DEPENDABLE COMPUTING

Proving Security Properties in Software of Unknown Provenance

SOUND STATIC ANALYSIS FOR SECURITY WORKSHOP

Ashlie B. Hocking[1]

2018 June 2017

In collaboration with: Ben Rodes[1], John Knight[1], Jack Davidson[2], and Clark Coleman[2]

[1]Dependable Computing

[2]Zephyr Software

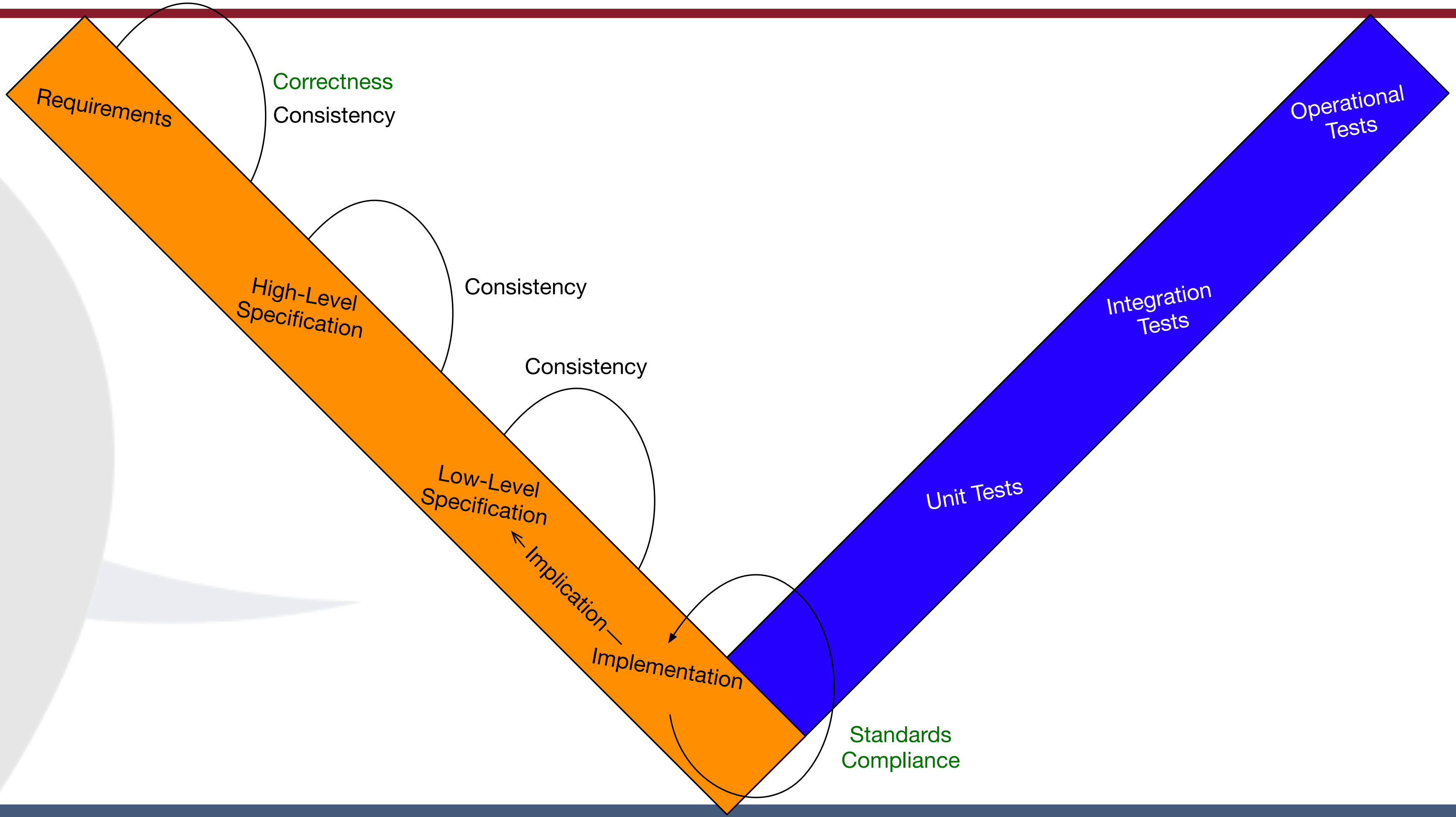
Formal Methods

Q: When should formal methods be applied?

A: As soon as you can!

Amey, P. (2002). Correctness by Construction: Better can also be Cheaper. *CrossTalk: the Journal of Defense Software Engineering*, 2, 24-28.

Formal Methods and the V-Model



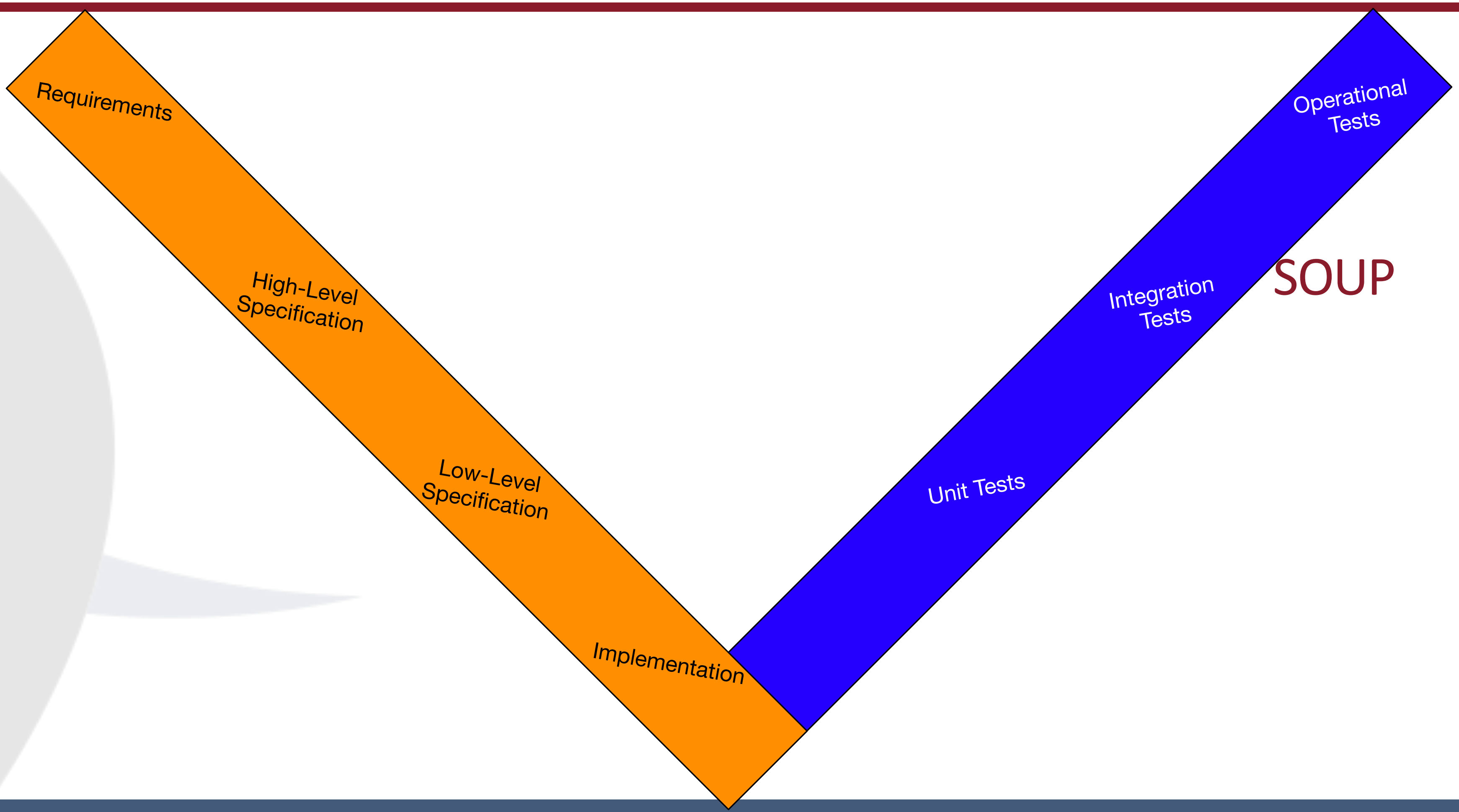
Software of Unknown Provenance (SOUP)

MS15-078

Heartbleed

- Formal methods are best when applied at the beginning
- Embedded systems may rely on software with no source code or with source code contributed by unknown authors
  - Even when you have source code, compiler can introduce errors
- New software might use existing libraries of unknown provenance
- How can we leverage formal methods with binary code?

Formal Methods and the V-Model



Formal Methods

Q: When should formal methods be applied?

A: As soon as *reasonably practicable*!

If we are given an existing software binary (library or executable) to use, how should we apply formal methods to it?

Is It Too Late?

Has the safety/security “horse” already left the stable?

Goal and Approach

Goal: Prove Specific Security Properties about software for which we do not have the source code

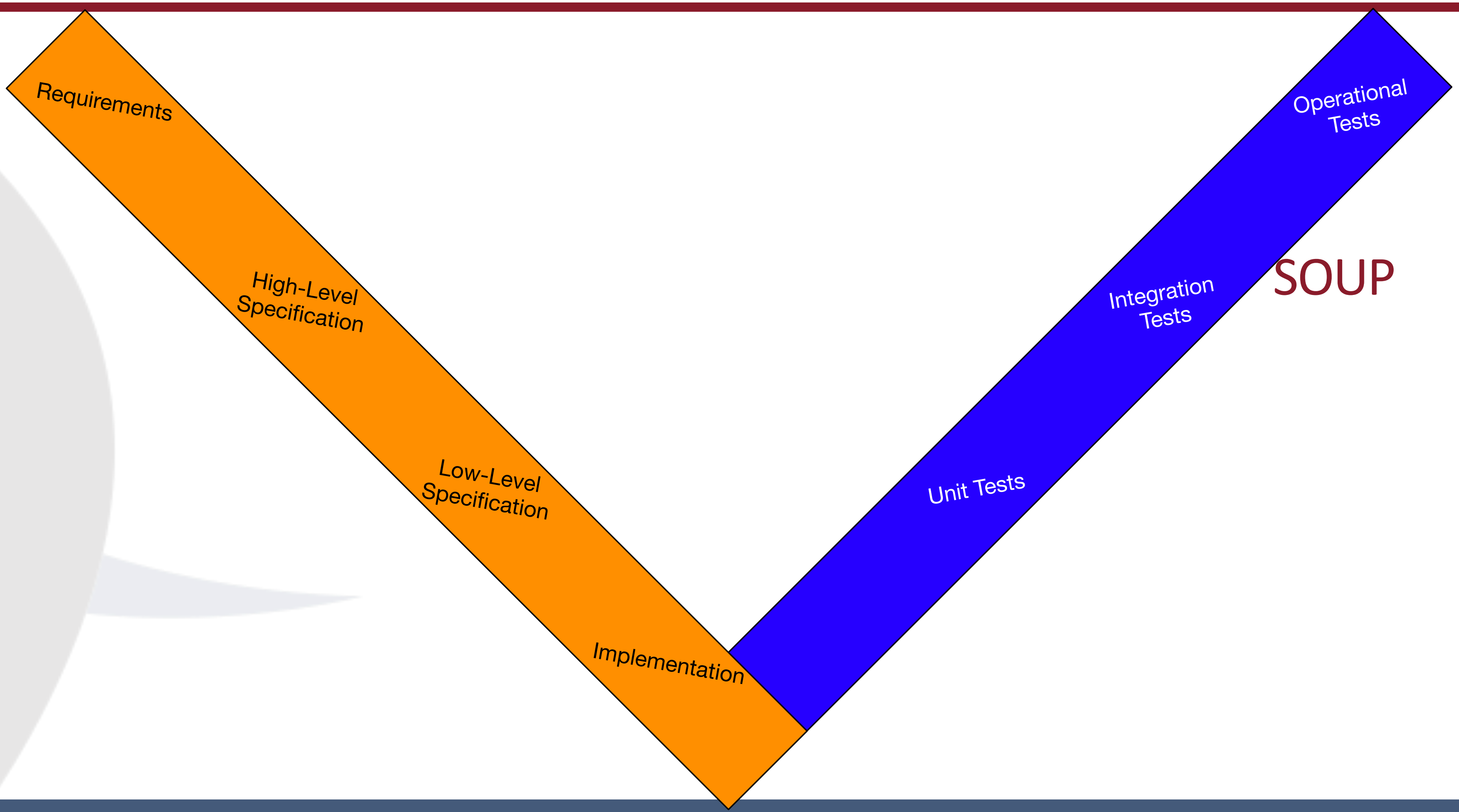
Approach:

1. Generate SPARK Ada code from the binary software
2. Prove properties about the generated SPARK Ada code
3. Insert guards for unsafe binaries

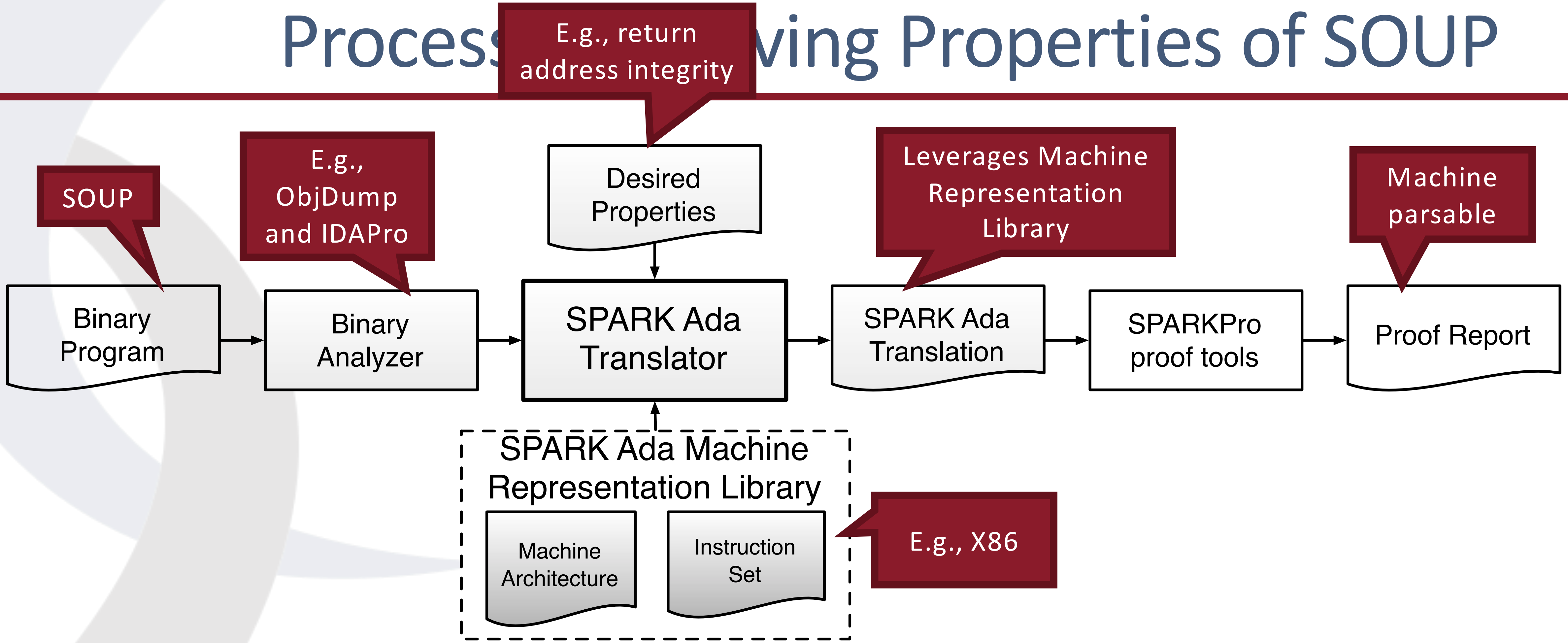
Why SPARK Ada and SPARKPro?

- SPARK Ada language
  - Designed for proof
  - Familiar
  - Simple
- SPARKPro
  - Proof tools provide capability to establish proofs
    - cvc4, z3, alt-ergo (by default, but also coq, isabelle, pvs…)
  - Industrial strength support
  - Can generate an executable for testing

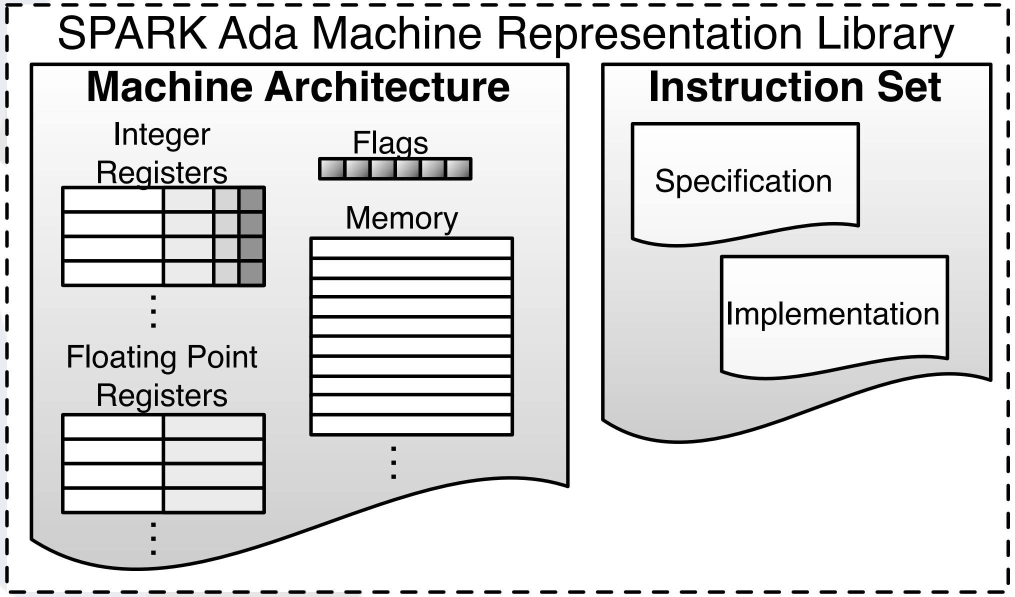
Formal Methods and the V-Model



Process[illegible]ving Properties of SOUP

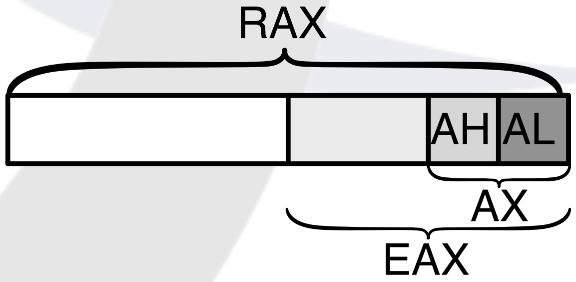


Details of the Representation Library




```
type Mem_Array is array (Unsigned64) of Unsigned8;
Memory: Mem_Array := Mem_Array'(others => 0);
function ReadMem16(addr: in Unsigned64) return Unsigned16 with
  Global => (Input => Memory),
  Post => (((ReadMem16'Result and 16#00FF#) = Unsigned16(Memory(addr))) and
           ((ReadMem16'Result and 16#FF00#) = Unsigned16(Memory(addr+1))*16#100#));
procedure WriteMem16(addr : in Unsigned64; Val : in Unsigned16) with
  Global => (In_Out => Memory),
  Post => ((ReadMem16(addr) = Val) and (for all i in Unsigned64 =>
   (if ((i /= addr) and (i /= addr + 1)) then (Memory(i) = Memory'Old(i)))));
```

```
procedure setnbe_CL with
  Global => (Input => (ZeroFlag, CarryFlag), In_Out => RCX),
  Post => (if ((not CarryFlag) and (not ZeroFlag)) then (CL = 1) else (CL = 0));
```




```
function EAX return Unsigned32 with
  Global => (Input => RAX),
  Post => (EAX'Result = Unsigned32(RAX and 16#00000000FFFFFFFF#));
procedure Write_EAX(Val : in Unsigned32) with
  Global => (In_Out => RAX),
  Post => ((EAX = Val) and ((RAX and 16#FFFFFFFF00000000#) = (16#0000000000000000#)));
```

SPARK Ada Translation

```
procedure zero_array is
begin
  --100000ed4: test esi,esi
  X86.ZeroFlag := (X86.ESI = 0);
  X86.SignFlag := (X86.ESI > X86.MaxS
  X86.OverflowFlag := False;
  --100000ed6: jle 100000eec <_zero_arr
  if (X86.ZeroFlag or X86.SignFlag /= X
    --100000eec: f3 c3 repz ret
    X86.RSP := X86.RSP + 8;
    return;
  end if;
  --100000ed8: mov eax,0x0
  X86.Write_EAX(0);
  loop
    --100000edd: DWORD PTR [rdi+rax*4
    X86.WriteMem32(X86.RDI +(X86.RA
    --100000ee4: add rax,0x1
    X86.RAX := X86.RAX + 1;
    --100000ee8: cmp esi,eax
    X86.ZeroFlag := ((X86.ESI - X86.EAX) = 0);
    X86.SignFlag := (X86.ESI < X86.EAX);
    X86.OverflowFlag := ((X86.SignFlag and (X86.EAX > X86.MaxSignedInt32) and
     (X86.ESI <= X86.MaxSignedInt32)) or ((not X86.SignFlag) and
     (X86.ESI > X86.MaxSignedInt32) and (X86.EAX <= X86.MaxSignedInt32)));
    --100000eea: jg 100000edd <_zero_array+0x9>
    exit when(not(X86.ZeroFlag=False and X86.SignFlag=X86.OverflowFlag));
  end loop;
  --100000eec: repz ret
  X86.RSP := X86.RSP + 8;
  return;
end zero_array;
```

Pre-condition required for return address integrity

Additional information for downstream analysis

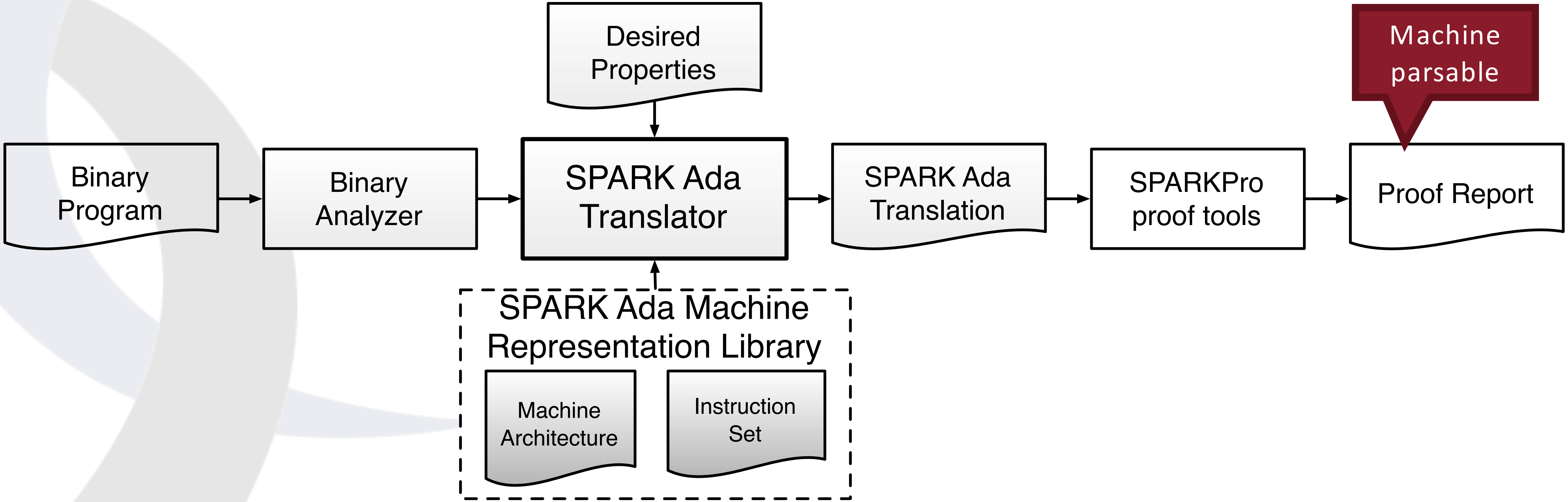
Stack pointer incremented by 8

Return address integrity

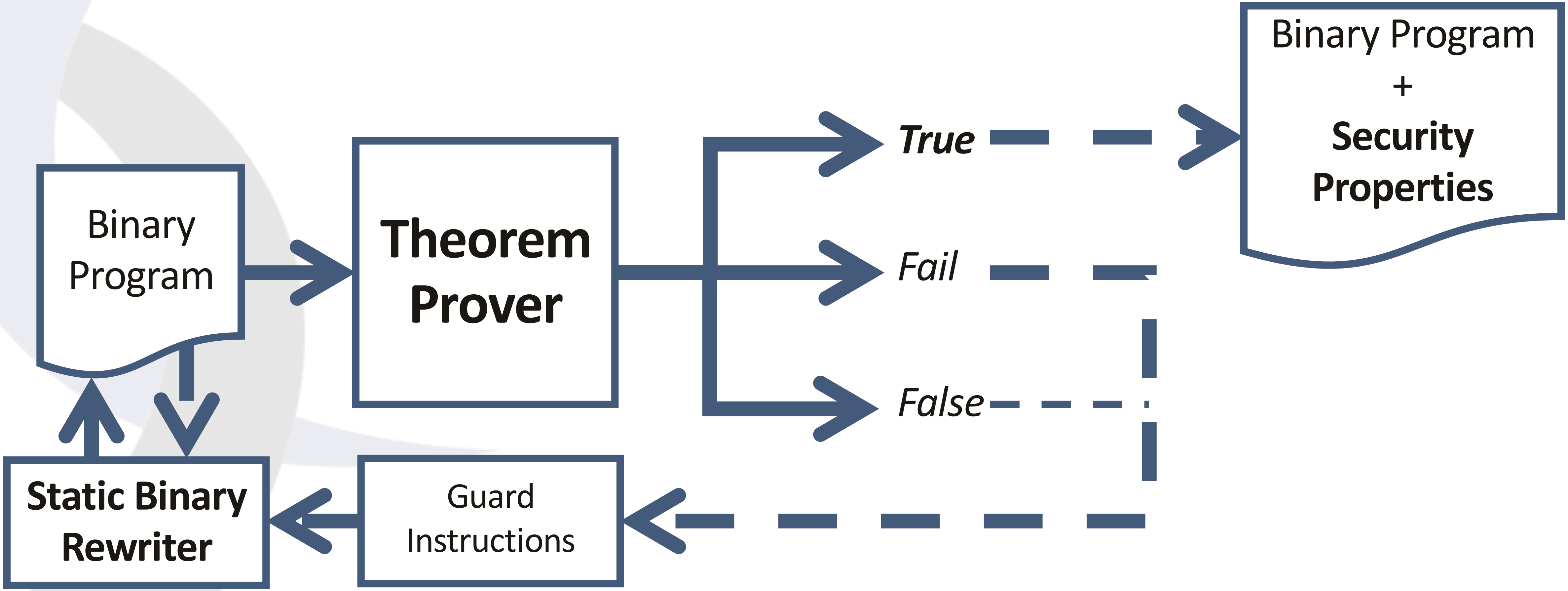
```
procedure zero_array with
  Global => (In_Out => (X86.Memory, X86.RSP, X86.RAX, X86.SignFlag,
                        X86.OverflowFlag, X86.CarryFlag, X86.ZeroFlag),
             Input => (X86.RSI, X86.RDI)),
  Pre => ((X86.RDI < Unsigned64'Last - Unsigned64(X86.ESI) * 4) and
         ((X86.RSP + 7 < X86.RDI) or (X86.RSP >= X86.RDI + Unsigned64(X86.ESI) * 4))),
  Post =>
         (for all i in Unsigned64 =>
          (if ((i < X86.RDI) or (i >= (X86.RDI + (Unsigned64(X86.ESI)*4))))
                 then X86.Memory(i) = X86.Memory'Old(i))) and
         (X86.RSP = (X86.RSP'Old + 8)) and
         (X86.Memory(X86.RSP'Old)     = X86.Memory'Old(X86.RSP'Old)) and
         (X86.Memory(X86.RSP'Old + 1) = X86.Memory'Old(X86.RSP'Old + 1)) and
         (X86.Memory(X86.RSP'Old + 2) = X86.Memory'Old(X86.RSP'Old + 2)) and
         (X86.Memory(X86.RSP'Old + 3) = X86.Memory'Old(X86.RSP'Old + 3)) and
         (X86.Memory(X86.RSP'Old + 4) = X86.Memory'Old(X86.RSP'Old + 4)) and
         (X86.Memory(X86.RSP'Old + 5) = X86.Memory'Old(X86.RSP'Old + 5)) and
         (X86.Memory(X86.RSP'Old + 6) = X86.Memory'Old(X86.RSP'Old + 6)) and
         (X86.Memory(X86.RSP'Old + 7) = X86.Memory'Old(X86.RSP'Old + 7));
```

```
void zero_array(int *array, int size) {
  for (int i = 0; i < size; i++) array[i] = 0;
}
```

Process for Proving Properties of SOUP



Completing The Proof



Guards and Proofs

- Guards can be quite effective
- Added code can require additional computational resources
  - Real-time constraints might be at risk
  - Embedded systems often have limited room for additional code
- Can we ***prove*** that software does not have a security violation?
  - If so, guards are not required for those sit
- When we cannot prove that software d urity violation…
  - Guards can be added to guarantee that the insecure situation is protected against

And then prove that the modified code does not have a security violation

Case Study

- Looked at 3 security properties:
  - The exit value in the RSP register is 8 larger than the entry value in the RSP register for all possible execution paths.
  - The argument to `setuid` (in `RDI`) is non-zero for every call to `setuid` for all possible execution paths.
  - The return address of a function is unmodified. Specifically, the 8 bytes in memory pointed to by the RSP register contain the same value when the function exits as they did when the function begins.
- Examined 11 programs, 2 of which used `setuid`
  - All 11 programs were able to prove correct stack pointer (RSP).
  - Both programs using setuid were proven to use it with non-zero values.
  - Proved unmodified return address in 5 of 7 programs instrumented for checking this property — the other 2 programs could possibly modify the return address.

Summary

- Advantages
  - Can prove security properties for SOUP without overhead of guards
  - Automatable
- Disadvantages
  - When proofs do not automatically discharge, manual proofs are difficult
- Future Work
  - Robust heuristics for automatically generating provable SPARK Ada representation
    - Assertions and loop invariants
  - Additional security properties

DEPENDABLE COMPUTING

Fin

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) under contract W31P4Q–14-C– 0086. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. The authors thank the software engineers of AdaCore, in particular Yannick Moy, for providing support.