# Special Topics in Logic and Security 1
## Domain Interaction

Paul Irofti

Master Year II, Sem. I, 2023-2024

# Memory Access

What happens with the *Pts* domain in the program below?

```
int A[4][8] = {...};
uint i, j;
uint sum = 0;

for (i = 0; i < 4; i++)
    for (j = 0; j < 8; j++)
        sum += A[i][j];

printf("sum = %d\n", sum);
```

# Memory Access

What happens with the *Pts* domain in the program below?

```
int A[4][8] = {...};
uint i, j;
uint sum = 0;

for (i = 0; i < 4; i++)
    for (j = 0; j < 8; j++)
        sum += A + 8*sizeof(*A)*i + sizeof(*A)*j;

printf("sum = %d\n", sum);
```

## Memory Access

What happens with the *Pts* domain in the program below?

```
for (i = 0; i < 4; i++)
    for (j = 0; j < 8; j++)
        sum += A + 8*sizeof(*A)*i + sizeof(*A)*j;
```

On an architecture where `int` has 64-bits:

```
for (i = 0; i < 4; i++)
    for (j = 0; j < 8; j++)
        sum += A + 8*8*i + 8*j
```

If A starts at *byte* 0, it leads to the following memory access sequence:

$$\underbrace{0, 8, 16, 24, \ldots, 56}_{\texttt{A[0]}} \quad \underbrace{64, 72, \ldots, 120}_{\texttt{A[1]}} \quad \underbrace{128, 136, \ldots, 184}_{\texttt{A[2]}} \quad \underbrace{192, 200, \ldots, 248}_{\texttt{A[3]}}$$

# Memory Access

$$\underbrace{0, 8, 16, 24, \ldots, 56}_{\texttt{A[0]}} \ \underbrace{64, 72, \ldots, 120}_{\texttt{A[1]}} \ \underbrace{128, 136, \ldots, 184}_{\texttt{A[2]}} \ \underbrace{192, 200, \ldots, 248}_{\texttt{A[3]}}$$

We can rewrite

```
for (i = 0; i < 4; i++)
    for (j = 0; j < 8; j++)
        sum += A + 64*i + 8*j
```

in the *Lin* domain where *a* and *x* are:

$$\boldsymbol{a} = [64\ 8], \quad \boldsymbol{x} = [i\ j]$$

which leads to two half-spaces in the *Ineq* domain:

$$0 \leq i \leq 3, \quad 0 \leq j \leq 7 \quad, [\![64i \leq 192]\!], \quad [\![8j \leq 56]\!]$$

that define a convex surface in the polyhedra domain *Poly*.

But what can we say about the *Pts* domain, what can we say about the values taken by <u>pointers</u> accessing array A?

# Aligned versus Unaligned Access

The way we declare the array

```
int A[4][8];
A[i][j] == *(A + 8*sizeof(*A)*i + sizeof(*A)*j)
```

offers information regarding the way memory access is intended to happen: 32 integers laid out in 4 contiguous memory blocks.
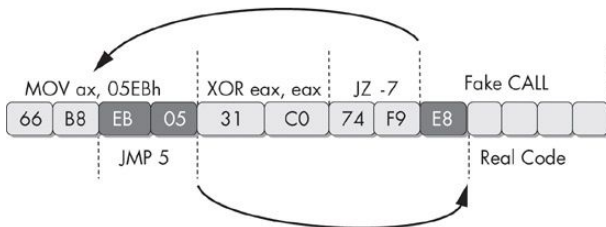
What happens when we write:

```
u = *((char *)A + 2);
v = *((char *)A + 111);
t = *((char *)A + 254);
```

what are the values of u, v and t?

Some architectures constrain memory access to be aligned in hardware.

Intel, AMD and ARM allow unaligned access. Thus, the entire range $[0, 255]$ has to be taken into consideration, not just the set $\{0, 8, 16, \dots, 248\}$.

# Example: Unaligned Access



https://stackoverflow.com/questions/30192694/
jump-to-the-middle-of-an-instruction

# Polyhedra Connection

Coming back to the remark: "Thus, the entire range $[0, 255]$ has to be taken into consideration, not just the set $\{0, 8, 16, \ldots, 248\}$."

In fact the Polyhedra domain is given us exactly this information: half-planes include all byte ranges, not just the type-induced multiples.

## Example

In the *Pts* domain of `s = strings[i]` we showed that `s` can only point to $\{s_1, s_2, s_3\}$ because `i` can only take the values $\{0, 1, 2\}$ inside the loop.

But in fact the *Poly* half-plane is actually $[\![8i \leq 16]\!]$ and we have no analysis to show that other bytes inside `strings` can not be accessed.

We now need to equip *Poly* to check if pointers access intended memory addresses (or not) within the arrays.

# Granger Domain

The *Granger* domain is used for variables inference and analysis that take the form $c + m\mathbb{Z}$

$$\{\ldots, -2m + c, -m + c, c, c + m, c + 2m, \ldots\}$$

the domain is abstract enough to permit constructions of the type

```
int i = 5;
struct {
    int a;
    char v[100];
    short f;
} a[10];
a[i].f = 0;
*((char *)a + 13) = 2;
```

**Exercise:** Show how we can model the two references.

# Multiplicity Domain

## Proposition

*For the analysis of simple vectors and arrays, that use basic types (e.g. `int`, `char`), it suffices to show that a variable is a multiple of $2^n$ in order to prove that the memory access is aligned.*

**Exercise:** Prove the proposition on the example that was using array `A[4][8]`.

## Definition

Let $Mult = \mathcal{X} \to \{0, \ldots, 64\}$ be the functional space that records the number of least significant bits (LSB) that are always zero, null.

## Remark

*The linear transform $M \in Mult$ assigns a value $n = M(x)$ to all the variables $x \in \mathcal{X}$.*

# Example: Multiplicity Domain

**Remark**

*The linear transform $M \in Mult$ assigns a value $n = M(x)$ to all the variables $x \in \mathcal{X}$.*

Suppose that all the variables are represented on at most 64-bits.

| Type | Var. | Max Mult. | |
|---:|:---:|:---|---:|
| uint | x | $M(x) =$ | 63 |
| uint32_t | y | $M(y) =$ | 31 |
| ushort | f | $M(f) =$ | 15 |
| uchar | c | $M(c) =$ | 7 |

Thus, no matter the type, x=0 can be represented as $M(x) = 64$.

# The Lattice $(Mult, \subseteq_M, \vee_M, \wedge_M)$

Let $M, M', M_1, M_2 \in Mult$.

**Update:** $M \to M' = M[x \to n'] \implies M'(x) = n'$ and $M'(y) = M(y), \forall y \neq x$.

**Join:** $M' = M_1 \vee_M M_2$ s.t. $M'(x) = \min(M_1(x), M_2(x)), \forall x \in \mathcal{X}$.

**Inclusion:** $M_1 \subseteq_M M_2 \iff M_1(x) \geq M_2(x), \forall x \in \mathcal{X}$.

**Exercise:** Find the $\top$ element: the largest element from the lattice. Explain.

Let $Equ = Lin \times \mathbb{Z}$ be the set of linear equations of the type $e = c$, where $e \in Lin, c \in \mathbb{Z}$.

**Meet:** The intersection operator adds the information provided by a new equation: $M' = M \wedge_M (e = c)$.

### Definition

$\wedge_M : Mult \times Equ \to (Mult \cup \{\perp_M\})$, where $\perp_M$ denotes an unsatisfiable (impossible) state.

# The $\wedge_M$ Operation

**Definition**

Let $\delta : \mathbb{Z} \to \{0, \ldots, 64\}$ s.t. $\delta(c)$ represents the number of unused (zero) LSB from $c$.

Let $e \equiv a_1 x_1 + \cdots + a_n x_n$ s.t. $a_i \neq 0, \forall i = 1, \ldots, n$. We recompute the multilpicity of variable $x_j$ by rewriting $e = c$

$$-a_j x_j = a_1 x_1 + \cdots + a_{j-1} x_{j-1} + a_{j+1} x_{j+1} + \cdots + a_n x_n - c$$

**Remark**

*The multiplicity of $a_i x_i$ is $\delta(a_i) + M(x_i)$, and the multiplicity of $c$ is simply $\delta(c)$.*

**Intuition:** *Mult* is similar to the exponent operations: $2^m 2^n = 2^{m+n}$.

**Proposition**

*The operation $\wedge_M$ adds information, thus the number of null LSB from $x_j$ can not decrease. On the contrary, this number can increase due to the presence of $c$.*

# Number of null <u>LSB</u>'s from $x_j$ cannot decrease

$$-a_j x_j = a_1 x_1 + \cdots + a_{j-1} x_{j-1} + a_{j+1} x_{j+1} + \cdots + a_n x_n - c$$

The equation's right-hand-side multiplicity has to be greater than or equal to the one of every individual term:

$$\min(\delta(c), \min_{i, i \neq j} \delta(a_i) + M(x_i))$$

Example (`A[i][j]=*(A + 8*8*i + 8*j)`)

`64*i + 8*j` $\implies \min(\delta(64) + M(i), \delta(8) + M(j)) = \min(6 + M(i), 3 + M(j))$.
Let $i = 2, j = 4$, then $\min(6 + 1, 3 + 2) = 5$, and $64i + 8j = 160 = 1010\ 0000_2$.

If $a_j > 1 \vee a_j < -1$, then the number in the equation above has to be reduced by $\delta(a_j)$ in order to obtain the new $M' = M(x_j \to n')$

$$M' = M\left[ x_j \to \max\left( M(x_j), \min(\delta(c), \min_{i, i \neq j} \delta(a_i) + M(x_i)) - \delta(a_j) \right) \right]$$

# Example: Updating the Multiplicity

Let $M$ be the initial multipliers state of the three variables $x, y, z$, where $x$ is a multiple of 8, and $y$ and $z$ are multiples of 2.

We add the equation $x + y + 2z = 0 \in Equ$ that resulted from a <u>pointer</u> arithmetic problem in the domain of $M$.

$$M' = M \wedge_M \{x + y + 2z = 0\}$$

which we solve by iteratively updating the muliplicity of each variable.

|        | $M(x)$ | $M(y)$ | $\delta(2) + M(z)$ | $\delta(0)$ |
|--------|--------|--------|--------------------|-------------|
| $M$    | 3      | 1      | $1 + 1$            | 64          |
| $M'(x)$| 3      | 1      | $1 + 1$            | 64          |
| $M'(y)$| 3      | 2      | $1 + 1$            | 64          |
| $M'(z)$| 3      | 2      | $1 + 1$            | 64          |

**Exercise:** What happens if we add the equation $x + y + 2z = 1$?

# Properties

**Proposition**

*The $\wedge_M$ operation leads to an invalid state $\bot_M$ if*

$$\min_{i=1,\dots,n} \delta(a_i) + M(x_i) > \delta(c)$$

**Complexity:** The update of a single variable has to take into account all the variables: quadratic cost. In practice we have to deal with at most 2–3 variables.

**Remark**

*The special operations from Poly are solved similarly:*

- $M' = M \triangleright x := e \implies M(x) = M(t) = 0$ *due to the projection operator*
- *updating terms other than $M(x), M(t)$ from e does not bring new information*
- $M \triangleright x := y \gg n \implies M(x)$ *is at least $(M(y) - n)$, $M(y)$ does not change*

# Properties

**Alignment:** We can verify if the access is aligned through the operation $M \wedge_M \{x = 2^n\}$. If the result is $\perp_M$ then we have an illegal access error.

**Example:** Let $x$ denote the offset of pointer $s$ inside `strings` and let the pointer type be 8 bytes long. Then $M' \wedge_M \{s = 8\}$ ensures that $M'(s) \geq 3$ or $M' = \perp_M$. If $M \subseteq_M M'$, then we have aligned access. Otherwise emit warning.

**Projection:** Let the function $\exists_x : Mult \to Mult$ and let $M' = \exists_x(M)$. Then $M'(x) = 0$ and $M'(y) = M(y), \forall y \neq x$.

### Theorem

$(Mult, \subseteq_M, \wedge_M, \vee_M)$ *forms a complete lattice.*

**Exercise:** Prove the theorem.

# Proof: $(Mult, \subseteq_M, \wedge_M, \vee_M)$ forms a lattice

Let $M, M_1, M_2, M_3 \in Mult$

$(Mult, \subseteq_M)$ POSET:

- reflexive: $M \subseteq_M M \iff M(x) \geq M(x) \quad \forall M$
- anti-symmetric:
  $M_1 \subseteq_M M_2, M_2 \subseteq_M M_1 \implies M_1(x) \geq M_2(x), M_2(x) \geq M_1(x)$
  $\implies M_1 = M_2$
- transitivity: $M_1 \subseteq_M M_2, M_2 \subseteq_M M_3 \implies M_1 \subseteq_M M_3$ ;
  $M_1(x) \geq M_2(x)$ and $M_2(x) \geq M_3(x) \implies M_1(x) \geq M_3(x) \implies M_1 \subseteq_M M_3$

Lattice:

- associative:
  $(M_1 \vee_M M_2) \vee_M M_3 = M_1 \vee_M (M_2 \vee_M M_3)$
  $\impliedby \min(a, min(b, c)) = \min(\min(a, b), c);$
  $(M_1 \wedge_M M_2) \wedge_M M_3 = M_1 \wedge_M (M_2 \wedge_M M_3) \impliedby$ update can not decrease
  LSB and order does not affect the update operation
- commute: $\quad M_1 \vee_M M_2 = M_2 \vee_M M_1 \impliedby \min(a, b) = \min(b, a);$
  $\qquad\qquad M_1 \wedge_M M_2 = M_2 \wedge_M M_1 \impliedby$ update can not decrease LSB
- absorb: $\qquad M_1 \wedge_M (M_1 \vee_M M_2) = M_1$ ; $\qquad\qquad M_1 \vee_M (M_1 \wedge_M M_2) = M_1$

# *Poly* and *Mult* Interaction

Let $Num = (Poly \times Mult) \cup \{\perp_N\}$, where $\perp_N$ represents an <u>unreachable</u> state, that is impossible to attain, in the program definition. We define:

- $(P, M) \subseteq_N (P', M') \iff (P \subseteq_P P') \wedge (M \subseteq_M M')$
- $(P', M') = (P_1, M_1) \vee_N (P_2, M_2) \iff (P' = P_1 \vee_P P_2) \wedge (M' = M_1 \vee_M M_2)$
- $(P', M') = (P, M) \triangleright x := e \iff (P' = P \triangleright x := e) \wedge (M' = M \triangleright x := e)$
- $(P', M') = (P, M) \triangleright x := e \gg n \iff (P' = P \triangleright x := e \gg n) \wedge (M' = M \triangleright x := e \gg n)$
- $(P', M') = \exists_x(P, M) \iff (P' = \exists_x(P)) \wedge (M' = \exists_x(M))$
- $(P, M) \wedge_N \{e = c\} = \begin{cases} \perp_N & \text{if } P' = \emptyset \text{ or } M' = \perp_M \\ (P', M') & \text{otherwise} \end{cases}$ , where

  $P' = P \wedge_P [\![\{e = c\}]\!]$ and $M' = M \wedge_M \{e = c\}$.