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

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

(* 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.
X86 assembly code, in Coq

```
Definition main (printfSlot: DWORD) :=
  (* Argument in EBX *)
let proc 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.
```

Actually, “just” a definition in Coq

Macros are “just” parameterized Coq definitions

Assembler syntax is “just” user-defined Coq notation

Scoped labels “just” use Coq binding
In previous work...

- **High-Level Separation Logic for Low-Level Code**
  Jonas B. Jensen  
  IT University of Copenhagen  
  job@itu.dk  
  Nick Benton  
  Andrew Kennedy  
  Microsoft Research Cambridge  
  {nick,akenn}@microsoft.com

- **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)

- **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

- **Coq: The world’s best macro assembler?**
  Andrew Kennedy  
  Microsoft Research  
  {akenn,nick}@microsoft.com  
  Jonas B. Jensen  
  ITU Copenhagen  
  job@itu.dk  
  Pierre-Evariste Dagand  
  University of Strathclyde  
  dagand@cis.strath.ac.uk
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

```coq
| 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

\[
\text{Inductive program := } \\
\quad \text{prog_instr (c: Instr)} \\
\quad | \text{prog_skip} \\
\quad | \text{prog_seq (p1 p2: program)} \\
\quad | \text{prog_declabel (body: DWORD -> program)} \\
\quad | \text{prog_label (l: DWORD)} \\
\]

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

\[
\text{Notation "'LOCAL' l ';' p" := (prog_declabel (fun l => p))}
\]
* First implement instruction encoder:

```latex
\begin{verbatim}
| 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)
\end{verbatim}
```
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

- 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`...

```c
    (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.
We have developed Coq tactics to help prove that programs behave as specified.

* Sometimes routine, sometimes careful reasoning required.

Example proof fragment:

```coq
(* add EDI, EDX *)
eapply basic_seq; first eapply basic_basic;
fapply ADD RR ruleAux; sbazooka.

(* shl ECX, 1 *)
eapply basic_seq; first eapply basic_basic;
fapply SHL RI rule; sbazooka.

(* add EDI, ECX *)
eapply basic_basic;
fapply 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!
**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
What’s in an executable? Let’s look inside

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

compile & link
dumpbin /all

| 7000 size of code |
| 5800 size of initialized data |
| 0 size of uninitialized data |
| 1244 entry point (00401244) |
| 1000 base of code |
| 8000 base of data |
| 4000000 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 |
| 400 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

Import a Dynamic Link Library

Import a named function from the DLL

Declare a code section containing our factorial code

Declare a code section containing our factorial code

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

Compile...

...and run!
Example DLL

counter.dll

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 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 \#"12E30000" "usecounter.exe" useCounterCode.
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
Logically: a list of \langle\text{name}, \text{address}\rangle \text{ pairs}

Imports
Logically: for each imported DLL,
* Its name
* A list of imported symbols (by name or \textit{ordinal})
* A list of slots, one for each imported symbol: the Import Address Table or IAT

In binary format, this is all somewhat messier!
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

Code at RVA 0x230

Base = 0x3000
Code for Inc
Code for Get

Export table

“Inc” 0x100
“Get” 0x230

counter.dll

usecounter.exe

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

Import table

Slot at RVA 0x570
What does the OS loader do?
After loading: in-memory

Starting at address 0x3000

Code section

Base = 0x3000
Code for Inc
Code for Get

Export table

“Inc” 0x100
“Get” 0x230

counter.dll

Starting at address 0x9000

Code section

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

Import table

“Inc” 0x3100
“Get” 0x3230

usecounter.exe
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