Towards Formalizing EXE's, DLL's and all that

Nick Benton, **Andrew Kennedy**
(Microsoft Research Cambridge)

Interns: Jonas Jensen (ITU), Valentin Robert (UCSD), Pierre-Evariste Dagand (INRIA), Jan Hoffman (Yale)

Our dream

Highest assurance software correctness

for

machine code programs

through

machine-assisted proof

"Prove what you run"

One tool: Coq

* **Model (sequential, 32-bit, subset of) x86 in Coq:**
  bits, bytes, memory, instruction decoding, execution
* **Generate x86 programs from Coq:**
  assembly syntax in Coq, with macros, run assembler in Coq to produce machine code, even EXEs and DLLs
* **Specify x86 programs in Coq:**
  separation logic for low-level code
* **Prove x86 programs in Coq:**
  tactics and manual proof for showing that programs meet their specifications

x86 assembly code

```
(* Argument in EBX *)
letproc fact :=
  MOV EAX, 1;;
  MOV ECX, 1;;
    (* while ECX <= EBX *)
    while (CMP ECX, EBX) CC_LE true (
      MUL ECX;; (* Multiply EAX by ECX *)
      INC ECX
    )
in
  LOCAL format;
    MOV EBX, 10;; callproc fact;;
    MOV EDI, printfSlot;;
    call_cdecl3 [EDI] format EBX EAX;;
    MOV EBX, 12;; callproc fact;;
    MOV EDI, printfSlot;;
    call_cdecl3 [EDI] format EBX EAX;;
    RET 0;;
  format:;;
    ds "Factorial of %d is %d";; db #10;; db #0.
```

Macro for local procedure

Macro for while loop

Intel instruction syntax

Macro for calling external C code

Scoped labels

Inline string data

Inline byte data

X86 assembly code, in Coq

Actually, "just" a definition in Coq

```
Definition main (printfSlot: DWORD) :=
  (* Argument in EBX *)
  letproc fact :=
    MOV EAX, 1;;
    MOV ECX, 1;;
      (* while ECX <= EBX *)
      while (CMP ECX, EBX) CC_LE true (
        MUL ECX;; (* Multiply EAX by ECX *)
        INC ECX
      )
  in
    LOCAL format;
      MOV EBX, 10;; callproc fact;;
      MOV EDI, printfSlot;;
      call_cdecl3 [EDI] format EBX EAX;;
      MOV EBX, 12;; callproc fact;;
      MOV EDI, printfSlot;;
      call_cdecl3 [EDI] format EBX EAX;;
      RET 0;;
    format:;;
      ds "Factorial of %d is %d";; db #10;; db #0.
```

Macros are "just" parameterized Coq definitions

```
Definition while (ptest: program)
  (cond: Condition) (value: bool)
  (pbody: program) : program :=
  LOCAL BODY; LOCAL test;
    JMP test;;
  BODY:;; pbody;;
  test:;;
    ptest;;
    JCC cond value BODY.
```

Assembler syntax is "just" user-defined Coq notation

Scoped labels "just" use Coq binding

In previous work…

POPL 2013

High-Level Separation Logic for Low-Level Code

Jonas B. Jensen
IT University of Copenhagen
jobr@itu.dk

Nick Benton Andrew Kennedy
Microsoft Research Cambridge
{nick,akenn}@microsoft.com

Low-level program logic for assembly; proof of soundness wrt machine model

Program specifications; program logic tactics; proofs of correctness for assembly programs

Higher-level languages; compilers; compiler correctness

Model of x86 machine: binary reps, memory, instruction decoding, instruction execution

Assembly-code representation; assembler; proof of correctness

Simple macros (if, while); User macros; DSLs (e.g. regexps)

PPDP 2013

Coq: The world's best macro assembler?

Andrew Kennedy Nick Benton
Microsoft Research
{akenn,nick}@microsoft.com

Jonas B. Jensen
ITU Copenhagen
jobr@itu.dk

Pierre-Evariste Dagand
University of Strathclyde
dagand@cis.strath.ac.uk

Today's talk

Extend generation, specification and verification of x86 machine code to

* Generate binary link formats: EXEs and DLLs for Windows (i.e. **practice**)
* Specify and verify behaviour of EXEs and DLLs
* (Future work) Specify and verify loading and dynamic linking of EXEs and DLLs

But first, a quick overview of our x86 machine model.

Model x86

* Use Coq to construct a “reference implementation” of sequential x86 instruction decoding and execution

```
| CALL src =>
  let! oldIP = getRegFromProcState EIP;
  let! newIP = evalSrc src;
  do! setRegInProcState EIP newIP;
  evalPush oldIP

| RET offset =>
  let! oldSP = getRegFromProcState ESP;
  let! IP' = getDWORDFromProcState oldSP;
  do! setRegInProcState ESP (addB (oldSP+#4) (zeroExtend 16 offset));
  setRegInProcState EIP IP'
```

Example fragment: semantics of call and return.

Design an assembly language

* Define datatype of programs, with sequencing, labels, and scoping of labels

```
Inductive program :=
  prog_instr (c: Instr)
| prog_skip | prog_seq (p1 p2: program)
| prog_declabel (body: DWORD -> program)
| prog_label (l: DWORD)
```

* Use Coq variables for object-level ‘variables’ (labels), à la higher-order abstract syntax

```
Notation "'LOCAL' l ';' p" := (prog_declabel (fun l => p))
```

Build an assembler (1)

* First implement instruction encoder:

```
| PUSH (SrcI c) =>
  if signTruncate 24 (n:=7) c is Some b
  then do! writeNext #x"6A"; writeNext b
  else do! writeNext #x"68"; writeNext c

| PUSH (SrcR r) =>
  writeNext (PUSHPREF ## injReg r)

| PUSH (SrcM src) =>
  do! writeNext #x"FF";
  writeNext (#6, RegMemM true src)
```

Build an assembler (2)

* Using instruction encoder, implement multi-pass assembler that determines a consistent assignment for scoped labels

```
assemble
      : DWORD -> program -> option (seq BYTE)
```

* Prove "round-trip" lemma stating that instruction decoding is inverse wrt instruction encoding
* Extend this to a full round-trip theorem for the assembler

Design a logic

* It's usual to use a *program logic* such as *Hoare logic* to specify and reason about programs

$\{P\}\ C\ \{Q\}$

Precondition

Postcondition

Command

* Recent invention of *separation logic* makes reasoning about pointers tractable
* But still not appropriate for machine code
  * Machine code programs don't "finish" (what postcondition?)
  * Code and data are all mixed up ("command" is just bytes in memory), also code can be "higher-order" with code pointers
* We have devised a new separation logic that solves all these problems, embedded it in Coq, and proved it sound with respect to the machine model

Example:
Specifying memory allocation

If it is safe to exit through *failLabel* or *j*…

```
(
  safe @ (EIP ~= failLabel ** EDI?) //\\
  safe @ (EIP ~= j ** Exists pb,
                 EDI ~= pb +# bytes **
                 memAny pb (pb +# bytes))
-->>
  safe @ (EIP ~= i ** EDI?)
)
@ (ESI? ** OSZCP_Any ** allocInv heapInfo)
<@ (i -- j :-> inlineAlloc heapInfo bytes failLabel).
```

…such that (at j), EDI points just beyond accessible memory block of size *bytes*…

…then it is safe to enter at i

…under the assumption that memory at i..j decodes to allocator code, ESI and flags are arbitrary, and a data invariant is maintained

Trivial implementation of allocator

```
Definition inlineAlloc heapInfo
 (bytes:nat) (failLabel:DWORD) : program :=
  mov ESI, heapInfo;;
  mov EDI, [ESI];;
  add EDI, bytes;;
  jc failLabel;;
  cmp [ESI+4], EDI;;
  jc failLabel;;
  mov [ESI], EDI.
```

```
Definition allocInv (heapInfo:DWORD) :=
  Exists heapPtr:DWORD,
  Exists heapLimit:DWORD,
  heapInfo :-> heapPtr **
  heapInfo+#4 :-> heapLimit **
  memAny heapPtr heapLimit.
```

Prove some theorems

- We have developed Coq *tactics* to help prove that programs behave as specified
- Sometimes routine, sometimes careful reasoning required. Example proof fragment:

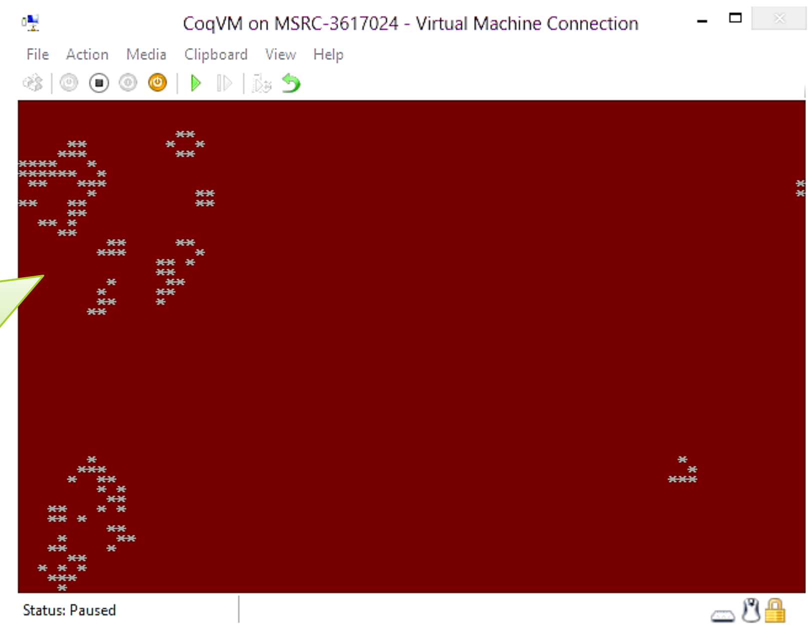
```
(* add EDI, EDX *)
eapply basic_seq; first eapply basic_basic;
first apply ADD_RR_ruleAux; sbazooka.

(* shl ECX, 1 *)
eapply basic_seq; first eapply basic_basic;
first apply SHL_RI_rule; sbazooka.

(* add EDI, ECX *)
eapply basic_basic;
first apply ADD_RR_ruleAux; rewrite /regAny; sbazooka.
```

Put it all together

1. Use Coq to produce raw bytes, link with a small boot loader, to produce a bootable image
2. Under assumptions about state of machine following boot loading, prove that program meets spec
3. Run!



Game of life, written in assembler using Coq, running on bare metal!

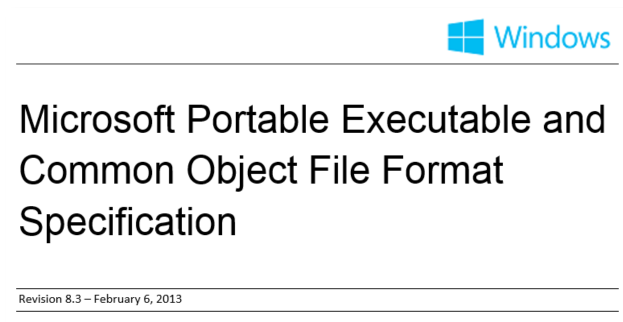
Executables

* That’s all well and good but
  * We’d like to formalize the process of loading programs, and support dynamic linking, and
  * Rather than booting the machine (or a VM) it would be nice to experiment on an existing OS e.g. Windows
  * Also good to test our ideas on linking and loading using existing formats
* So: model EXE’s, DLL’s, loading and dynamic linking

What's in an executable?

Some machine code, with an entry point, preferred base address, and…

* Several **sections** (code, data, r/o data, thread local data, etc.)
* **Relocation** information (if not loaded at preferred base address)
* **Imports**, by name or number
* **Exports** (if executable is a DLL)
* A *lot* of metadata
* Legacy cruft (e.g. MSDOS stub!)
* Informally documented in a ~100 page spec


Windows

Microsoft Portable Executable and Common Object File Format Specification

Revision 8.3 – February 6, 2013

What's in an executable? Let's look inside

```c
static void main() {
  printf("hello, world.\n");
}
```

compile & link

```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789abcdef
00000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000  MZ..............
00000010: b800 0000 0000 0000 4000 0000 0000 0000  ........@.......
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 e000 0000  ................
00000040: 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468  ........!..L.!Th
00000050: 6973 2070 726f 6772 616d 2063 616e 6e6f  is program canno
00000060: 7420 6265 2072 756e 2069 6e20 444f 5320  t be run in DOS 
00000070: 6d6f 6465 2e0d 0d0a 2400 0000 0000 0000  mode....$.......
00000080: f238 02ab b659 6cf8 b659 6cf8 b659 6cf8  .8...Yl..Yl..Yl.
00000090: b659 6df8 fb59 6cf8 4a2e d5f8 b559 6cf8  .Ym..Yl.J....Yl.
000000a0: 729c a3f8 af59 6cf8 729c a1f8 bc59 6cf8  r....Yl.r....Yl.
000000b0: 729c a2f8 ef59 6cf8 919f a2f8 b759 6cf8  r....Yl......Yl.
000000c0: 919f a0f8 b759 6cf8 5269 6368 b659 6cf8  .....Yl.Rich.Yl.
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 5045 0000 4c01 0400 e826 e152 0000 0000  PE..L....&.R....
```

dumpbin /all

```
    7000 size of code
    5E00 size of initialized data
       0 size of uninitialized data
    1244 entry point (00401244)
    1000 base of code
    8000 base of data
  400000 image base (00400000 to 0040EFFF)
    1000 section alignment
     200 file alignment
    6.00 operating system version
    0.00 image version
    6.00 subsystem version
       0 Win32 version
    F000 size of image
     400 size of headers
       0 checksum
       3 subsystem (Windows CUI)
    8140 DLL characteristics
           Dynamic base
           NX compatible
           Terminal Server Aware
  100000 size of stack reserve
    1000 size of stack commit
```

Example .EXE, in Coq

```
Definition winfact :=
  IMPORTDLL "msvcrt.dll";
  IMPORT "printf" as printfSlot;
  SECTION CODE
    main printfSlot.

Compute makeEXE #x"00760000" "winfact.exe" winfact.
```

Import a Dynamic Link Library

Import a named function from the DLL

Declare a code *section* containing our factorial code

Generate the bytes of the .EXE at a given load address!

Developer Command Prompt for VS2012

```
C:\coqx86\trunk>build x86\win\winfact.exe
'Makefile.deps' is up-to-date
        coqtop -dont-load-proofs -quiet [illegible] 86/lib/
regexp -I x86/win -I contribs -I contribs/ATBR -I [illegible] in/win
fact.hex
        tools\hexbin x86/win/winfact.hex x86/win/winfact.exe

C:\coqx86\trunk>x86\win\winfact
Factorial of 10 is 3628800
Factorial of 12 is 479001600

C:\coqx86\trunk>_
```

Compile…

…and run!

Example DLL
counter.dll

Export module-level labels by name

Declare a module-level label without exporting it

Read/write data section

```
Example counterDLL :=
  GLOBAL Inc as "Inc";
  GLOBAL Get as "Get";
  GLOBAL Counter;
  SECTION CODE
    Inc:;;  MOV ECX, Counter;; INC [ECX];; RET 0;;
    Get:;;  MOV ECX, Counter;; MOV EAX, [ECX];; RET 0;
  SECTION DATA
    Counter:;; dd #0.

Compute makeDLL #x"00AC0000" "counter.dll" counterDLL.
```

Example client usecounter.exe

```
Example useCounterCode :=
  IMPORTDLL "msvcrt.dll";
  IMPORT "printf" as printfSlot;
  IMPORTDLL "counter.dll";
  IMPORT "Inc" as incSlot; IMPORT "Get" as getSlot;
  SECTION CODE
    LOCAL formatString;
    MOV EDI, incSlot;;    CALL [EDI];; CALL [EDI];;
    MOV EDI, getSlot;;    CALL [EDI];;
    PUSH EAX;;
    MOV EBX, formatString;; PUSH EBX;;
    MOV EDI, printfSlot;; CALL [EDI];;
    ADD ESP, 8;;
    RET 0;;
  formatString:;;
    ds "Got %d";; db #10;; db #0.

Compute makeEXE #x"12E30000" "usecounter.exe" useCounterCode.
```

Import Get from counter.dll

Call indirect through Get's "slot"

The messy details

* Our assembly datatype and assembler give us all the mechanisms we need to generate the structures found in EXE's and DLL's
  * Byte, word, string representations
  * RVAs (Relative Virtual Address)
  * Padding
  * Alignment constraints
  * Bitfields
  * Multi-pass fixed-point iteration to deal with forward references
* One small annoyance: file image not identical to in-memory image (e.g. alignment of sections); RVAs wrt in-memory image
  * Hack: add "skip" primitive in our writer monad to advance the assembler's "cursor" without producing any bytes

Exports and imports

Exports

Logically: a list of ⟨name,address⟩ pairs

Imports

Logically: for each imported DLL,

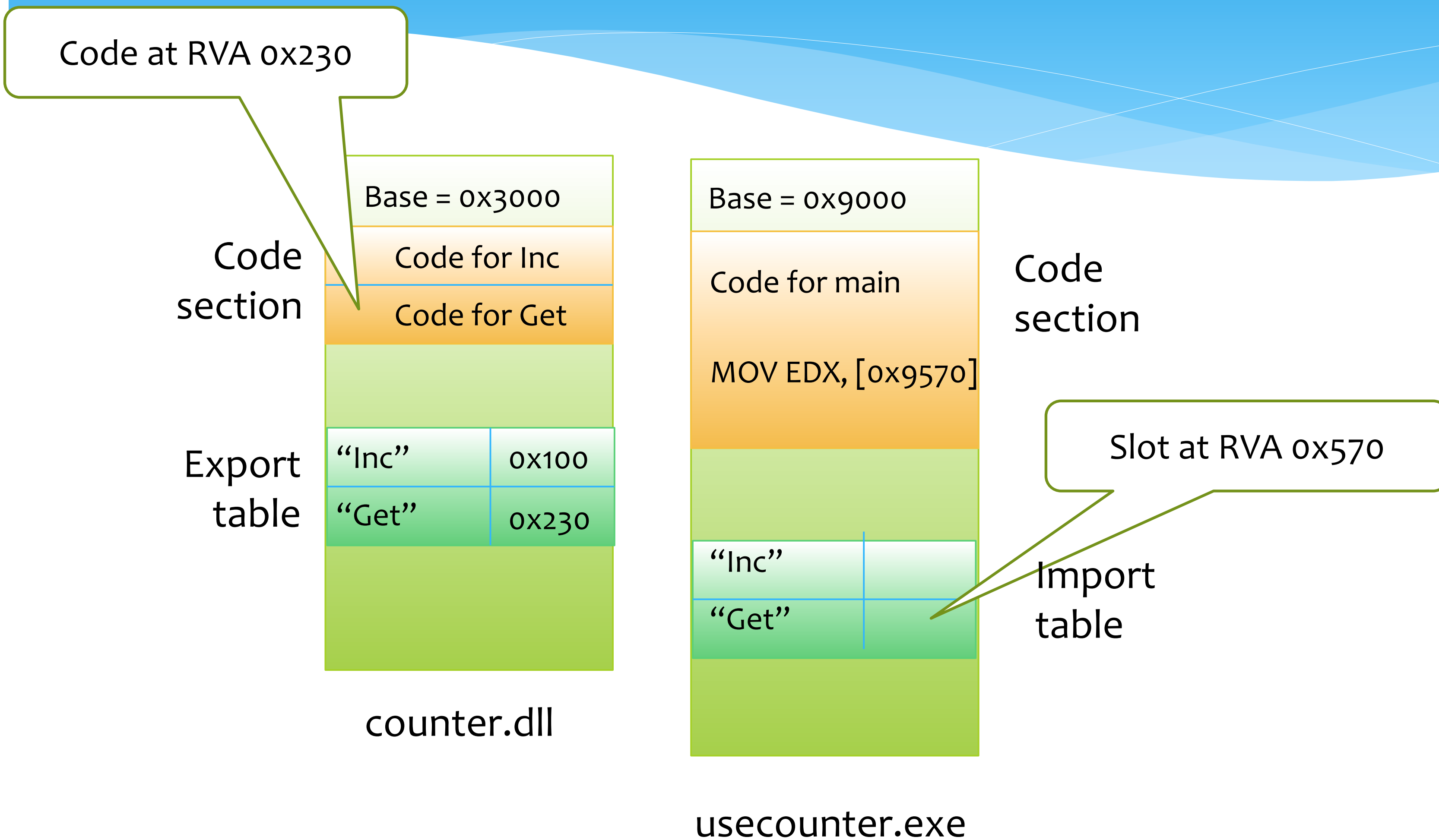
* Its name
* A list of imported symbols (by name or *ordinal*)
* A list of slots, one for each imported symbol: the Import Address Table or IAT

In binary format, this is all somewhat messier!

Relocateable code

* Some x86 code is position independent e.g. makes use of PC-relative offsets (jumps)
* But much is not: especially on 32-bit, it’s hard to refer to global *data* in position independent way
* So: executables have a “preferred *base* address”
* If not loaded at this address, absolute addresses embedded in code and data must be *rebased* i.e. patched at load-time
* The executable lists these in a special “.reloc” section

What does the OS loader do? Before: in-file



What does the OS loader do? After loading: in-memory

counter.dll

Starting at address 0x3000

Code section

Base = 0x3000
Code for Inc
Code for Get

Export table

"Inc"	0x100
"Get"	0x230

usecounter.exe

Starting at address 0x9000

Code section

Base = 0x9000
Code for main MOV EDX, [0x9570]

Import table

"Inc"	0x3100
"Get"	0x3230

Patching of instructions

* We want to relocate addresses ("rebasing") and perhaps link modules (in some non-Windows loader) by in-place update of instructions
* Encodings matter. Prove lemmas such as

```
Lemma PUSH_decoding (p addr: DWORD) q :
     p -- q :-> PUSH addr -|-
     p -- q :-> (#x"68", addr) \\//
    (Exists b:BYTE, signTruncate _ (n:=7) addr = Some b
 /\\ p -- q :-> (#x"6A", b)).
```

(Towards) Specifying calling conventions

* "fastcall" calling convention for function of one argument (passed in ECX) and one result (in EAX)

```
Definition fastcall_nonvoid1_spec (f: DWORD) (FS: FunSpec (mkFunSig 1 true)) : spec :=
  Forall arg:DWORD,
  Forall sp:DWORD,
  Forall iret:DWORD,
  (
    safe @ (EIP ~= iret ** EAX ~= fst (post FS arg) ** ECX? **
            ESP ~= sp    ** sp-#4 :-> ?:DWORD ** snd (post FS arg)) -->>

    safe @ (EIP ~= f     ** EAX?              ** ECX ~= arg **
            ESP ~= sp-#4 ** sp-#4 :-> iret    ** pre FS arg)
  )
  @ (EDX? ** OSZCP_Any).
```

What's to do?

* Separately specify different modules; prove correctness of combination, already loaded and with imports resolved
* Model the loading process itself
* Implement a small loader, in machine code using Coq, with export/import resolution
* Prove its correctness