The VFiasco approach for a verified operating system

Michael Hohmuth   Hendrik Tews

Department of Computer Science, TU Dresden
`vfiasco@os.inf.tu-dresden.de`

presented by Hendrik Tews

VERIFIED

FIASCO

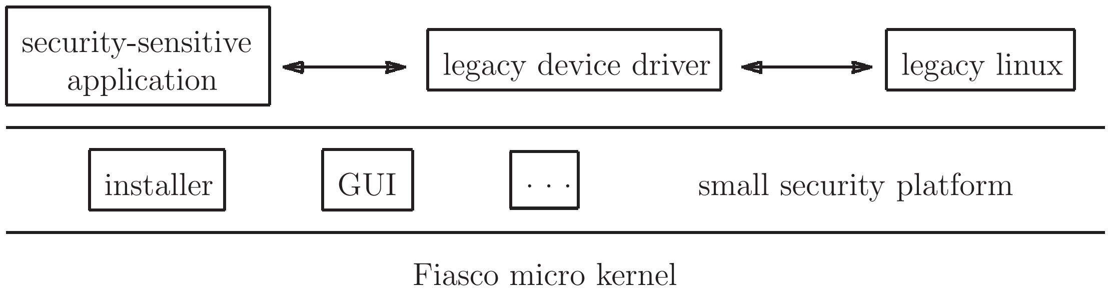
Fiasco

- 2nd generation micro kernel, compatible with L4
- developed in the DROPS project
  (Dresden Real-Time Operating System)
- 15.000 lines of C++ (without kernel debugger)
- real time: view movies with load 10
- security: sign with newest viruses present

VFiasco

- Verification of some properties of Fiasco
  (on source code level)
  - internal page-fault handler terminates
  - internal memory management is correct
- Semantics for system-level programming
  - control flow `break`, `goto`, `longjmp`
  - type casts (`unsigned` $\Longleftrightarrow$ `void *`)
  - hardware manipulations

Structure overview



Properties and Conclusions

- address spaces separate *and* isolate components
- open system: new (bogus) software can be installed (without affecting security)
- security-sensitive applications can rely on (unsecure) legacy components
- kernel *needs* to manipulate address spaces
- (closed systems are different)

Advantages of strong type systems (for user level programs)

- no runtime type errors (although sane C/C++ programmers come close with `-Wall`)
- enforce the use of certain interfaces

An wrong address-space manipulation can be devasting effects

- the contents of the memory changes
- variables suddenly contain random bitpatterns
- execution might continues at a random point

For kernel programming:

- No type system can prevent errors in the address-space manipulation.
- To prove the type correctness of the kernel one always has to treat prove obligations about type correctness.
- **Strong type systems are less useful than expected** when *guarantees* must be provided.

(Different story for user-level programs and closed systems.)

Memory allocation

- memory allocation turns muddy bytes into shiny values of abstract types
- relies on address calculations and type casts
- no type system can prevent that memory is allocated twice
  (thus breaking the guarantees of the type system)
- the kernel must do its memory allocation itself

Low level memory allocation needs to break the type system as well!

If type systems are not that useful . . .

. . . might as well verify the original C++ code . . .

. . . please the kernel hackers . . .

. . . and verify a *real* system

Challenges

- type casts
- abrupt termination (`break`, `continue`)
- jumps (`goto`, `setjmplongjmp`)
- templates (are a different story)

Initialization code (Address = unsigned int)

```
Address Kmem::alloc_from_page(Address *from, Address size) {
  Address ret = *from;
  *from += (size + 0xf) & ~0xf;
  return ret;
}

Address cpu_page_vm = ...;
io_bitmap_delimiter = reinterpret_cast<Unsigned8 *>(alloc_from_page(& cpu_page_vm, 1));
```

The interrupt handler Thread::handle_slow_trap:

```
  if (EXPECT_FALSE (gdb_trap_recover)) goto generic_debug;
  ...
  if (!check_trap13_kernel (ts, from_user)) return 0;
  ...
  if (EXPECT_FALSE (! ((ts->cs & 3) || (ts->eflags & EFLAGS_VM)))) {
      if (check_trap13_smas (ts)) goto success;
      goto generic_debug; }
  ...
success:
  _recover_jmpbuf = 0;
  return 0;

generic_debug: ...
```

Abrupt termination

- `break`, `continue`, `return`
- `setjmp/longjmp`

Goto: structured jumps

- jump labels have a limited scope (function body)
- no jumps into other functions (and more restrictions)
- jumps into and out of blocks/loops/if's possible

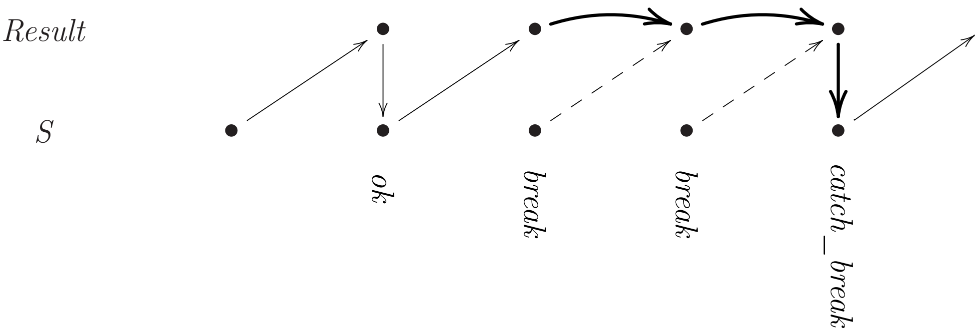
Switch

- computed jump (in contrast to a case switch)
- fall through (without `break`)
- labels may appear inside other blocks (Duff's device)

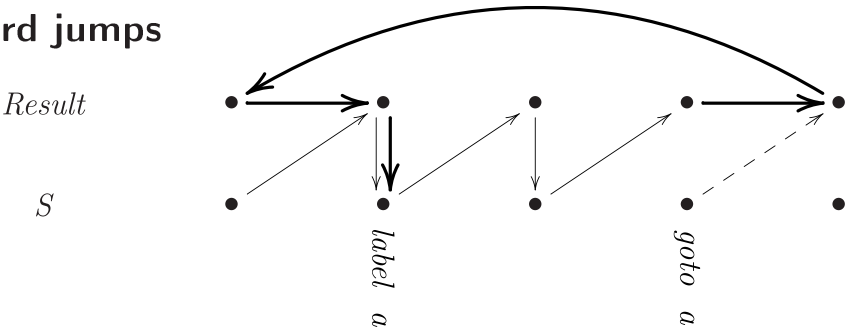
Recall nonsense:

```
if(a == 0) goto x;
switch(b){
  case 0: if(c == 0)
       x:    d = 1;
          else
  case 1:    d = 2;  }
```

abrupt termination and forward jumps



backward jumps



Solution is based on the LOOP project (see [1])

- Semantic domain $\quad Result \longrightarrow Result$
- $Result$ is a disjoit union
- defined as follows (with current memory S, set of labels L):

$$
Result \quad = \quad \begin{cases} S \ \uplus & ok \ \text{(normal termination)} \\ S \ \uplus & break \\ \ldots & \\ S \times \mathbb{L} \ \uplus & goto \\ \{*\} & hang \ \text{(non-termination)} \\ \{*\} & fail \ \text{(crash)} \end{cases}
$$

- Semantic Equations

$$
[\![v = a]\!] \quad = \quad \left| \begin{array}{lll} ok(s) & \mapsto & ok(s[v := a]) \\ x & \mapsto & x \end{array} \right.
$$

$$
[\![st_1; st_2]\!] \quad = \quad [\![st_2]\!] \circ [\![st_1]\!]
$$

[1] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, Berlin, 2000.

$$[\![\texttt{break}]\!] = \left|\begin{array}{lcl} ok(s) & \mapsto & break(s) \\ x & \mapsto & x \end{array}\right.$$

$$[\![\texttt{goto}\ l]\!] = \left|\begin{array}{lcl} ok(s) & \mapsto & goto(s,l) \\ x & \mapsto & x \end{array}\right.$$

$$[\![l\ :]\!] = \left|\begin{array}{lcl} goto(s,l) & \mapsto & ok(s) \\ x & \mapsto & x \end{array}\right.$$

$$[\![\texttt{if}(bx)\ st_1\ \texttt{else}\ st_2]\!] = \left|\begin{array}{lcl} ok(s) & \mapsto & \begin{cases} ok(s') & \textsf{if } [\![bx]\!](s) = true \quad \wedge \\ & \qquad [\![st_1]\!](ok\ s) = ok(s') \\ [\![st_2]\!] \circ [\![st_1]\!](ok\ s) & \textsf{if } [\![bx]\!](s) = true \quad \wedge \\ & \qquad [\![st_1]\!](ok\ s) \neq ok(-) \\ [\![st_2]\!](ok\ s) & \textsf{if } [\![bx]\!](s) = false \end{cases} \\ x & \mapsto & [\![st_2]\!] \circ [\![st_1]\!](x) \end{array}\right.$$

$$iter\ bo\ i \quad = \quad \underbrace{bo \circ bo \circ \cdots \circ bo}_{i+1 \text{ times}}$$

$$term? \quad : \quad Result \longrightarrow bool$$

$$term? \quad = \quad \left|\begin{array}{lcl} goto(s) & \mapsto & false \\ x & \mapsto & true \end{array}\right.$$

$$hangs? \quad : \quad (Result \to Result) \times Result \longrightarrow bool$$

$$hangs?(bo, x) \quad = \quad \begin{cases} true & \text{if } \forall i \in \mathbb{N} \,.\, term?(iter\ bo\ i\ x) = false \\ false & \text{otherwise} \end{cases}$$

$$[\![block]\!] \quad : \quad S \longrightarrow Result$$

$$[\![block]\!](s) \quad = \quad \begin{cases} hang & \text{if } hangs?([\![block]\!], ok(s)) = true \\ iter\ [\![block]\!]\ n\ (ok\ s) & \text{otherwise} \\ \quad \text{where } n = \min\{i \in \mathbb{N} \mid term?(iter\ [\![block]\!]\ i\ (ok\ s))\} & \end{cases}$$

- remarkably simple denotational semantics
  - no CPO's
  - no fixedpoints
  - no continuations necessary
  - relies only on disjoint union, case distinction and function composition
- can treat all of C++ control flow manipulations
  - `switch`, `case` (with fall through)
  - jumps into the middle of loops
- simple encoding in higher-order logic
- runs well in PVS (and would do well in other theorem provers)

- Fiasco verification uses an IA32 hardware model, which contains a byte-wise organised memory (see [1])
- for this talk: $S \stackrel{\text{def}}{=} \mathbb{N} \longrightarrow \mathit{Value}$
- each variable is assigned a unique *slot index* from $\mathbb{N}$ (could exploit standard solutions for local variables)
- Value is a disjoint union:

$$\mathit{Value} \quad = \quad \begin{cases} \mathbb{Z} \uplus & \mathit{int} \ \text{(integers)} \\ \{\mathit{true}, \mathit{false}\} \uplus & \mathit{bool} \\ \ldots & \end{cases}$$

- memory S is dynamically typed: integers can change into booleans at runtime
- generalise Result for expressions:

$$\mathit{Result}[T] \quad = \quad \begin{cases} S \times T \uplus & \mathit{ok} \ \text{(normal termination)} \\ \ldots & \text{remainder does not change} \end{cases}$$

[1] S. Hoffmann. Formalising PC Hardware: A Model of the x86 Architecture. Diploma thesis, TU Dresden, September 2003.

- Variable access functions

$$write\_int\ i\ v \quad = \quad \left|\ \begin{array}{lll} ok(s) & \mapsto & ok(s[i := int(v)]) \\ x & \mapsto & x \end{array}\right.$$

$$read\_int\ i \quad = \quad \left|\ \begin{array}{lll} ok(s,\_) & \mapsto & \begin{cases} ok(s,n) & \text{if } s\,i = int(n) \\ fail & \text{otherwise} \end{cases} \\ x & \mapsto & x \end{array}\right.$$

- each variable access generates a type correctness proof obligation
- **to prove type correctness it is enough to prove absence of crashes**
- even for simple programs the type correctness proof obligations are overwhelming

Use standard automation facilities

type correctness invariant

specifies the expected type of all interesting variable slots
additionally array sizes and that arrays do not overlap

simplification rules

automatically simplify type correctness proof obligations for states in the type correctness invariant

invariant proofs

simplification rules to automatically prove the invariance condition

Loop unrolling in C without extra treatment of leftovers [1]

```
int rounds = count / 8;
switch(count % 8){
  case 0: while(rounds-- > 0){ *to++ = *from++;
  case 7:                      *to++ = *from++;
  case 6:                      *to++ = *from++;
  case 5:                      *to++ = *from++;
  case 4:                      *to++ = *from++;
  case 3:                      *to++ = *from++;
  case 2:                      *to++ = *from++;
  case 1:                      *to++ = *from++;
                        }          };
```

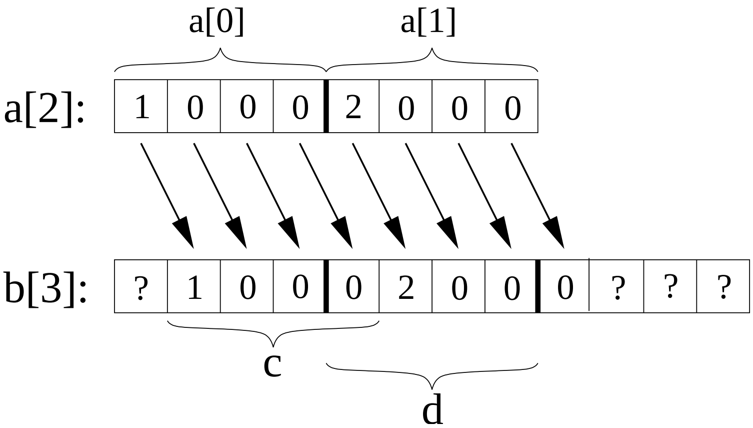
[1] T. Duff. Tom Duff on Duff's Device, 2004. Available at www.lysator.liu.se/c/duffs-device.html.

For C one also needs to handle the following

```
unsigned a[2] = { 1, 2 };
unsigned b[3];

memcpy((char *)(b) + 1, a, sizeof(a));
unsigned c = * (unsigned *) ((char *)(b) + 1);        //   √

unsigned d = b[1];                                    //   ↯
```



- In kernel programming type systems **cannot** give guarantees.
- Verification is needed to provide guarantees.
- Semantics of C/C++ is surprisingly simple.
- Verification of type correctness can be automated and goes beyond what type systems provide (array bounds).

- See also [1, 2]
- ASM Semantics [3, 4]
- non-deterministic semantics of C in HOL [5, 6]

[1] M. Hohmuth and H. Tews. The semantics of C++ data types: Towards verifying low-level system components. In D. Basin and B. Wolff, editors, *TPHOLs 2003, Emerging Trends Proceedings*, pages 127–144. 2003. TR No. 187 Inst. für Informatik Universität Freiburg.

[2] H. Tews. Verifying Duff's device: A simple compositional denotational semantics for goto and computed jumps. Submitted, 2005. Available via www.vfiasco.org.

[3] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[4] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[5] M. Norrish. Deterministic expressions in c. In D. S. Swierstra, editor, *Programming Languages and Systems; ESOP '99 Proceedings*, volume 1675 of *LNCS*, pages 147–161. Springer, 1999.

[6] M. Norrish. *Formalising C in HOL*. PhD thesis, University of Cambridge, 1998.