Putting Numerical Abstract Domains to Work: A Study of Array-Bound Checking for C Programs

Arnaud J. Venet
Carnegie Mellon University
NASA Ames Research Center
arnaud.j.venet@nasa.gov

This work was partially supported by the NSF Grant #1136008 and the SSAT project under the Aviation Safety program of NASA’S Aeronautics Research Mission Directorate.
Abstract Interpretation

• A theory of sound semantic approximation introduced by Patrick & Radhia Cousot in the mid 70’s
• First application to the computation of variable ranges (1976)
• Verification of the numerical algorithms in the A380 flight software (2005)
• Numerical abstract interpretation is an active field of research
Roadmap

• The domain of convex polyhedra

• Application to array-bound checking:
  – The buffer library of OpenSSH (700 LOC)
  – The flight software of Mars Exploration Rovers (550 KLOC)

• Improving scalability: the gauge domain
The domain of convex polyhedra
A simple example

for(i = 0; i < 10; i++) {
    if(message[i].kind == SHORT_DATA)
        allocate_space (channel, 1000);
    else
        allocate_space (channel, 2000);
}

What are the memory requirements?
Control flow graph

```c
i = 0
if (i < 10) {
    i++
    if (message[i].kind == SHORT_CMD) {
        allocate_space(channel, 2000);
        allocate_space(channel, 1000);
    }
}
stop
```
Abstract model of the code

\[
\begin{align*}
i &= 0 \\
i < 10 \quad &\text{?} \\
i &= i + 1 \\
\text{message}[i].\text{kind} &= \text{SHORT_CMD} \quad &\text{?} \\
\text{allocate_space} &= (\text{channel}, 2000) \\
\text{allocate_space} &= (\text{channel}, 1000) \\
\text{stop} &
\end{align*}
\]

\[
\begin{align*}
M &= M + 2000 \\
M &= M + 1000 \\
M &= M + 0
\end{align*}
\]
Analyzing the model

\[ i = 0 \]

\[ i < 10 \]?

\[ M = 0 \]

\[ M = M + 1000 \]

\[ M = M + 2000 \]

\[ i++ \]
Initially

\[ M = 0 \]

\[ i = 0 \]

\[ i < 10 ? \]

\[ M = M + 1000 \]

\[ M = M + 2000 \]

\[ i++ \]

stop

\[ M \]

\[ i \]
Loop initialization

\[ M = 0 \]

\[ i = 0 \]

\[ i < 10 \]?

\[ M = M + 1000 \]

\[ M = M + 2000 \]

\[ i++ \]

stop
Loop entry

```
M = 0

i = 0

i < 10 ?

M = M + 1000

M = M + 2000

i++
```

<table>
<thead>
<tr>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Analyzing a branching (1)

```
M = 0

i = 0

i < 10 ?

M = M + 1000

M = M + 2000

i++

stop
```
Analyzing a branching (2)

\[ i = 0 \]

\[ i < 10 \, ? \]

\[ M = 0 \]

\[ M = M + 1000 \]

\[ M = M + 2000 \]

\[ i++ \]
Accumulating all possible values

\[ M = 0 \]
\[ i = 0 \]
\[ i < 10 \ ? \]

\[ M = M + 1000 \]
\[ M = M + 2000 \]
\[ i++ \]

\[ i \]
\[ M \]
Abstraction of point clouds

- We want the analysis to terminate in reasonable time
- We need a tractable representation of point clouds in arbitrary dimensions
- Convex polyhedra (Cousot & Halbwachs, 1978)
- Compute the convex hull of a point cloud
Analyzing a branching

```
i = 0
M = 0

M = M + 1000
i < 10 ?
i = 0

M = M + 2000

i++
```

Graph:

- Start at $M = 0$
- $i = 0$
- $M = M + 1000$
- Check $i < 10$
- If true, go to $M = M + 2000$
- $i = i + 1$
- Check again
- If $i < 10$, repeat; otherwise, stop.
Convex hull

```
M = 0

i = 0

i < 10 ?

M = M + 1000

M = M + 2000

i++

stop
```
Iterating the loop analysis

i = 0

M = M + 1000

i < 10 ?

M = M + 2000

M = 0

i++

stop
Building the loop invariant

\[ i = 0 \]

\[ i < 10 \]

\[ M = M + 1000 \]

\[ M = M + 2000 \]

\[ M = 0 \]

\[ i++ \]

\[ \text{stop} \]
Analyzing a branching

```
i = 0
M = M + 2000
M = M + 1000
M = 0
```

```java
i++
```
Analyzing a branching

\[
\begin{align*}
M &= 0 \\
i &= 0 \\
i < 10 \quad ? \\
M &= M + 1000 \\
M &= M + 2000 \\
i++ \\
\end{align*}
\]
Convex hull

\[
\begin{align*}
M &= 0 \\
i &= 0 \\
i &< 10 \ ? \\
M &= M + 1000 \\
M &= M + 2000 \\
i &++ \\
\text{stop}
\end{align*}
\]
Building the loop invariant

i = 0

i < 10 ?

M = 0

M = M + 1000

M = M + 2000

i++

stop
Keep iterating...

```plaintext
M = 0
i = 0

while (i < 10):
    M = M + 1000
    M = M + 2000
    i++

stop
```
Passing to the limit

- We want the analysis to terminate when analyzing loops
- After a few iteration steps, we use a **widening** operation at loop entry to enforce convergence
Widening $\nabla$

- Let $a_1, a_2, \ldots a_n, \ldots$ be a sequence of polyhedra, then the sequence
  - $w_1 = a_1$
  - $w_{n+1} = w_n \nabla a_{n+1}$

is ultimately stationary

- The widening is a *join* operation:
  $$a \subseteq a \nabla b \ \& \ b \subseteq a \nabla b$$
Widening for intervals

• $[a, b] \triangledown [c, d] =$

  $$[\text{if } c < a \text{ then } -\infty \text{ else } a, \text{ if } b < d \text{ then } +\infty \text{ else } b]$$

• Example:

  $$[10, 20] \triangledown [11, 30] = [10, +\infty]$$
Widening for polyhedra

• We eliminate the faces of the computed convex envelope that are not stable

• Convergence is reached in at most $N$ steps where $N$ is the number of faces of the polyhedron at loop entry
Widening

\[ i = 0 \]
\[ i < 10 \ ? \]
\[ M = M + 1000 \]
\[ M = M + 2000 \]
\[ i++ \]

\[ \text{stop} \]

Graphs:
- \( M \) vs. \( i \)
- \( M \) vs. \( i \) for different cases
After the widening

\[ i = 0 \]

\[ i < 10 ? \]

\[ M = 0 \]

\[ i++ \]

\[ M = M + 1000 \]

\[ M = M + 2000 \]
Detecting convergence

• Abstract iteration sequence
  – \( F_1 = P \) (initial polyhedron)
  – \( F_{n+1} = F_n \) if \( S(F_n) \subseteq F_n \)
    \[ F_n \bigtriangledown S(F_n) \] otherwise
  \( S \) is the semantic transformer associated to the loop body

• **Theorem:** if there exists \( N \) such that \( F_{N+1} \subseteq F_N \), then \( F_n = F_N \) for \( n > N \).
Convergence

The computation has converged
We are not done yet...

• The analyzer has just proven that
  \[ 1000 \times i \leq M \leq 2000 \times i \]

• But we have lost all information about the termination condition \( 0 \leq i \leq 10 \)

• Since we have obtained a superset of all possible values of the variables, if we run the computation again we still get a superset

• This new envelope may be smaller

• This refinement step is called *narrowing*
Refinement

```
i = 0
M = M + \begin{cases} 2000 & i < 10 \\ 1000 & \text{stop} \end{cases}
i = i + 1
```

Diagram:
- Initial state: $M = 0$, $i = 0$
- Transition: if $i < 10$, then $M = M + 2000$, $i = i + 1$
- Transition: if $i \geq 10$, then stop

Graph:
- $M$ vs $i$
- $M$ increases linearly with $i$
- $M$ reaches $9$ at $i = 10$
Analyzing a branching

```
for (i = 0; i < 10; i++) {
    M = M + 2000;
}
```

**Graphical representation:**
- **Start:** M = 0
- **First loop:** i = 0
- **Condition:** i < 10?
- **If true:** M = M + 1000
- **If false:** i++
- **End:** M = M + 2000

**Graph:**
- X-axis: i
- Y-axis: M
- Points: (0, 0), (1, 1000), (2, 2000), ..., (9, 9000)
Convex hull

\[ \text{Convex hull} \]

\[ M = 0 \]

\[ i = 0 \]

\[ i < 10 \ ? \]

\[ M = M + 1000 \]

\[ M = M + 2000 \]

\[ i++ \]

\[ \text{stop} \]
i = 0
i < 10 ?
M = M + 1000
M = M + 2000
i++
M = 0

stop

M

i

1
10

M

i

1
10
Narrowing

\[ i = 0 \]
\[ i < 10 \] ?
\[ M = 0 \]
\[ M = M + 1000 \]
\[ M = M + 2000 \]
\[ i++ \]

Graph:

- Start at \( M = 0 \)
- If \( i < 10 \), increment \( M \) by 1000
- If \( i < 10 \), increment \( M \) by 2000
- Increment \( i \)
- Stop when \( i = 10 \)
Refined loop invariant

i = 0

M = M + 1000
M = M + 2000

i++

i < 10 ?

M = 0

stop
Invariant at loop exit

\(i = 0\)

\(i < 10 \text{ ?}\)

\(M = 0\)

\(i = 0\)

\(i < 10 \text{ ?}\)

\(M = M + 1000\)

\(i \geq 10\)

\(M = M + 1000\)

\(M = M + 2000\)

\(i++\)

\(M = M + 2000\)

Graph:

- Initial state: \(M = 0\)
- Transition: \(i = 0\)
- Condition: \(i < 10\)
- Transition: \(M = M + 1000\)
- Condition: \(i \geq 10\)
- Transition: \(M = M + 2000\)
- Transition: \(i++\)

Graph labels:
- Stop
- \(i \geq 10\)
- \(M = M + 1000\)
- \(M = M + 2000\)

Diagram:
- Axes: \(i\) horizontal, \(M\) vertical
- Scale: 10 units on \(i\) axis
- Scale: 20,000 units on \(M\) axis
- Graph lines connect states with transitions.
Static array-bound checking
The problem

- Do all array access operations occur within bounds?
- Requires the computation of numerical invariants

```c
double a[10];
for (i = 0; i < 10; i++) {
    a[i] = 1.0; ✔
}

i ∈ [0, 9] → a[i] = 1.0; ✔

i = 10 → a[i] = 0.0; ✗
```
Why is it important?

• Most critical applications are written in C (flight software, SSH, BIND)
• No runtime checks
• The memory is silently corrupted
  – Source of nondeterminism
  – Vulnerability to malicious attacks
  – Standard test practices are of little help
• About 50% of all CERT reports originate from a buffer overflow
Arrays or pointers?

• In C, every memory access goes through a pointer:

\[ a[i] = * (a + i) \]

• Tracking a pointer \( p \) requires
  – A symbolic address \( p_{\text{addr}} = \&A, \text{malloc}(...) \)
  – A numerical offset \( p_{\text{off}} \) expressed in \text{bytes}

• It is not safe to rely on the type information in C
• \text{S.f.g} is translated into \text{<&S, off(f) + off(g)>}
Example

```c
struct bytes {
    unsigned char b[4];
};
int i;
struct bytes *p = (struct bytes *)&i;
p->b[1] = 0x03;
...
```

• This comes from a real embedded application
• Byte-level granularity is required
Taxonomy (I)

• Ideal case: static allocation and bounded offsets

```c
double a[10];
for (i = 0; i < 10; i++) {
    a[i] = 1.0;
}
a[i] = 0.0;
```

• Usually occurs at the function level
  – Local manipulations on stack allocated buffers

• In practice it is a small fraction of all array accesses
Taxonomy (II)

• Interprocedural pointers and bounded offsets

```c
void f(struct S *p) {
    int i;
    for (i = 0; i < 8; i++) {
        p->a[i] = ...;
    }
}

f(&big_struct.s);
```

• Very common in embedded code
• MATLAB/Simulink autocode falls under this category
Taxonomy (III)

• Offsets and pointers are intertwined

```c
void f(double *p, int n) {
    int i;
    for (i = 0; i < n; i++) {
        p[i] = ...;
    }
}
```

• This is the worst case and is also very common

• Complex, critical codes:
  – Mars Exploration Rovers mission control software
  – Intelligent flight controllers
  – Security-sensitive applications (SSH, BIND)
What analysis to use?

• Type I:
  – Intervals at the function level

• Type II:
  – Separate pointer analysis: field sensitive, flow-insensitive, context-sensitive
  – Intervals at the function level
  – 99% accuracy on MATLAB/Simulink autocode

• Type III:
  – Relational numerical domain
  – Inline function calls and/or compute function summaries
  – Scalability is an issue
Roadmap

• There are many numerical domains available in the literature

• How to put the existing domains to work on real applications:
  – The buffer library of OpenSSH (700 LOC)
  – The flight software of Mars Exploration Rovers (550 KLOC)

• We may need different types of abstractions:
  – The gauge domain
OpenSSH

• **Description**
  – Open-source implementation of utilities based on the SSH protocol (ssh, scp, sftp, etc.)
  – Widely used, security sensitive

• **Implementation**
  – OpenSSH uses a single data structure to represent buffers
  – Cryptographic keys, deciphered messages, etc. are all stored in buffers
  – Good target for verification by static analysis
Buffer structure

typedef struct {
  u_char *buf;
  u_int alloc;
  u_int offset;
  u_int end;
} Buffer
Characteristics

• Standard FIFO queue
• 700 LOC
• Lots of Boolean logic added for fault tolerance
• The queue expands by increments if there is not enough space
  – The most complex algorithm in the library
  – “Weird” implementation using a backward goto
void *
buffer_append_space(Buffer *buffer, u_int len)
{
    u_int newlen;
    void *p;

    if (len > BUFFER_MAX_CHUNK)
        fatal("buffer_append_space: len %u not supported", len);

    /* If the buffer is empty, start using it from the beginning. */
    if (buffer->offset == buffer->end) {
        buffer->offset = 0;
        buffer->end = 0;
    }

    restart:
    /* If there is enough space to store all data, store it now. */
    if (buffer->end + len < buffer->alloc) {
        p = buffer->buf + buffer->end;
        buffer->end += len;
        return p;
    }

    /* If the buffer is quite empty, but all data is at the end, move the */
    /* data to the beginning and retry. */
    if (buffer->offset > MIN(buffer->alloc, BUFFER_MAX_CHUNK)) {
        memmove(buffer->buf, buffer->buf + buffer->offset,
                 buffer->end - buffer->offset);
        buffer->end -= buffer->offset;
        buffer->offset = 0;
        goto restart;
    }

    /* Increase the size of the buffer and retry. */
    newlen = buffer->alloc + len + 32768;
    if (newlen > BUFFER_MAX_LEN)
        fatal("buffer_append_space: alloc %u not supported",
              newlen);
    buffer->buf = xrealloc(buffer->buf, newlen);
    buffer->alloc = newlen;
    goto restart;
/* NOTREACHED */
}
Appending data to the buffer

```c
void buffer_append(Buffer *buffer, const void *data, u_int len)
{
    void *p;
    p = buffer_append_space(buffer, len);
    memcpy(p, data, len);
}
```

Automatically prove that the operation stays within the bounds of the buffer
Design of the analysis

- The expressive power of convex polyhedra is required
- Inlining the library into the OpenSSH code is not conceivable
- Modular approach:
  - We build a simplified model of a client of the library on one buffer
  - The client nondeterministically calls functions of the library on the buffer with consistent arguments
  - We inline the library code into the client and analyze it
volatile u_int random;
Buffer buffer;

buffer_init(&buffer);
for(random) {
    switch(random) {
        case 0: {
            u_int len = random;
            u_char *data = malloc(len);
            buffer_append(buffer, data, len);
            break;
        }
        ...
    }
}
First try

• Settings
  – Polyhedral domain: Bertrand Jeannet’s New Polka
  – C front-end: CIL
  – Fixpoint iterator: Bourdoncle’s algorithm

• Running the analysis:
  – Failure
  – The widening operation on polyhedra crashes because there are too many variables
Optimizations

• The front-end generates a lot of auxiliary variables, which weigh on the polyhedral domain
• Inlining also introduces lots of redundancy
• We run initial passes that perform:
  – Constant propagation
  – Copy propagation
  – Dead variable elimination
• The number of variables is greatly reduced
• New run: Crash!
A bit of head scratching

- The crash always occurs during the widening
- We make two observations:
  - The invariants contain a lot of linear equalities
  - Most of these equalities are common to both operands of the widening
- We decide to remove the common equalities from the invariants, apply the widening and add them back to the result
It works!

• The analysis runs in few seconds
• But all the nontrivial checks are flagged as warnings...
• It finally scales but now it’s not precise enough
• The problem comes from the logic inserted to make the library robust
```c
int
buffer_consume_ret(Buffer *buffer, u_int bytes)
{
    if (bytes > buffer->end - buffer->offset) {
        error("buffer_consume_ret: trying to get more bytes than in buffer");
        return (-1);
    }
    buffer->offset += bytes;
    return (0);
}

void
buffer_consume(Buffer *buffer, u_int bytes)
{
    if (buffer_consume_ret(buffer, bytes) == -1)
        fatal("buffer_consume: buffer error");
}
```

Join of invariants
Loss of precision
Solution

• We could use trace partitioning techniques (Rival & Mauborgne)
  – Dramatically complicates the analysis

• We are only interested in execution traces that do not abort
  – We model the *fatal* function as *bottom*
  – We perform an iterated forward/backward analysis between the beginning and the end of each library operation

• **Full verification is achieved in 35 seconds!**
Observations

• If we turn off the initial optimizations the analyzer crashes
• How far can we push the scalability with the optimized widening?
• Not very far
  – We added one variable to the main loop of the client
  – The analyzer crashes
• The approach based on a general-purpose expressive domain seems very brittle
Mars Exploration Rovers

• Large flight software (550+ KLOC)
• Developed with an object-oriented approach
• Thousands of small generic functions
• Our approach:
  – Compute function summaries
  – No loops in summaries, just numerical invariants and symbolic pointer constraints
  – Use a weakly relational numerical domain to achieve scalability: difference-bound matrices (DBMs)
Example

```c
void assign(double *p, double *q, int n) {
    int i;
    for (i = 0; i < n; i++) {
        p[i] = q[i];
    }
}
```

\[ p_{\text{off}} \leq x \leq p_{\text{off}} + 8n \]

• Not expressible in the domain of DBMs or even octagons
Templates for pointer arithmetic

• We introduce a symbolic expression based on the syntax of the pointer expression from the AST:

\[ p[i][j] \rightarrow b + k_1 o_1 + k_2 o_2 \]

• Constraints on the parameters of the template are expressible as DBMs:

\[ \begin{align*}
  b &= p_{\text{off}} \\
  k_1 &= 64 \\
  o_1 &= i \\
  k_2 &= 8 \\
  o_2 &= j
\end{align*} \]
Scalability

• We can express general linear inequalities at the price of a larger number of variables

• First experiments are a disaster
  – It takes hours to analyze a single function
  – The DBMs were supposed to scale better (cubic in the worst case)

• The problem is that the upper complexity bound is always attained!
Explanation

• Range constraints in DBMs (or octagons) are expressed using a special variable $Z$ that is semantically equal to 0

• $x = [a, b]$ is expressed as $x - Z \leq b$ and $Z - x \leq -a$

• Variables in a program are always initialized (hopefully)

• The graph of unitary relations over the program variables is then strongly connected
  – Worst case for the closure algorithm
Variable packing

• A solution is to only consider relations over small sets of variables like in ASTREE

• Problem:
  – A good packing can be determined statically in ASTREE because of the specificities of the code considered
  – In our case we have a fairly general C program

• Our approach:
  – Dynamic variable packing at analysis time
  – Variables appearing in a statement are put together
Technicalities

• Doing dynamic packing is not straightforward as partitions must be merged on the fly:
  – Complex domain structure (cofibered domain)

• Implicit relations must be taken into account:

```c
for(...) {
    i++;  // i = j
    j++;
}
```

• Variables modified within a loop are put in the same pack
Outcomes

• The whole MER flight software can be analyzed in less than 24 hours
• The precision is over 80%
• Downsides of the approach:
  – Scalability is achieved at the price of a careful and complex engineering
  – There isn’t much margin left to improve on the precision
Scalability and precision?
The gauge domain

• The domain of polyhedra is expressive enough but doesn’t scale
• Weakly relational domains scale better (somewhat) but are not expressive enough
• Design a specialized domain for a certain type of invariants: the gauge domain
  – Focuses on finding implicit loop invariants among variables
From Intervals to Gauges

• Intuitively, a gauge is an integer interval that linearly varies across the iteration space.

• Interval:
  \[ a \leq x \leq b \]

• Gauge:
  \[ a_0 + a_1 \lambda_1 + \ldots + a_n \lambda_n \leq x \leq b_0 + b_1 \lambda_1 + \ldots + b_n \lambda_n \]

  – \( \lambda_1, \ldots, \lambda_n \geq 0 \)
  – \( a_i \leq b_i \)
  – The parameters \( \lambda_i \) denote the iteration counters of all enclosing loops.
Exposing Loop Counters

• We label each loop with a fresh counter $\lambda$
• We introduce operations on the $\lambda$’s to model the semantics of loop iterations

```c
int i = 0;
while (i < 10) {
    int j = 0;
    while (j < i) {
        ...
        j++;
    }
    i++;
}
```

```c
int i = 0; new $\lambda_1$
$\lambda_1$: while (i < 10) {
    int j = 0; new $\lambda_2$
    $\lambda_2$: while (j < i) {
        ...
        j++; inc $\lambda_2$
    } forget $\lambda_2$
    i++; inc $\lambda_1$
} forget $\lambda_1$
```

• This is an entirely automated process
How do we compute gauges?

\[ j = 1; \]
\[ \lambda: \text{for } (i = 0; i < 10; i++) \{ \]
\[ \text{if } (...) \{ \]
\[ j += 2; \]
\[ \} \text{ else } \{ \]
\[ j += 3; \]
\[ \} \]

Linear Interpolation

\[ j \]
\[ \lambda \]
\[ 0 \]
\[ 1 \]
\[ 1 \]
\[ 4 \]
\[ 4 \]
\[ 0 \]
\[ 3 \]
\[ 1 + 2\lambda \]
\[ 1 + 3\lambda \]
Computational Complexity

| Variables | = n
| Loop Depth | = k

- Joins and widenings: $O(\text{kn})$
- Arithmetic operations: $O(k)$
- Loop operations (new, forget, inc): $O(\text{kn})$
- If k is assumed bounded:
  - Linear complexity for domain operations
  - Constant complexity for semantic transformers
  - It is the same complexity as the domain of intervals
Experimental Results

- Buffer-overflow analysis performed on an intelligent flight control system developed at NASA
- 144 KLOC of C
- Complex adaptive avionics
- Analyses run on a laptop
  - Commercial tool: high-end server with 32 cores and 64GB memory

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Analysis Time</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intervals + Complete Inlining</td>
<td>41 min</td>
<td>79%</td>
</tr>
<tr>
<td>Commercial Tool</td>
<td>5 hours</td>
<td>91%</td>
</tr>
<tr>
<td>Octagons</td>
<td>&gt; 27 hours</td>
<td>N/A</td>
</tr>
<tr>
<td>Gauges</td>
<td>10 min</td>
<td>91%</td>
</tr>
</tbody>
</table>
Unexpected benefits

- Some loops in the MATLAB/Simulink autocode have an unusual control structure:

```c
p = &a[0];
i = 10;
while (i != 0) {
    *p++ = ...;
    i--;
}
```

- This is bad for static analysis where only inequalities can be analyzed precisely T
  - The 1% not resolved by intervals
Gauges can help

• Relation between variables and loop counters

```c
p = &a[0];
i = 10;
while (i != 0) {
    *p++ = ...;
i--;
}
i = 10 - \lambda
p = 4\lambda
```

• Since counters are monotonic and positive, we can automatically replace the test with \( i > 0 \)

• We obtain 100% precision
Limitations of gauges

• The domain only provides information inside loops
  – The $\lambda$’s are loop counters
• Outside of loops gauges are mere intervals
• Gauges have to be combined with other domains using the reduced product

\[ D = \text{Gauge} \times D_1 \times D_2 \times \ldots \]
Perspectives

• There are many numerical domains available but few have been applied to real code
• We believe in combining simpler, specialized and efficient abstract domains over using a monolithic approach
• We are still a long way from being to able to automatically verify security-sensitive applications, even small ones