```
BUG: unable to handle kernel NULL pointer dereference at virtual address 0000009c
 printing eip:
c01e41ee
*pde = 00000000
Oops: 0000 [#1]
SMP
Modules linked in:
CPU:    0
EIP:    0060:[<c01e41ee>]    Not tainted VLI
EFLAGS: 00010202   (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 c02b670c
       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!
```

Slicing Object Code

… and finding memory safety violations.

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

University of York, UK

Theory Seminars, Birmingham, 28th February 2008

Motivation

- *"BLASTing Linux Code"* (Mühlberg and Lüttgen, 2006)
- *"Model-checking Part of a Linux File System"*

  (Galloway et al., 2007)
- Results:
  - The biggest problem is to abstract a faithful model from a given program to be analysed.

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)

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
- But: Exhaustive and push-button

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:

$$
\begin{aligned}
Types &= (I8|I16|I32) \\
t &= I16 \rightarrow (type : Types, val : Values) \\
r &= I16 \rightarrow I8 \\
h &= Addresses \rightarrow Values \\
l &= Addresses \rightarrow (alloc : Bool, init : Bool, \\
&\quad start : I32, size : I32)
\end{aligned}
$$

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

  (with c being a command, t the set of temporary registers, r the set of CPU registers, h the current heap and l the "Locations function")

Intermediate Representation

- Defining a semantics:

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

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

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 allowed to hold any possible value initially
- 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

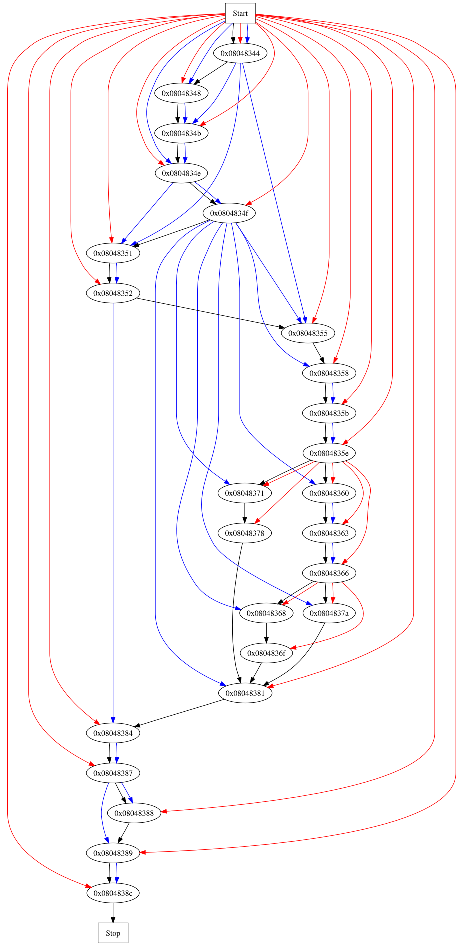
- Why don’t we verify on the compiled code?
- Find application domain: Linux device drivers
- Chose an intermediate representation: Valgrind
- For each program location, check safety properties:
  bounded model checking, symbolic execution
  – Of course it doesn’t work. . .

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?

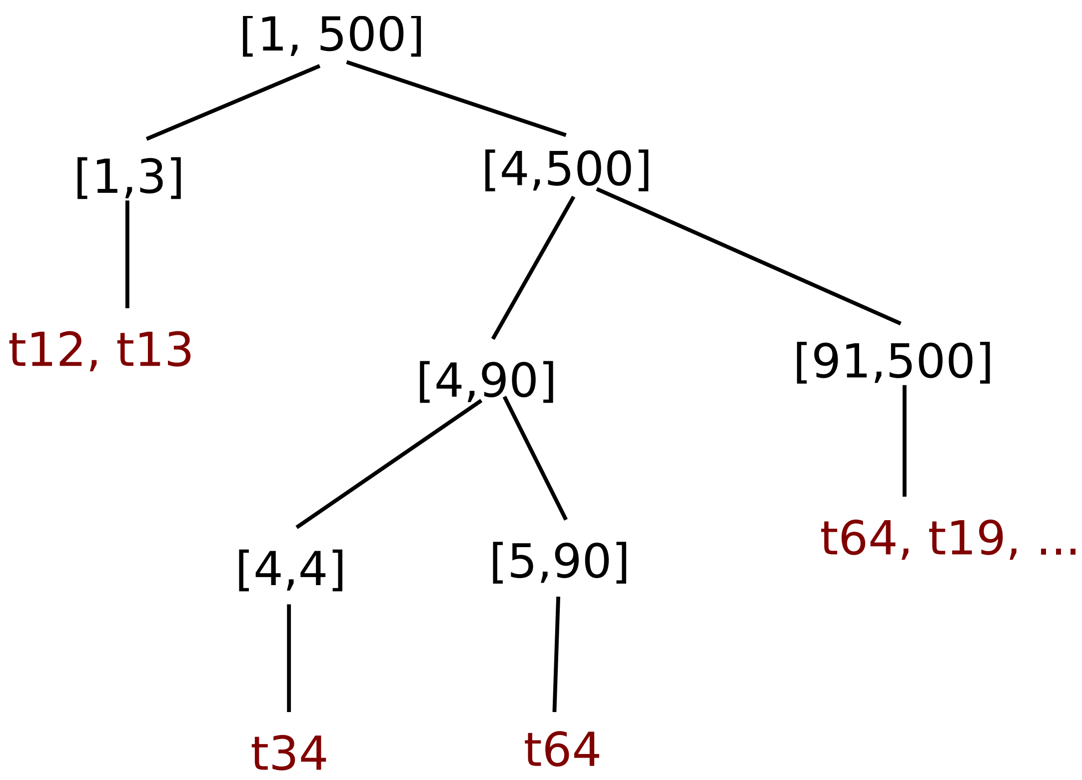
```
...
t64 = LDle:I32(t62)
...
STle(t64) = t63
STle(t34) = t1
...
t17 = LDle:I32(t18)
...
STle(t17) = t12
------------------------
 (assert (= t17 0b00000000000000000000000000000000))
 (check)
```

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



Slicing Object Code

- All satisfying values 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 precise slices!

Slicing Object Code

- Is it any good?
  - Slices are usually ≤ 200 constraints long and are solved less than a second
  - We can analyse a whole driver within an hour and using about 1 GByte of RAM (crypto drivers – quite simple, depth 2000)
  - Works fine for finding possible NULL-dereferences and access to memory that is not allocated
  - Many more experiments to do...

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:
  - "Generisches Slicing auf Maschinencode" (Schlickling, 2005)

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

Test Case Execution

- Device drivers contain lots of hardware dependent operations – we can't just run them in user space
- . . . but we can construct a new binary program by compiling all statements from (a union of) slices back to object code

→ Executes quickly without waiting for hardware interaction or similar things. But that's another talk. . .

Summary

- 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

Future Work

- Try more properties (i.e. bounds checking, etc.)
- Experimental evaluation
- Compute the test cases
- Soundness and Completeness?
- Just ask me how many bugs I've found so far...

Thank you! Questions?

References

Balakrishnan, G. and Reps, T.: 2004, Analyzing memory accesses in x86 executables, in *Proc. Int. Conf. on Compiler Construction*, Vol. 2985 of *LNCS*, pp 5 – 23

Balakrishnan, G. and Reps, T.: 2005, *Recovery of Variables and Heap Structure in x86 Executables*, Technical report, University of Wisconsin, Madison

Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T.: 2005, WYSINWYX: What You See Is Not What You eXecute, in *VSTTE*

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A.: 2006, Thorough static analysis of device drivers, in *EuroSys*

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R.: 2006, Exe: automatically generating inputs of death, in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pp 322–335, ACM, New York, NY, USA

Cifuentes, C. and Emmerik, M. V.: 1999, Recovery of jump table case statements from binary code, in *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, p. 192, IEEE Computer Society, Washington, DC, USA

Dutertre, B. and de Moura, L.: 2006, A fast linear-arithmetic solver for DPLL(T), in *CAV 2006*, No. 4144 in LNCS, pp 81 – 94

Galloway, A., Mühlberg, J. T., Siminiceanu, R., and Lüttgen, G.: 2007, *Model-checking Part of a Linux File System*, Technical Report YCS-2007-423, Department of Computer Science, University of York, UK

Horwitz, S., Reps, T., and Binkley, D.: 1990, *ACM Trans. Program. Lang. Syst.* **12(1)**, 26

Kiss, A., Jasz, J., Lehotai, G., and Gyimothy, T.: 2003, *scam* **00**, 118
Mühlberg, J. T. and Lüttgen, G.: 2006, Blasting linux code, in *FMICS 2006*, No. 4346 in LNCS, pp 211 – 226
Nethercote, N. and Fitzhardinge, J.: 2004, Bounds-checking entire programs without recompiling, in *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*
Ottenstein, K. J. and Ottenstein, L. M.: 1984, The program dependence graph in a software development environment, in *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pp 177–184, ACM, New York, NY, USA
Schlickling, M.: 2005, Generisches slicing auf maschinencode, *Master's thesis*
Weiser, M.: 1981, Program slicing, in *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pp 439 – 449, IEEE Press, Piscataway, NJ, USA
Yang, H., Lee, O., Calcagno, C., Distefano, D., and O'Hearn, P.: 2007, *On Scalable Shape Analysis*, Technical Report RR-07-10, Queen Mary, University of London