BAP: A Binary Analysis Platform

David Brumley

Ivan Jager

Thanassis Avgerinos

Edward J. Schwartz

Carnegie Mellon University

3 Simple Lines

```
add %eax , %ebx   ; ebx = eax+ebx   ( sets OF, SF, ZF, AF, CF, PF)
shl %cl , %ebx    ; ebx = ebx << cl  ( sets OF, SF, ZF, AF, CF, PF)
jc error          ; jump to error if carry flag is set
```

Can you reach the error?

SHL Specification

SAL/SAR/SHL/SHR—Shift

Opcode***	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL *r/m8*, 1	A	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SAL *r/m8***, 1	A	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SAL *r/m8*, CL	B	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SAL *r/m8***, CL	B	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SAL *r/m8, imm8*	C	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SAL *r/m8***, *imm8*	C	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SAL *r/m16*, 1	A	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SAL *r/m16*, CL	B	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SAL *r/m16, imm8*	C	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SAL *r/m32*, 1	A	Valid	Valid	Multiply *r/m32* by 2, once.
REX.W + D1 /4	SAL *r/m64*, 1	A	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SAL *r/m32*, CL	B	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SAL *r/m64*, CL	B	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SAL *r/m32, imm8*	C	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SAL *r/m64, imm8*	C	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /7	SAR *r/m8*, 1	A	Valid	Valid	Signed divide* *r/m8* by 2, once.
REX + D0 /7	SAR *r/m8***, 1	A	Valid	N.E.	Signed divide* *r/m8* by 2, once.
D2 /7	SAR *r/m8*, CL	B	Valid	Valid	Signed divide* *r/m8* by 2, CL times.
REX + D2 /7	SAR *r/m8***, CL	B	Valid	N.E.	Signed divide* *r/m8* by 2, CL times.
C0 /7 *ib*	SAR *r/m8, imm8*	C	Valid	Valid	Signed divide* *r/m8* by 2, *imm8* time.
REX + C0 /7 *ib*	SAR *r/m8***, *imm8*	C	Valid	N.E.	Signed divide* *r/m8* by 2, *imm8* times.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D1 /7	SAR *r/m16*,1	A	Valid	Valid	Signed divide* *r/m16* by 2, once.
D3 /7	SAR *r/m16*, CL	B	Valid	Valid	Signed divide* *r/m16* by 2, CL times.
C1 /7 *ib*	SAR *r/m16, imm8*	C	Valid	Valid	Signed divide* *r/m16* by 2, *imm8* times.
D1 /7	SAR *r/m32*, 1	A	Valid	Valid	Signed divide* *r/m32* by 2, once.
REX.W + D1 /7	SAR *r/m64*, 1	A	Valid	N.E.	Signed divide* *r/m64* by 2, once.
D3 /7	SAR *r/m32*, CL	B	Valid	Valid	Signed divide* *r/m32* by 2, CL times.
REX.W + D3 /7	SAR *r/m64*, CL	B	Valid	N.E.	Signed divide* *r/m64* by 2, CL times.
C1 /7 *ib*	SAR *r/m32, imm8*	C	Valid	Valid	Signed divide* *r/m32* by 2, *imm8* times.
REX.W + C1 /7 *ib*	SAR *r/m64, imm8*	C	Valid	N.E.	Signed divide* *r/m64* by 2, *imm8* times
D0 /4	SHL *r/m8*, 1	A	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SHL *r/m8***, 1	A	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SHL *r/m8*, CL	B	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SHL *r/m8***, CL	B	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SHL *r/m8, imm8*	C	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SHL *r/m8***, *imm8*	C	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SHL *r/m16*,1	A	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SHL *r/m16*, CL	B	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SHL *r/m16, imm8*	C	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SHL *r/m32*,1	A	Valid	Valid	Multiply *r/m32* by 2, once.
REX.W + D1 /4	SHL *r/m64*,1	A	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SHL *r/m32*, CL	B	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SHL *r/m64*, CL	B	Valid	N.E.	Multiply *r/m64* by 2, CL times.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C1 /4 *ib*	SHL *r/m32, imm8*	C	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SHL *r/m64, imm8*	C	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /5	SHR *r/m8*,1	A	Valid	Valid	Unsigned divide *r/m8* by 2, once.
REX + D0 /5	SHR *r/m8***, 1	A	Valid	N.E.	Unsigned divide *r/m8* by 2, once.
D2 /5	SHR *r/m8*, CL	B	Valid	Valid	Unsigned divide *r/m8* by 2, CL times.
REX + D2 /5	SHR *r/m8***, CL	B	Valid	N.E.	Unsigned divide *r/m8* by 2, CL times.
C0 /5 *ib*	SHR *r/m8, imm8*	C	Valid	Valid	Unsigned divide *r/m8* by 2, *imm8* times.
REX + C0 /5 *ib*	SHR *r/m8***, *imm8*	C	Valid	N.E.	Unsigned divide *r/m8* by 2, *imm8* times.
D1 /5	SHR *r/m16*, 1	A	Valid	Valid	Unsigned divide *r/m16* by 2, once.
D3 /5	SHR *r/m16*, CL	B	Valid	Valid	Unsigned divide *r/m16* by 2, CL times
C1 /5 *ib*	SHR *r/m16, imm8*	C	Valid	Valid	Unsigned divide *r/m16* by 2, *imm8* times.
D1 /5	SHR *r/m32*, 1	A	Valid	Valid	Unsigned divide *r/m32* by 2, once.
REX.W + D1 /5	SHR *r/m64*, 1	A	Valid	N.E.	Unsigned divide *r/m64* by 2, once.
D3 /5	SHR *r/m32*, CL	B	Valid	Valid	Unsigned divide *r/m32* by 2, CL times.
REX.W + D3 /5	SHR *r/m64*, CL	B	Valid	N.E.	Unsigned divide *r/m64* by 2, CL times.
C1 /5 *ib*	SHR *r/m32, imm8*	C	Valid	Valid	Unsigned divide *r/m32* by 2, *imm8* times.
REX.W + C1 /5 *ib*	SHR *r/m64, imm8*	C	Valid	N.E.	Unsigned divide *r/m64* by 2, *imm8* times.

NOTES:

* Not the same form of division as IDIV; rounding is toward negative infinity.

** In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

***See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (r, w)	1	NA	NA
B	ModRM:r/m (r, w)	CL (r)	NA	NA
C	ModRM:r/m (r, w)	imm8	NA	NA

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction's default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8–R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
IF 64-Bit Mode and using REX.W
    THEN
        countMASK ← 3FH;
    ELSE
        countMASK ← 1FH;
FI

tempCOUNT ← (COUNT AND countMASK);
tempDEST ← DEST;
WHILE (tempCOUNT ≠ 0)
DO
    IF instruction is SAL or SHL
        THEN
            CF ← MSB(DEST);
```



```
        ELSE (* Instruction is SAR or SHR *)
            CF ← LSB(DEST);
    FI;
    IF instruction is SAL or SHL
        THEN
            DEST ← DEST * 2;
        ELSE
            IF instruction is SAR
                THEN
                    DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
                ELSE (* Instruction is SHR *)
                    DEST ← DEST / 2 ; (* Unsigned divide *)
            FI;
    FI;
    tempCOUNT ← tempCOUNT - 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
        FI;
    ELSE IF (COUNT AND countMASK) = 0
        THEN
            All flags unchanged;
        ELSE (* COUNT not 1 or 0 *)
            OF ← undefined;
    FI;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit

shifts (see "Description" above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

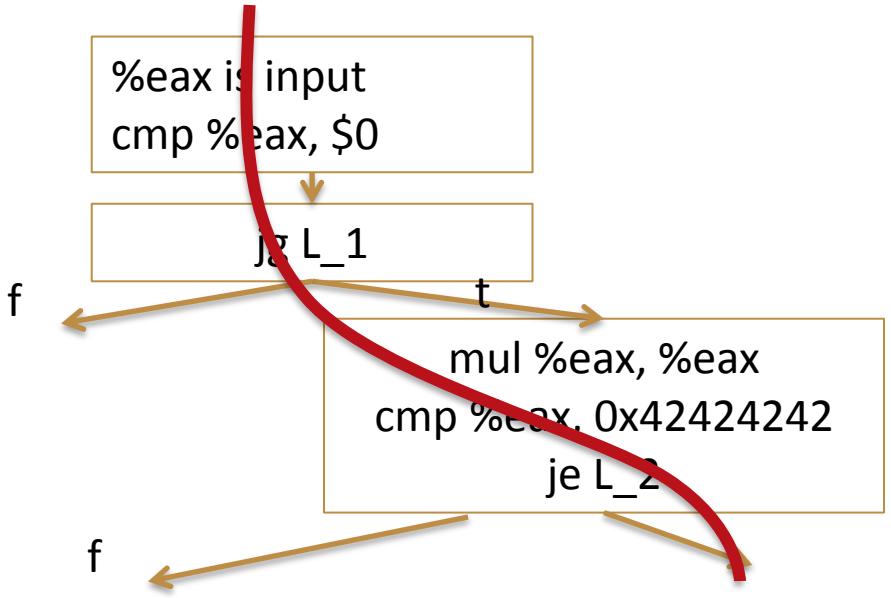
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

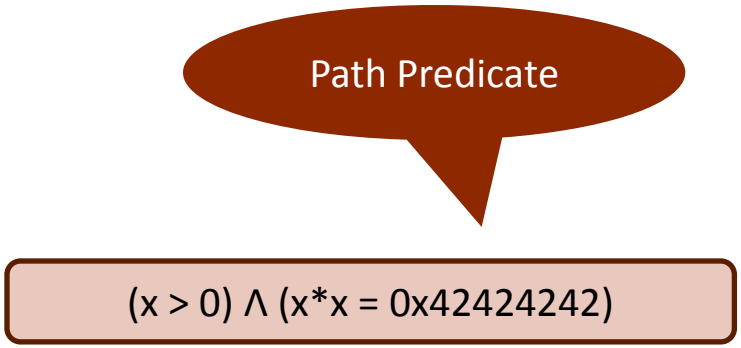
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

(taken from Intel Manual)

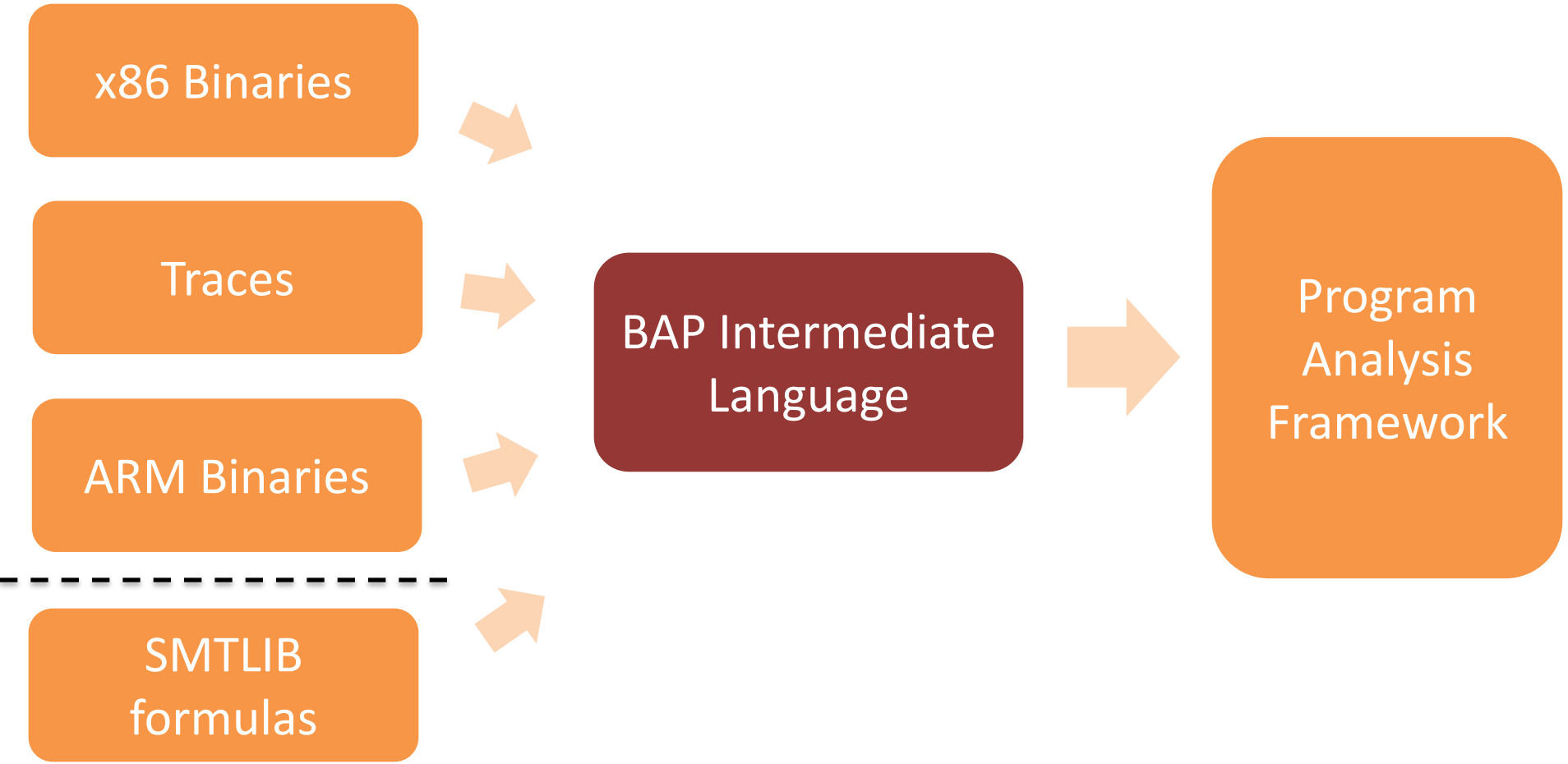
Dynamic Symbolic Execution on Binaries [Schwartz et al, USENIX'11]

- Finding inputs that explore different paths after executing 100,000s of assembly instructions

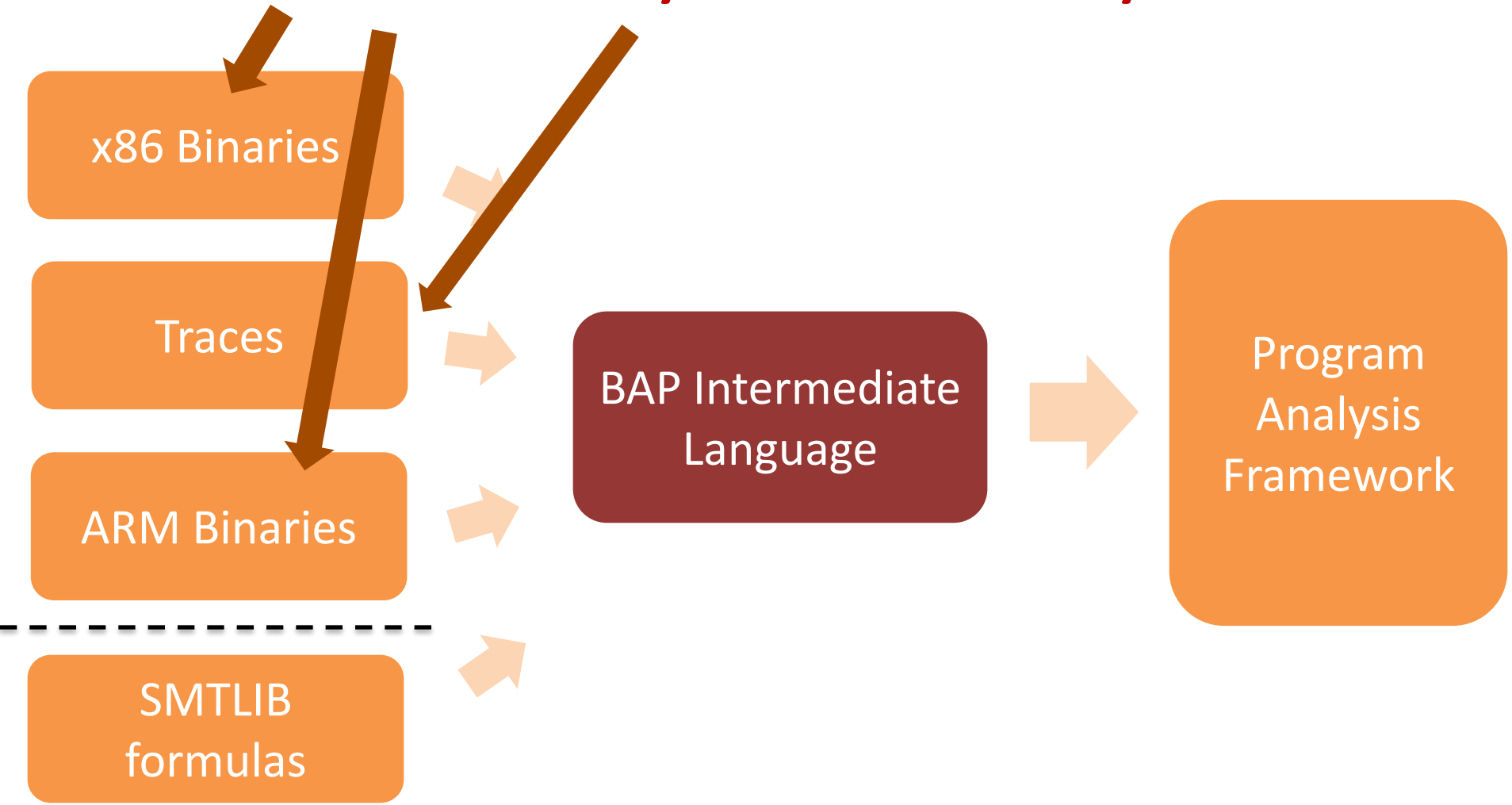




Compiler-like Design



Static & Dynamic Analysis



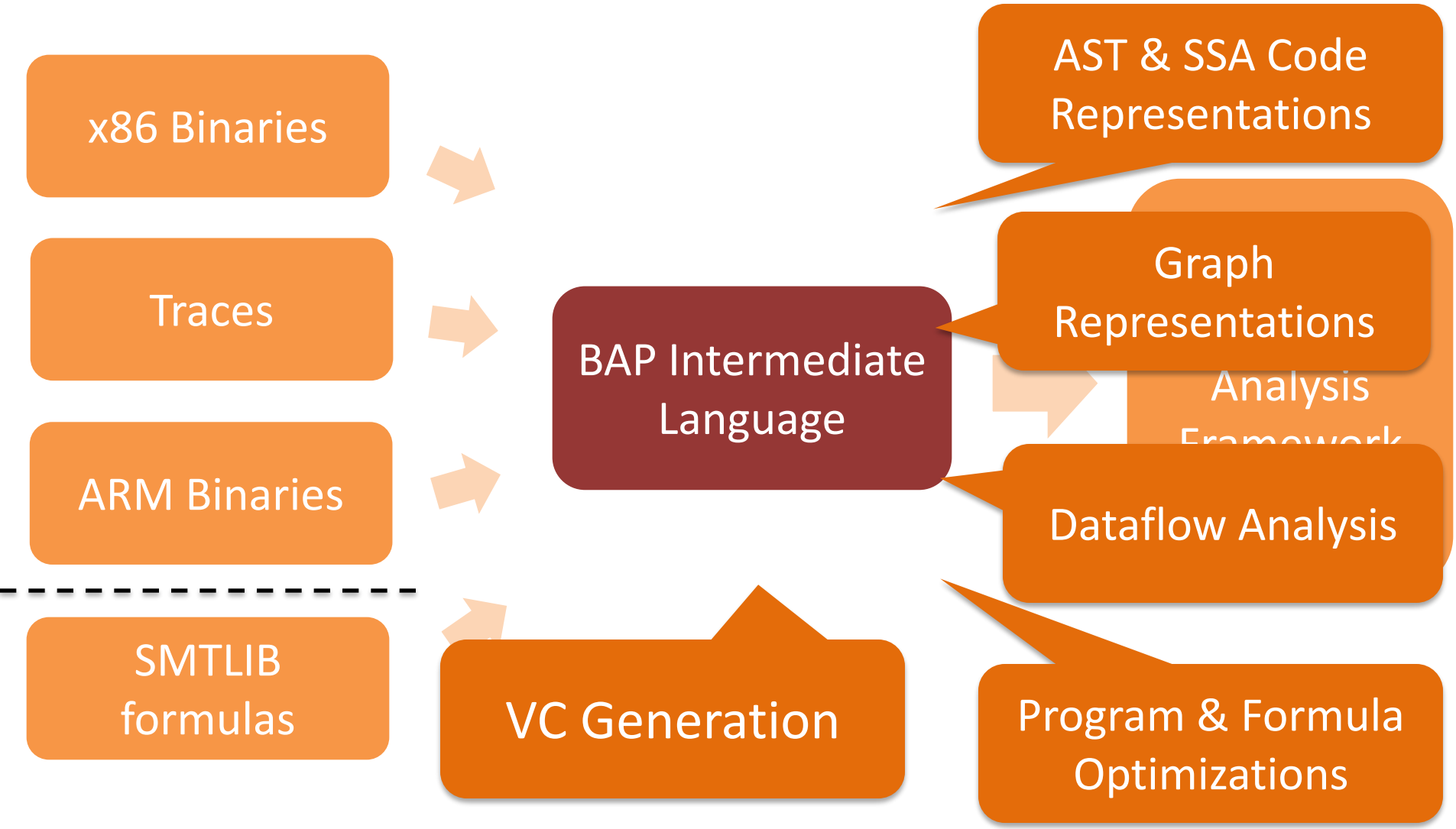
A Simple Intermediate Language

- Consists of 17 language constructs
  - 7 statements and 10 expressions

```
program ::= stmt*
stmt    ::= var := exp | jmp exp | cjmp exp,exp,exp | assert exp
          | label label_kind | addr address | special string
```

- Our binary symbolic executor consists of ~250 lines of OCaml code

Extensible Program Analysis



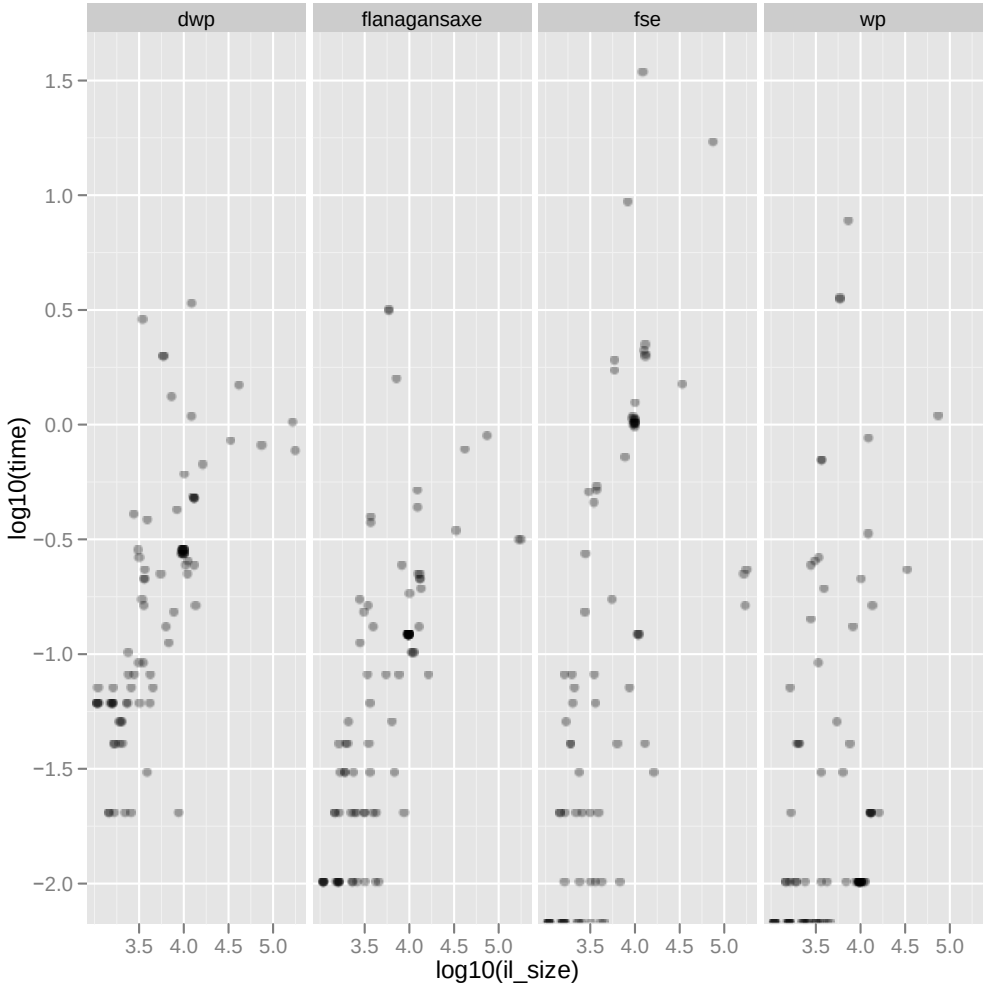
Verification Condition Generation

- BAP provides support for the following VC generation algorithms:
  - Dijkstra's WP
  - Flanagan & Saxe's WP
  - Directionless WP
  - Forward Symbolic Execution
- Interfaces to SMT solvers
  - Support for SMTLIB1 & SMTLIB2 formats

Static Checking of Safety Properties

[Jager et al, TR'10]

- GNU coreutils leaf functions
  - Integer overflows
  - Memory overwrites
- Formula optimizations improved performance up to 8x

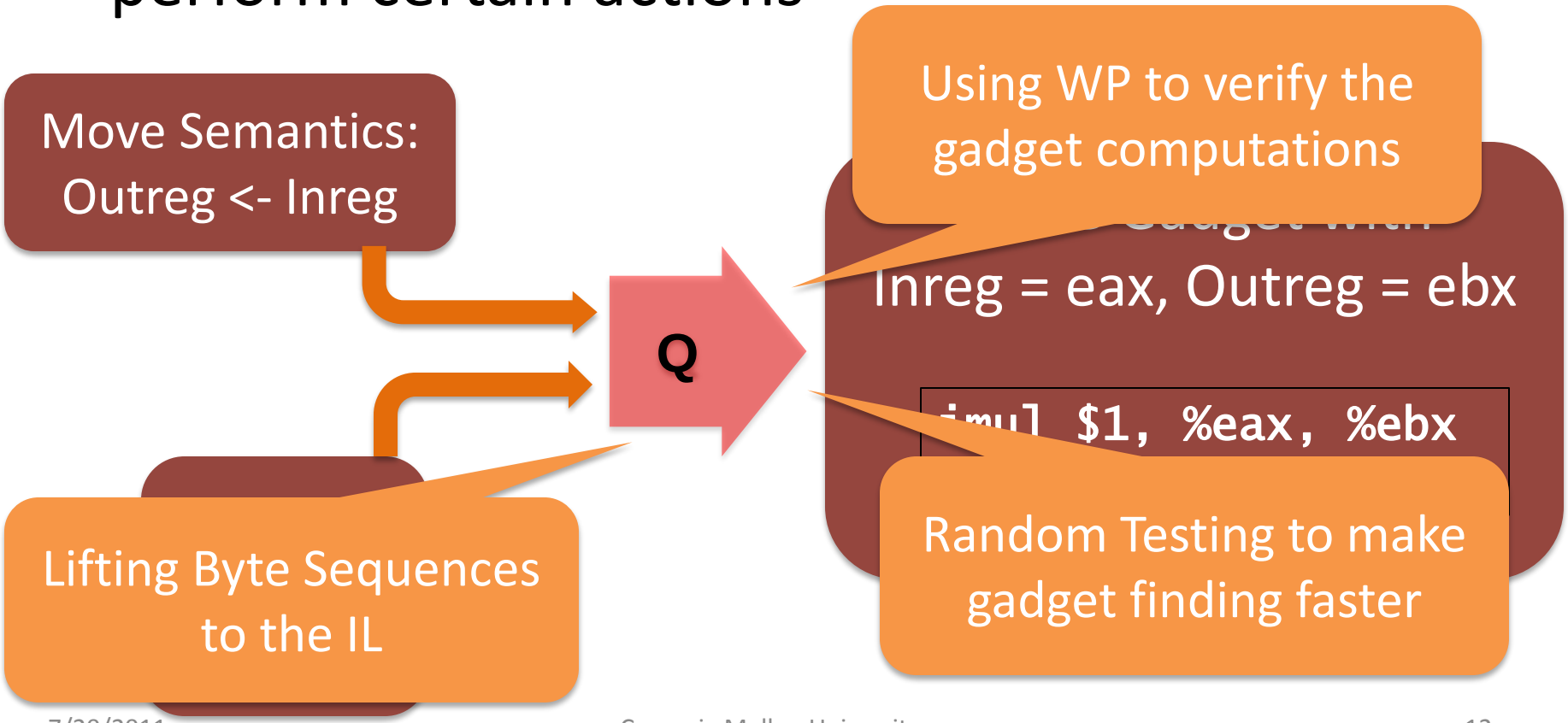


BAP in research

Q: Return-Oriented Programming

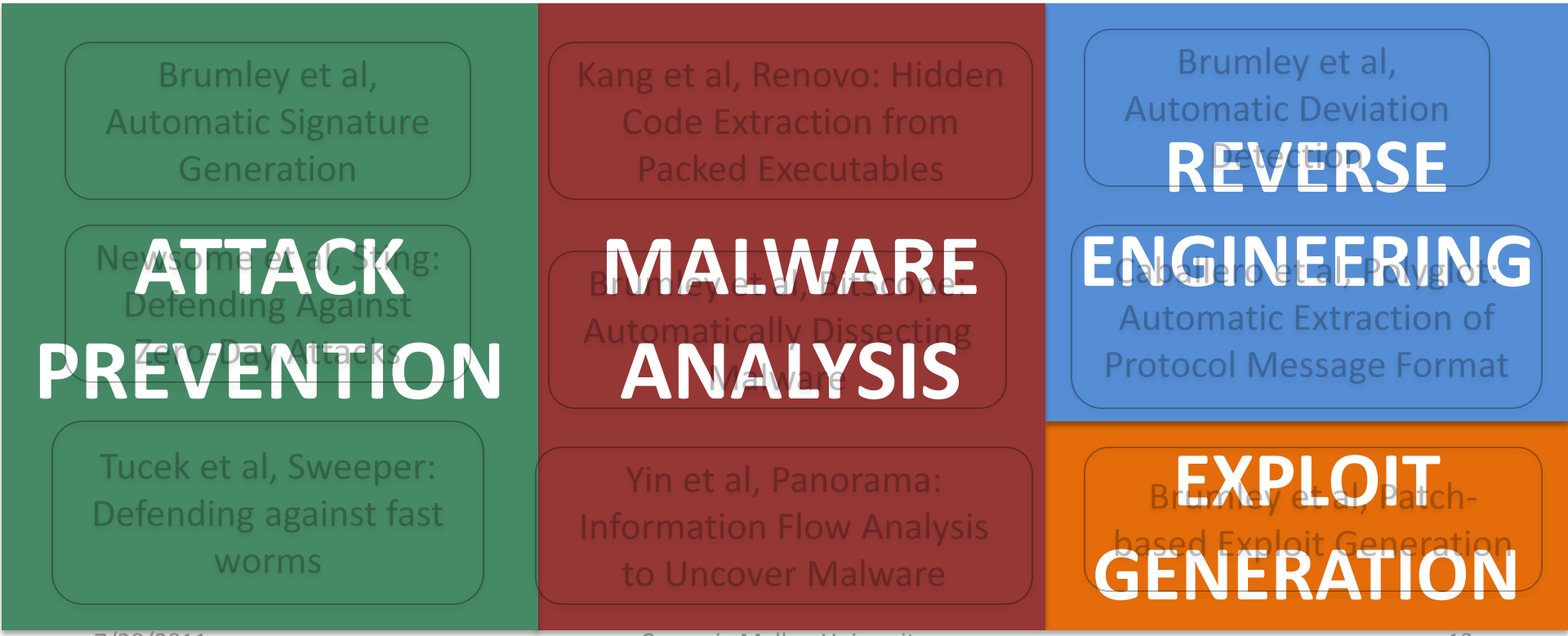
[Schwartz et al, USENIX'11]

- Finding byte-sequences (gadgets) that perform certain actions

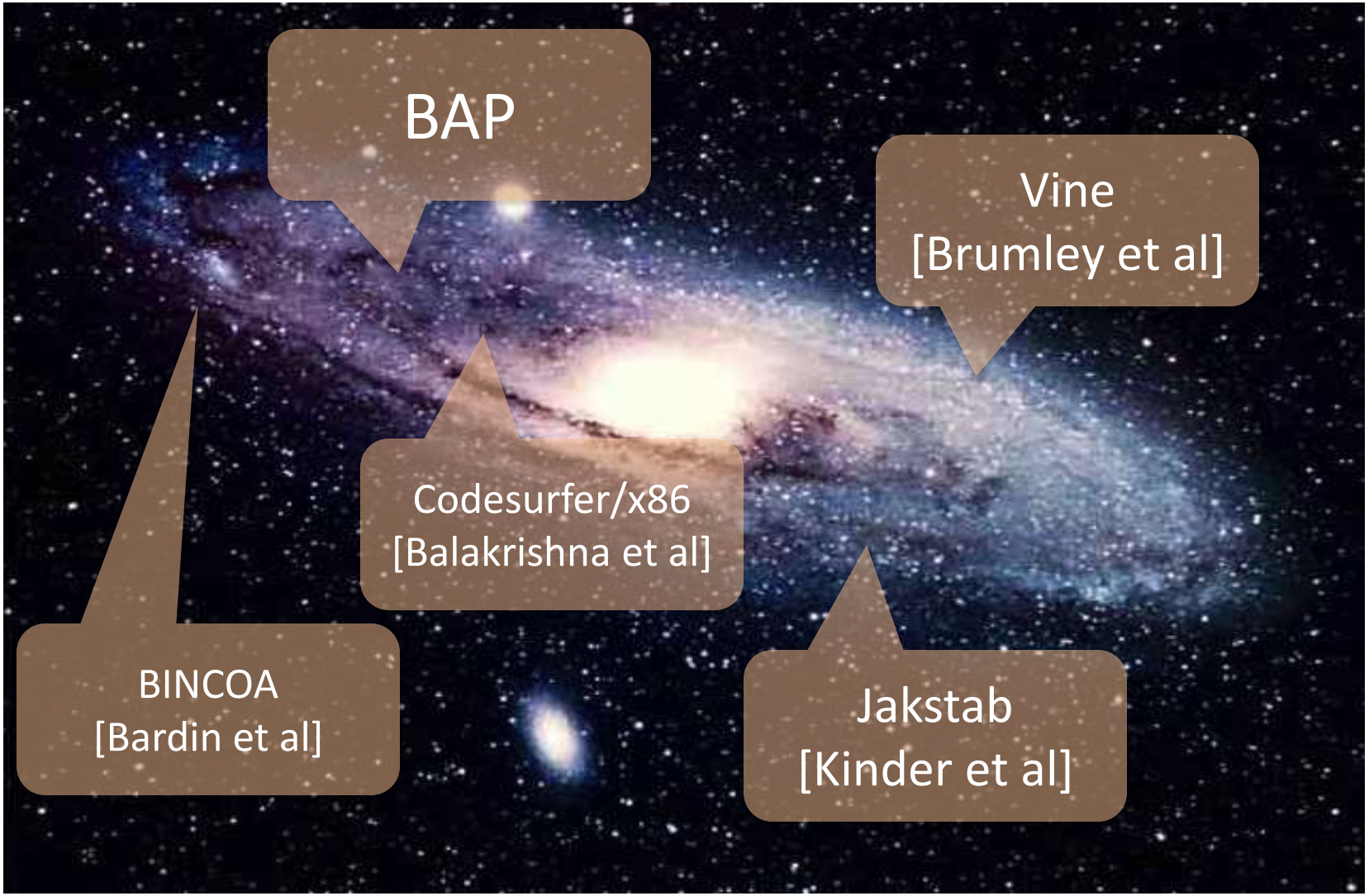


Many Applications in Security

Don’t redo the engineering. Do the science.



Are we alone?


BAP
Vine
[Brumley et al]
Codesurfer/x86
[Balakrishna et al]
BINCOA
[Bardin et al]
Jakstab
[Kinder et al]

Conclusion

- BAP is the newest incarnation of our framework for binary analysis
- BAP comes with a variety of algorithms and features to make analysis easier
- You can download it for free at:

http://bap.ece.cmu.edu/

BAP 0.3 just came out!

Thank you!

thanassis@cmu.edu
http://www.ece.cmu.edu/~aavgerin
http://bap.ece.cmu.edu/

Questions?