The VFiasco approach for a verified operating system

Hendrik Tews

Department of Computer Science, TU Dresden

http://wwwtcs.inf.tu-dresden.de/~tews/

I. **Fiasco overview**

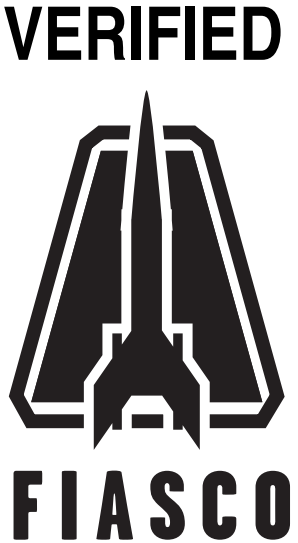
II. Insufficiency of strongly-typed languages

III. Semantics of C++ data types

IV. Semantics for abrupt termination and jumps

V. Duff's device

VI. Conclusion



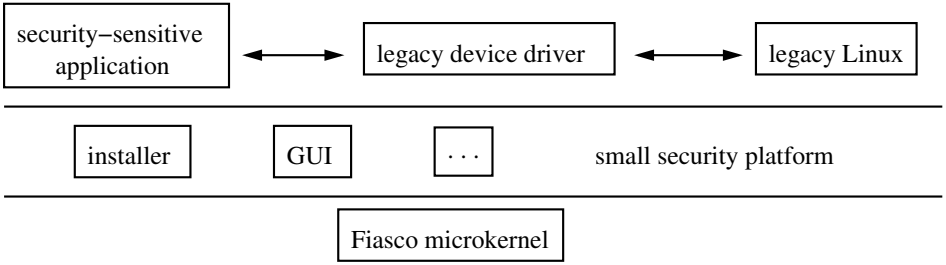
Fiasco

- 2nd generation microkernel, 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)

I. Fiasco overview

II. Insufficiency of strongly-typed languages

III. Semantics of C++ data types

IV. Semantics for abrupt termination and jumps

V. Duff's device

VI. Conclusion

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

A wrong address-space manipulation have be devastating effects

- memory contents might change
- variables suddenly contain random bitpatterns
- execution might continue at a random point

For kernel programming:

- No type system can prevent errors in address-space manipulations.
- 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 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`, `setjmp`/`longjmp`)
- 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: ...
```

I. Fiasco overview
II. Insufficiency of strongly-typed languages
III. Semantics of C++ data types
IV. Semantics for abrupt termination and jumps
V. Duff's device
VI. Conclusion

type casts

- interpreting some bits in a different type (`unsigned int` $\Longleftrightarrow$ `void *`)

Access non-existent memory

T a = * **reinterpret_cast**<T *>(50)

data types are not fully specified in the standard

- width of intergertypes (≥ 7), one's or two's complement ...

structures / unions

Conclusion

- untyped memory (in contrast to $heap : address \longrightarrow value$)
- general, scalable semantics
  e.g. for 8 bit `char` and 32 bit `unsigned`

C++ Standard

- describes objects (i.e. memory representations of values)
- objects have an address, a type and occupy memory
- standard distinguishes between values, the value representation and the object representation
- reading in memory works only at addresses with valid object representations

Conclusion

- standard permits runtime type checking (a la lisp)
- to prove type safety, it is sufficient to prove termination without errors for every possible C++ implementation

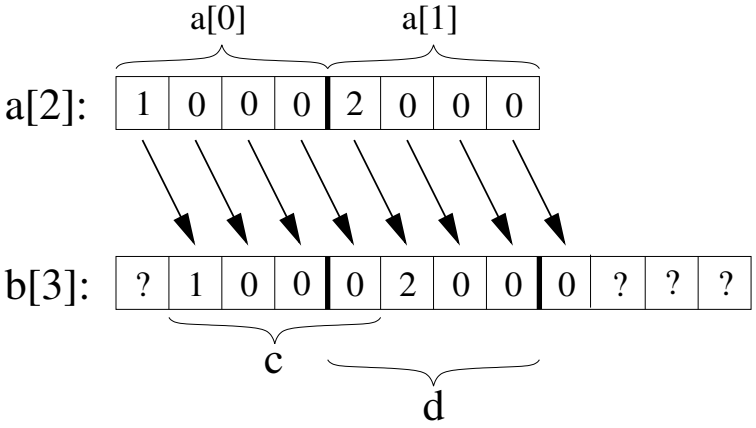
Approach for a semantics of data types

- define sets of values only for simple types (not for composite ones)
- semantics follows the standard
- semantics is quite loose (leaving many details open) and has many models (e.g. gcc)
- **normal termination $\Longrightarrow$ type safety**

```
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];                                    //   ↯
```



General structure for base types

```
Data_type_structure : Type = [#
  size : nat,
  to_byte : [Data, Address -> list[Byte]],
  from_byte : [list[Byte], Address -> lift[Data]]
#]

data_type? : PRED[Data_type_structure] = Lambda(ct : Data_type_structure) :
  (Forall(d : Data, a : Address) : length(to_byte(ct)(d,a)) = size(ct))
     And
  (Forall(d : Data, a : Address) : up?(from_byte(ct)(to_byte(ct)(d,a), a)))
     And
  (Forall(d : Data, a : Address) : down(from_byte(ct)(to_byte(ct)(d,a), a)) = d )
     And
  (Forall(l : list[Byte], a : Address) : up?(from_byte(ct)(l, a)) Implies length(l) = size(ct))
     And
  (Forall(d : Data, a1,a2 : Address) : up?(from_byte(ct)(to_byte(ct)(d,a1), a2)))
```

Semantics

```
Semantics_bool: Type = bool
dt_bool_exists : Axiom Exists (x: (data_type?[Semantics_bool])): true
dt_bool : (data_type?[Semantics_bool])
```

Model

```
dt_bool : (data_type?[Semantics_bool]) = (#
  size := 1,
  to_byte := Lambda(b : bool, a : Address) : IF b Then (: 1 :) Else (: 0 :) Endif,
  from_byte := Lambda(bl : list[Byte], a : Address) :
                 IF bl = (: 1 :) Then up(true)
                 Elsif bl = (: 0 :) Then up(false)
                 Else bottom   Endif    #)
```

I. Fiasco overview
II. Insufficiency of strongly-typed languages
III. Semantics of C++ data types

IV. Semantics for abrupt termination and jumps

V. Duff's device
VI. Conclusion

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 possible

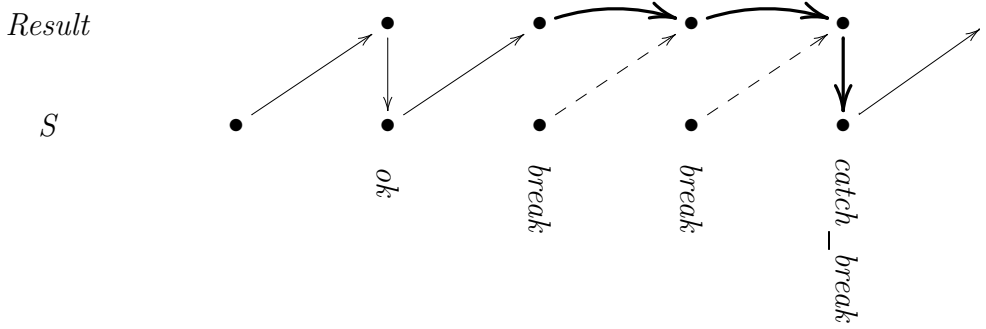
Switch

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

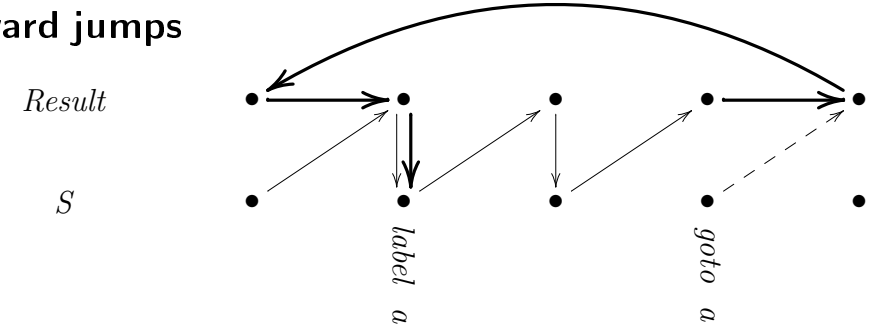
Recall possible nonsense:

```
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 $\mathit{Result} \longrightarrow \mathit{Result}$
- Result is a disjoint union
- defined as follows (with current memory S, set of labels $\mathbb{L}$):

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

- Semantic Equations

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

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

[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.

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

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

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

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

$$
\begin{array}{rcl}
\mathit{iter}\ \mathit{bo}\ i & = & \underbrace{\mathit{bo} \circ \mathit{bo} \circ \cdots \circ \mathit{bo}}_{i+1 \text{ times}} \\
\mathit{term}? & : & \mathit{Result} \longrightarrow \mathit{bool} \\
\mathit{term}? & = & \left| \begin{array}{lcl} \mathit{goto}(s) & \mapsto & \mathit{false} \\ x & \mapsto & \mathit{true} \end{array} \right. \\
\mathit{hangs}? & : & (\mathit{Result} \to \mathit{Result}) \times \mathit{Result} \longrightarrow \mathit{bool} \\
\mathit{hangs}?(\mathit{bo}, x) & = & \left\{ \begin{array}{ll} \mathit{true} & \text{if } \forall i \in \mathbb{N} \,.\, \mathit{term}?(\mathit{iter}\ \mathit{bo}\ i\ x) = \mathit{false} \\ \mathit{false} & \text{otherwise} \end{array} \right. \\
[\![\mathit{block}]\!] & : & S \longrightarrow \mathit{Result} \\
[\![\mathit{block}]\!](s) & = & \left\{ \begin{array}{ll} \mathit{hang} & \text{if } \mathit{hangs}?([\![\mathit{block}]\!], \mathit{ok}(s)) = \mathit{true} \\ \mathit{iter}\ [\![\mathit{block}]\!]\ n\ (\mathit{ok}\ s) & \text{otherwise} \\ \quad \text{where } n = \min\{i \in \mathbb{N} \mid \mathit{term}?(\mathit{iter}\ [\![\mathit{block}]\!]\ i\ (\mathit{ok}\ s))\} & \end{array} \right.
\end{array}
$$

- remarkably simple denotational semantics
  - no CPO's
  - no fixed points
  - no continuations necessary
  - relies only on disjoint unions, case distinctions 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)

I. Fiasco overview
II. Insufficiency of strongly-typed languages
III. Semantics of C++ data types
IV. Semantics for abrupt termination and jumps

V. Duff's device

VI. Conclusion

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.

Translate source code into PVS

```
rounds : posnat = 1
i : posnat = 2

duff(source, dest : posnat, count : nat) : [Result[State, Unit] -> Result[State,Unit]] =
  write_int(rounds, div(const_int(count), const_int(8))) ##
  write_int(i, const_int(0)) ##
  int_switch_stm( rem( const_int(count), const_int(8)),
      int_case(0) ##
              gwhile_stm(
                 const_int(0) < post_decr_const(rounds),
                      write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(7) ##  write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(6) ##  write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1)) ##
           ..................
      int_case(2) ##  write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(1) ##  write_int_array(dest, read_int(i), read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1))
              )  % end gwhile_stm
  ) % end int_switch_stm
```

Proof it

```
duff_total : Lemma
  Forall(s : State, source, dest : posnat, count : nat) :
    duff_var_ok(source, dest, count,s) Implies
      duff(source, dest, count)(ok(s,unit)) =
      ok(s WITH [
          'vars := Lambda(j : posnat) :
              IF j = rounds Then int(−1)
              Elsif j = i Then int(count)
              Elsif cell_in_array(s, dest)(j) And
                       index_from_cell(s, dest, j) < count
                  Then
                       s'vars(get_array(s'vars(source))
                                'fields(index_from_cell(s, dest, j)))
              Else s'vars(j)
              Endif
        ], unit)
```

I. Fiasco overview
II. Insufficiency of strongly-typed languages
III. Semantics of C++ data types
IV. Semantics for abrupt termination and jumps
V. Duff's device

VI. Conclusion

Working Semantics for "dirty" programs

- type casts
- abrupt termination
- goto, switch
- simple compositional denotational semantics (no continuations)
- Duff's device verified

other approaches

- object code verification [1]
- ASM Semantics [2, 3]
- non-deterministic semantics of C in HOL [4, 5]

[1] M. Wahab. Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Dep. of Comp. Sci., University of Warwick, 1998.

[2] 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.

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

[4] 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.

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