartbleed

1. Health Corp employee enters sensitive diagnoses via “secure” web client
2. Hacker repeatedly sends Heartbleed attacks
3. Server leaks sensitive diagnoses in response; detects attack, sends zeros

This material is based upon work supported by the United States Air Force under Contract No. FA8650-10-C-7025
Thank you!