BUG: unable to handle kernel NULL pointer dereference at virtual address 0000009c
  printing eip:
  c01e41ee
  *pde = 00000000
Ooops: 0000 [#1]
SMP
Modules linked in:
CPU: 0
EIP: 0060:[<c01e41ee>] Not tainted VLI
EFLAGS: 0010202 (2.6.18-1-k7 #1)
EIP is at acpi_hw_low_level_read+0x7/0x6a
  eax: 00000010  ebx: 00000001  ecx: 00000094  edx: c18e1f80
  esi: c18e1f94  edi: 00000000  ebp: 00000000  esp: c18e1f68
ds: 007b  es: 007b  ss: 0068
  Process swapper (pid: 1, ti=c18e0000 task=f7b44aa0 task.ti=c18e0000)
Stack: 00000001 c18e1f94 00000000 c01e42ae 00fb3c00 00000000 00000000 f7fb3c00 c02b6834 c01c21b5 c02b66dc c01c1e26 f7fb3c00 c0344b6c 00000000 c01c12d0 00000000 c01003e1 c0102b46 00000202 c01002d0 00000000 00000000
  Call Trace:
  [<c01e42ae>] acpi_hw_register_read+0x5d/0x177
  [<c01c21b5>] quirk_via_abnormal_poweroff+0x11/0x36
  [<c01c1e26>] pci_fixup_device+0x68/0x73
  [<c01c12d0>] pci_init+0x11/0x28
  [<c01003e1>] init+0x111/0x28e
  [<c0102b46>] ret_from_fork+0x6/0x1c
  [<c01002d0>] init+0x0/0x28e
  [<c01002d0>] init+0x0/0x28e
  [<c0101005>] kernel_thread_helper+0x5/0xb
  Code: a0 82 2d c0 76 1b 50 68 85 8c 2a c0 68 f3 00 00 00 ff 35 ac ef 28
  c0 e8 c7 80 00 00 31 d2 83 c4 10 89 d0 c3 57 85 c9 56 53 74 5d <8b>
  71 08 8b 59 04 89 f7 09 df 74 51 c7 02 00 00 00 00 8a 09 84
  EIP: [<c01e41ee>] acpi_hw_low_level_read+0x7/0x6a SS:ESP 0068:c18e1f68
  <0>Kernel panic - not syncing: Attempted to kill init!
Is Your Program Memory Safe?

Can we use bounded model checking to find memory safety violations in compiled programs?

Jan Tobias Mühlberg
muehlber@cs.york.ac.uk

Supervisor: Dr. Gerald Lüttgen
Assessor: Prof. Jim Woodcock

Thesis Seminar, York, 10th July 2008
Motivation

- "BLASTing Linux Code" (Mühlberg and Lüttgen, 2006)
- "Model-checking Part of a Linux File System" (Galloway et al., 2007)

Results:
- Memory safety issues are outside of the scope of currently available software model checkers
- Biggest problem is to abstract a faithful model from a given program
Related Work

- O’Hearn and colleagues: SpaceInvader, Smallfoot
  (Yang et al., 2007)

- Microsoft Research: SLAM, VCC, Hypervisor
  (Ball et al., 2006)

- "EXE: automatically generating inputs of death"
  (Cadar et al., 2006)

- "Analyzing stripped device-driver executables"
  (Balakrishnan and Reps, 2008)
Memory Safety?

- What I am interested in:
  - Dereferencing invalid pointers
  - Uninitialised reads
  - Buffer overflows
  - Memory leaks
  - Violation of API usage rules for (de)allocation

- Not now: Shape safety
Project Outline

- Why don’t we verify on the compiled code?
Why Object Code? (Balakrishnan et al., 2005)

- Programs are not always available in source code (proprietary stuff, libraries)
- Do properties hold after compilation and optimisation?
- Many bugs exist because of platform specific details
- Programs may be modified after compilation
- Unspecified language constructs, use of inline assembly or multiple languages
Project Outline

• Why don’t we verify on the compiled code?

• Find application domain: Linux device drivers
Why Linux Device Drivers?

- Highly critical domain
- Modular software architecture
- Small programs with high complexity
- Almost no tool support for debugging and verification
- Plenty of case studies available to compare results with
Project Outline

- Why don’t we verify on the compiled code?
- Find application domain: Linux device drivers
- Chose an intermediate representation: Valgrind
Intermediate Representation

• IA32 assembly:
  – $\approx 500$ instructions, 3 byte opcodes
  – lots of instructions with multiple effects
    (i.e. POP, PUSH, CALL)
  – But still: clear semantics
Intermediate Representation

- Valgrind’s IR (Nethercote and Fitzhardinge, 2004)
  - RISC-like assembly language with arbitrary number of temporary registers
  - 12 expressions, \( \approx 130 \) operations
  - No side-effects
  - Explicit load/store operations
  - Static single assignment form
Intermediate Representation

push  %ebp
    t0 = GET: I32(20)
    t34 = GET: I32(16)
    t33 = Sub32(t34, 0x4: I32)
    PUT(16) = t33
    STle(t33) = t0

mov  %esp, %ebp
    PUT(60) = 0x8048375: I32
    t35 = GET: I32(16)
    PUT(20) = t35

sub  $0x8, %esp
    PUT(60) = 0x8048377: I32
    t4 = GET: I32(16)
    t2 = Sub32(t4, 0x8: I32)
    PUT(32) = 0x6: I32
    PUT(36) = t4
    PUT(40) = 0x8: I32
    PUT(16) = t2
## Intermediate Representation

- **Defining a semantics:**

  \[
  \begin{align*}
  &\text{Types} = \{I8, I16, I32\} \\
  &\text{Addresses} = \text{bvec}_{32} \\
  &\text{Values} = \text{bvec}_{8} \cup \text{bvec}_{16} \cup \text{bvec}_{32} \\
  &\text{Registers} = \text{Integer} \rightarrow \text{bvec}_{8} \\
  &\text{TempRegisters} = \text{Integer} \rightarrow (\text{type} \in \text{Types}, \text{val} \in \text{Values} \cup \{\bot\}) \\
  &\text{Heap} = \text{Addresses} \rightarrow \text{bvec}_{8}
  \end{align*}
  \]

  \[
  \begin{align*}
  &\text{HeapLocations} = \text{Addresses} \rightarrow (\text{alloc} : \text{Bool}, \text{init} : \text{Bool} \\
  &\quad \text{start} \in \text{Addresses}, \text{size} \in \text{bvec}_{32})
  \end{align*}
  \]

- **command-state pair:** \(\langle c, (t, r, h, l)\rangle\)
Intermediate Representation

- Defining a semantics:

\[ t(t_{\text{reg}}).val \neq \bot \]

\[ \langle \text{PUT}(\text{reg}) = t_{\text{reg}}, (t, r, h, l) \rangle \]

\[ \rightsquigarrow \begin{cases} 
(t, [r|\text{reg} : t(t_{\text{reg}}).\text{val}], h, l) & \text{if } t(t_{\text{reg}}).\text{type} = I8 \\
(t, [r|\langle \text{reg}..\text{reg} + 1 \rangle : t(t_{\text{reg}}).\text{val}], h, l) & \text{if } t(t_{\text{reg}}).\text{type} = I16 \\
(t, [r|\langle \text{reg}..\text{reg} + 3 \rangle : t(t_{\text{reg}}).\text{val}], h, l) & \text{if } t(t_{\text{reg}}).\text{type} = I32 
\end{cases} \]

\[ t(t_{\text{reg}}).\text{type} = \text{type} \land t(t_{\text{reg}}).\text{val} = \bot \]

\[ \langle \text{treg} = \text{GET} : \text{type}(%\text{reg}), (t, r, h, l) \rangle \]

\[ \rightsquigarrow \begin{cases} 
([t\mid t_{\text{reg}}.\text{val} : r(\text{reg})], r, h, l) & \text{if } \text{type} = I8 \\
([t\mid t_{\text{reg}}.\text{val} : r(\langle \text{reg}..\text{reg} + 1 \rangle)], r, h, l) & \text{if } \text{type} = I16 \\
([t\mid t_{\text{reg}}.\text{val} : r(\langle \text{reg}..\text{reg} + 3 \rangle)], r, h, l) & \text{if } \text{type} = I32 
\end{cases} \]
Intermediate Representation

- And translate the program into a set of bit-vector constraints for Yices (Dutertre and de Moura, 2006):

...  
(define t34.0x8048374.1::(bitvector 32) (bv-concat
  (bv-concat r19.0x00000001.0.0 r18.0x00000001.0.0)
  (bv-concat r17.0x00000001.0.0 r16.0x00000001.0.0)))

(define t33.0x08048374.1::(bitvector 32)
  (bv-sub t34.0x08048374.1 (mk-bv 32 4)))

...
Project Outline

• Why don’t we verify on the compiled code?
• Find application domain: Linux device drivers
• Chose an intermediate representation: Valgrind
• For each program location, check safety properties:
Symbolic Execution

- Construct constraint system for each possible path of the program (bounded loop unrolling)
- Registers and heap/stack are initially allowed to hold any possible value
- Add `(assert ...)` for all pointer operations
- `(check)`
Symbolic Execution

... 
(define t36.0x08048358.1::(bitvector 32) (bv-concat 
  (bv-concat (heap.00000010 (bv-add t34.0x08048358.1 (mk-bv 32 3))) 
    (heap.00000010 (bv-add t34.0x08048358.1 (mk-bv 32 2)))) 
  (bv-concat (heap.00000010 (bv-add t34.0x08048358.1 (mk-bv 32 1))) 
    (heap.00000010 t34.0x08048358.1))))

(define r0.0x08048358.5.1::(bitvector 8) 
  (bv-extract 7 0 t36.0x08048358.1))

(define r1.0x08048358.5.1::(bitvector 8) 
  (bv-extract 15 8 t36.0x08048358.1))

(define r2.0x08048358.5.1::(bitvector 8) 
  (bv-extract 23 16 t36.0x08048358.1))

(define r3.0x08048358.5.1::(bitvector 8) 
  (bv-extract 31 24 t36.0x08048358.1))

(define t19.0x0804835b.1::(bitvector 32) (bv-concat 
  (bv-concat r3.0x08048358.5.1 r2.0x08048358.5.1) 
  (bv-concat r1.0x08048358.5.1 r0.0x08048358.5.1))))

;; checking t19.0x0804835b.1 (r) 
(assert (= t19.0x0804835b.1 0b00000000000000000000000000000000))
(check)
Project Outline

• Why don’t we verify on the compiled code?
• Find application domain: Linux device drivers
• Chose an intermediate representation: Valgrind
• For each program location, check safety properties:
  bounded model checking, symbolic execution
  – Of course it doesn’t work...
Project Outline

- Why don’t we verify on the compiled code?
- Find application domain: Linux device drivers
- Chose an intermediate representation: Valgrind
- For each program location, check safety properties: bounded model checking, symbolic execution, slicing
Slicing Object Code

- Program Slicing: (Weiser, 1981), (Ottenstein and Ottenstein, 1984), (Horwitz et al., 1990)

- Decomposing programs based on control and data flow

- Basically, constructing a system dependence graph and searching for nodes the slicing criterion depends on
Slicing Object Code

push %ebp

t0 = GET:I32(20)
t34 = GET:I32(16) <-
t33 = Sub32(t34,0x4:I32) <-
PUT(16) = t33 <-
STle(t33) = t0

mov %esp, %ebp

PUT(60) = 0x8048375:I32
t35 = GET:I32(16)
PUT(20) = t35

sub $0x8,%esp

PUT(60) = 0x8048377:I32
t4 = GET:I32(16) <-
t2 = Sub32(t4,0x8:I32)
PUT(32) = 0x6:I32
PUT(36) = t4 <- criterion
PUT(40) = 0x8:I32
PUT(16) = t2
Slicing Object Code

- Now, how do we deal with LD/ST instructions?

```c
...  
t64 = LDle:I32(t62)  
...  
STle(t64) = t63  
STle(t34) = t1  
...  
t17 = LDle:I32(t18)  
...  
STle(t17) = t12  
-------------------------------  
(assert (= t17 0b00000000000000000000000000000000))  
(check)
```
Slicing Object Code

- If all pointers evaluate to exactly one value, it’s easy.
- However, often they don’t and we might end up with "symbolic" pointers that may hold any value between $lo \leq pointer \leq up$.
- Solution: Heap dependency tree.
Slicing Object Code

- Solution: Heap dependency tree

```
    [1, 500]  
   /   \     
[1,3]   [4,500]  
     \         \  
   t12, t13  [4,90]  [91,500]  
      \     /   \     \  
    [4,4]  [5,90]  t64, t19, ...  
       \   /     \    
        t34  t64   
```
Slicing Object Code

- Bounds have to be computed for all pointers – expensive

- We have to store the dependency tree – expensive as well (but probably okay for device drivers)

- We get very good slices: complete and small!
Slicing Object Code

• Is it any good? **Initial results:**
  – 30 crypto drivers (10 interface functions each, \(50 \leq n \leq 3000\) instructions) analysed within less than an hour each, exhaustively
  – Usually \(\leq 50\) constraints per slice, solved in less than a second; but we got up to \(10^3\) constr.
  – Works fine for finding NULL-dereferences and access to memory that is not allocated, but lots of meaningless errors yet
Slicing Object Code

- Is it any good? **Less initial results:**
  - It doesn’t scale very well.
  - Experiments were executed on 20 network card drivers and 20 file system drivers (up to 50 interface functions, $3000 \leq n \leq 30000$ instructions, lots of dependencies to the kernel)
  - Looks promising but SMT solver runs out of memory quickly
Slicing Object Code

• Optimisations:
  – **PUT/GET removal**: 60% speedup, 50% saving in memory consumption (for big systems)
  – **Constant replacement**: Not implemented yet
  – **Better initial state**: Not implemented yet
Slicing Object Code

- Using different coverage criteria:
  - Currently we do bounded loop unrolling, executing each loop up to 2000 times
  - Requiring a coverage criterion like Condition Coverage to be satisfied results in fewer and shorter paths that can be analysed without exhausting resources
Slicing Object Code

• Some pointers to literature:
  – "Recovery of Jump Table Case Statements from Binary Code" (Cifuentes and Emmerik, 1999)
  – "Interprocedural Static Slicing of Binary Executables" (Kiss et al., 2003)
  – "Analyzing Memory Accesses in x86 Executables" (Balakrishnan and Reps, 2004) and "Recovery of Variables and Heap Structure in x86 Executables" (Balakrishnan and Reps, 2005)
Slicing Object Code

- Some pointers to literature:
  - "New Developments in WCET Analysis" (Ferdinand et al., 2007)
Is Your Program Memory Safe?:

**Project Outline**

- Why don’t we verify on the compiled code?
- Find application domain: Linux device drivers
- Chose an intermediate representation: Valgrind
- For each program location, check safety properties: bounded model checking, symbolic execution, slicing
Project Outline

- Why don’t we verify on the compiled code?
- Find application domain: Linux device drivers
- Chose an intermediate representation: Valgrind
- For each program location, check safety properties: bounded model checking, symbolic execution, slicing
- If a property is violated, generate a test case that will make the program crash – quickly
Summary

- Presented an approach to model checking compiled programs in order to find memory safety bugs
- Does not require any abstraction, only path-sensitive program slicing and symbolic execution
- Scalability issues as an artifact of object code; good chance that it scales for device drivers
- Bugs found are reproducible, but not very meaningful due to initial state being "too random"
Work still to do

- Optimisations to get it work
- Experimental evaluation: use drivers with known errors, follow evolution of a driver over a series of releases
- Try more properties (i.e. bounds checking)
- Deal with concurrency: (Flanagan and Godefroid, 2005), (Lal and Reps, 2008)
- Soundness and Completeness?
- Write a thesis
Thank you! Questions?


