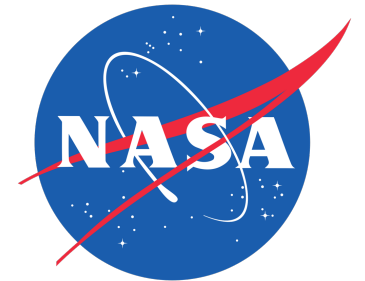
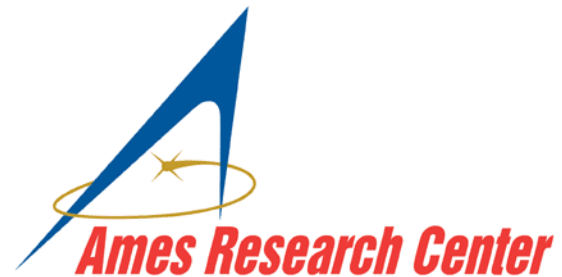




IKOS: Sound Static Program Analysis

Guillaume Brat



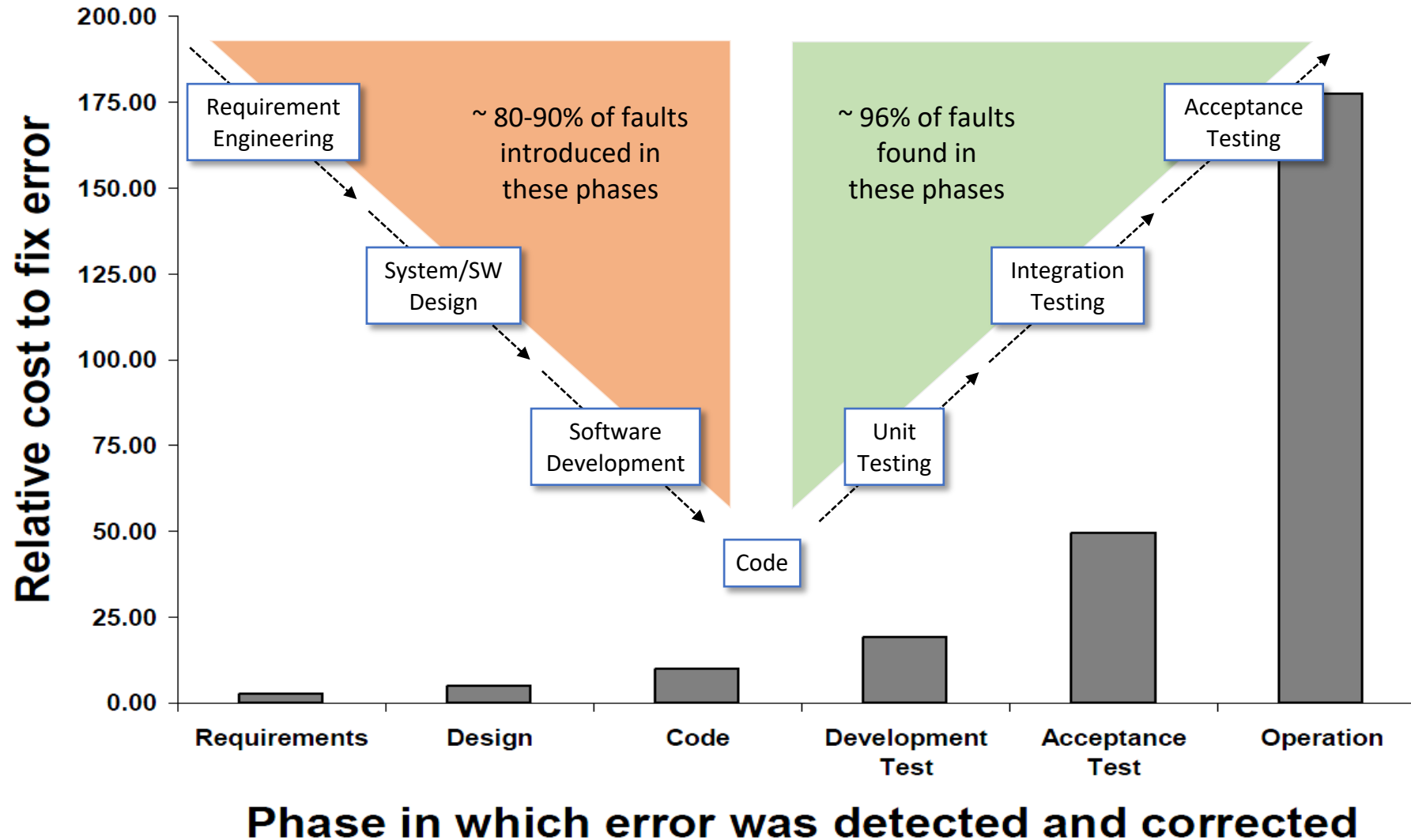
Acknowledgement



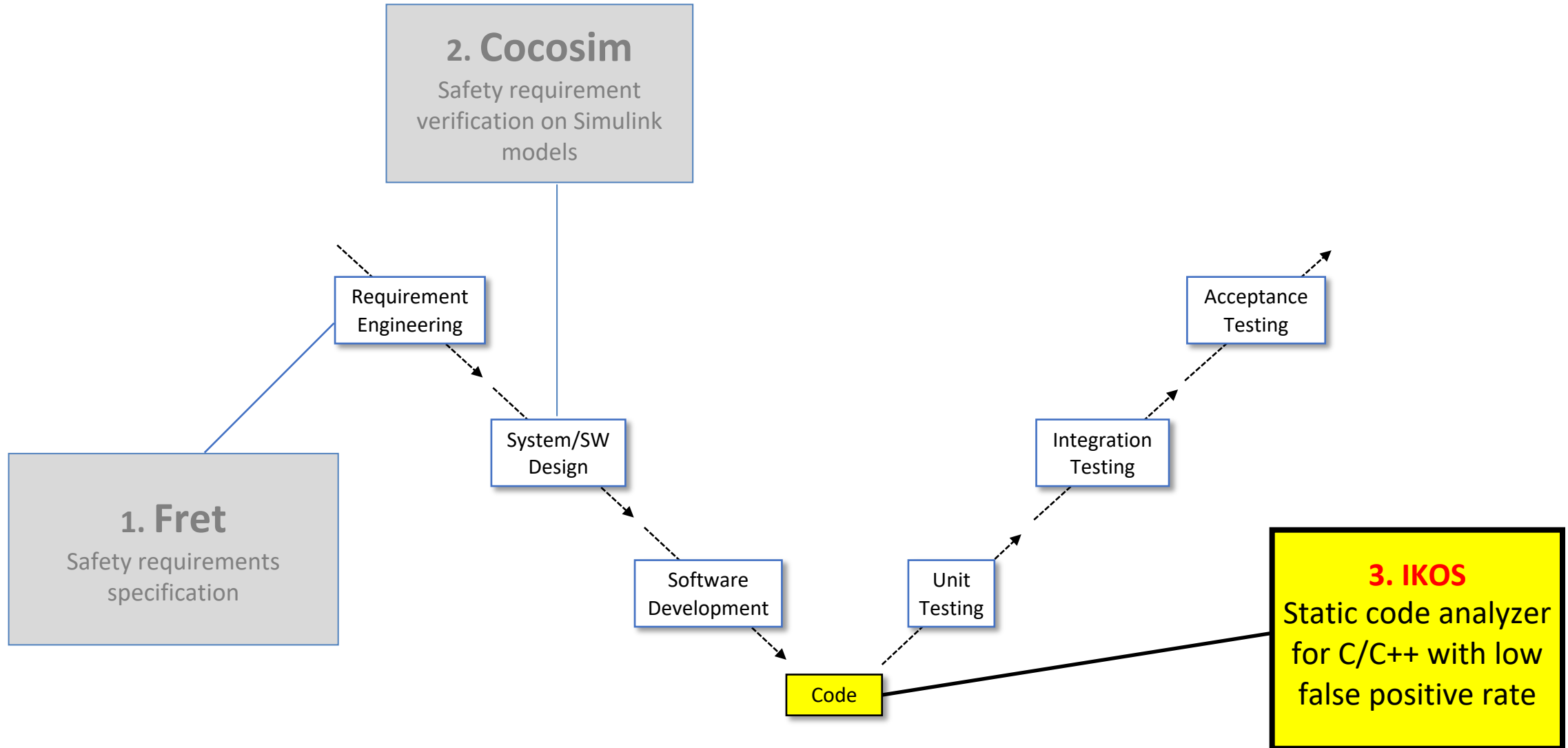
System-Wide Safety

This research was funded
by the System-Wide Safety Project
in the Airspace Operations & Safety Program
under the Aeronautics Research Mission Directorate at NASA

Motivation: V&V Cost Analysis



IKOS



IKOS

IKOS performs a **compile-time** analysis of a C/C++ source code.
It can **detect** or **prove the absence** of **runtime errors**.

```
int tab[10];  
  
for (int i = 0; i < 10; i++) {  
    tab[i] = i * i;  
}
```

C/C++ code

IKOS
Static
Analyzer

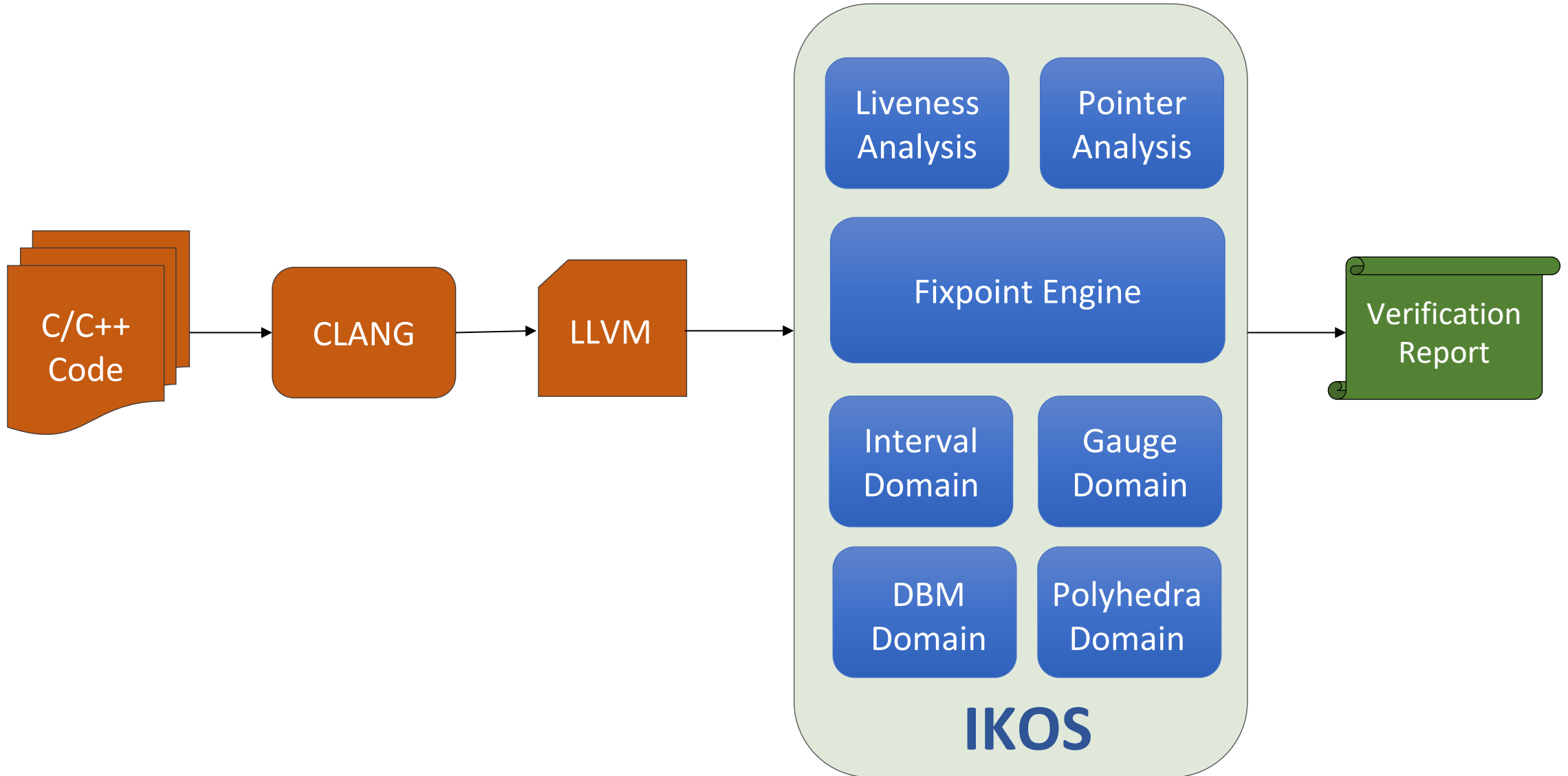
List of (possible) runtime errors:

- Buffer Overflows
- Null pointers
- Division by Zero
- Uninitialized Variables
- Assertion Prover
- Etc.

IKOS is **NOT** a code style checker

IKOS is **NOT** a compiler: It can detect errors that compilers cannot catch

IKOS Design



Verification Report

- **Safe**: The instruction is **proven free of runtime errors**
- **Error**: The instruction **always produces a runtime error**
- **Warning**:
 - The instruction **can produce an error** depending on the input
 - The instruction is **safe** but IKOS could **not prove it** (also called **false positive**)

Example


```
int tab[10];  
  
for (int i = 0; i < 10; i++) {  
    tab[i] = i * i;  
}
```

- The analysis discovers program properties: $0 \leq i \leq 9$

Example

```
int tab[10];  
  
for (int i = 0; i < 10; i++) {  
    tab[i] = i * i;  
}
```

Access within bounds?



- The analysis discovers program properties: $0 \leq i \leq 9$
- The verification uses the properties discovered:
 - Array-bound compliance
 - Check that array **tab** has at least 10 elements

IKOS Checks

- Buffer overflow
- Division by zero
- Null pointer dereference
- Assertion prover
- Unaligned pointer
- Uninitialized variable
- Integer overflow (signed, unsigned)
- Invalid bit shift
- Invalid pointer comparison
- Invalid function pointer call
- Dead code
- Double free and Invalid lifetime

IKOS Abstract Domains

Domain	Constraints	Complexity
Interval	$x \in [a, b]$	n
Congruence	$x \in aZ+b$	n
Gauge	$x \in [a*i + b*k + \dots, a'*i + b'*k + \dots]$	$K*n$
Difference Bound Matrices	$x - y \in [a, b]$	n^3
APRON Octagon	$x \pm y \in [a, b]$	n^3
APRON Polka Polyhedra	$a*x + b*y + \dots + c \leq 0$	Exponential
APRON PPL Polyhedra	$a*x + b*y + \dots + c \leq 0$	Exponential
Variable Packing of	n

Live demo

IKOS Installation

- Supported platforms:
 - Mac OS
 - Linux
 - Windows (using *MinGW*)
- **Dependencies** can be installed with a **package manager** ([brew](#), [apt-get](#), [yum](#), ..)
- **Installation instructions** for each platform available in: [doc/install/](#)
- **Bootstrap script** for **non-admin** installations: downloads and compiles all missing dependencies (*slow*)

IKOS Usage

- Analyze a single file: `ikos file.c`
 - Runs the analysis
 - Prints the results
 - Generates an output database containing the analysis results: `output.db`
- Analyze a whole project:
 - `ikos-scan make`
 - `ikos program.bc`
- Generate a report from an output database: `ikos-report output.db`
- Examine the results in a graphical interface: `ikos-view output.db`

IKOS-SCAN

- Analyze a whole project with: `ikos-scan <command>`
- It compiles all executables to LLVM bitcode: `program.bc`
- It runs IKOS on the LLVM bitcode: `ikos program.bc`
- Works with most build systems: `Make`, `CMake`, `Autoconf`, etc...
- Works by overriding environment variables: `CC`, `CXX`, `LD`

IKOS-SCAN

Live demo

Analyzing a library

- The analysis needs an **entry point** (i.e, `main`)
- **Workaround:** create a small program that uses the library
- Extract the LLVM bitcode from an object file: `ikos-scan-extract file.o`
- Analyze a program with a specific entry point: `ikos file.bc -e=MyMain`

IKOS-VIEW

- **Graphical interface** to examine the analysis results
- Starts a **web server** in the terminal, opens the default **browser**
- `ikos-view output.db`

IKOS-VIEW

Live demo

IKOS Abstract Domains Guidelines

- Start with **fast** but **imprecise** domain
- Go towards **slow** but **precise** domain
- Stop when the analysis is too slow for your use case
- Recommended order:
 - Interval: `-d=interval`
 - Gauge + Interval + Congruence: `-d=gauge-interval-congruence`
 - Variable Packing DBM: `-d=var-pack-dbm`
 - Variable Packing Polyhedra: `-d=var-pack-apron-ppl-polyhedra`

IKOS Assumptions

- The source code is compiled with Clang for the **host architecture**
- Clang defines:
 - The data model (size of types)
 - The memory layout (alignments)
 - The endianness
 - The semantic of floating points
 - Etc...

IKOS Assumptions

- The program is **single-threaded**
- The program does **not** receive **signals** or **interrupts**
- **Unknown extern functions:**
 - Do not update global variables
 - Can write on their pointer arguments
 - Do not call user-defined functions (no callbacks)
- **Assembly code** is treated as a call to an unknown extern function
- **Recursive functions** can update any value in memory

False positives

- **False positive:** invalid warning
- Objective: **low rate of false positives**
- Common source of false positives:
 - Unknown library functions
 - “Bad” code patterns
 - Imprecision of the analysis

Modeling library functions

- The analyzer does **not** require the **libraries** used by your program
- Unknown library functions will trigger a warning ("ignored call side effect" in [ikos-view](#))
- Modeling library functions can **reduce the number of warnings**
- Write "stubs": fake implementations of library functions

Modeling library functions

```
#include <ikos/analyzer/intrinsic.h>

char* fgets(char* restrict str,
            int size,
            FILE* restrict stream) {
    __ikos_assert(size >= 0);
    __ikos_forget_mem(stream, sizeof(FILE));
    __ikos_abstract_mem(str, size);
    errno = __ikos_nondet_int();
    return __ikos_nondet_int() ? str : NULL;
}
```

IKOS Annotations

- Annotating your source code can **reduce the number of warnings**
- List of intrinsic functions: [analyzer/include/ikos/analyzer/intrinsic.h](#)
 - `__ikos_assert(condition)`
 - `__ikos_assume(condition)`
 - `__ikos_nondet_int()`
 - `__ikos_check_mem_access(ptr, size)`
 - `__ikos_assume_mem_size(ptr, size)`
 - `__ikos_forget_mem(ptr, size)`
 - `__ikos_abstract_mem(ptr, size)`
 - `__ikos_print_values(description, var)`

IKOS Annotations

```
ret = talg->parse_algoid_params(buf, param_len, param);
```

IKOS Annotations

```
ret = talg->parse_algoid_params(buf, param_len, param);
```

```
int (*fun)(const u8*, u16, alg_param*) =  
    talg->parse_algoid_params;  
__ikos_assume(fun == parse_algoid_params_generic ||  
              fun == parse_algoid_params_ecdsa_with ||  
              fun == parse_algoid_params_ecPublicKey ||  
              fun == parse_algoid_params_rsa);  
ret = fun(buf, param_len, param);
```

Bad code pattern (1)

```
CommandResult = XXX();  
if (CommandResult == TRUE) {  
    FilenameState = YYY();  
    if (FilenameState == FM_NAME_IS_INVALID) {  
        CommandResult = FALSE;  
    }  
}  
if (CommandResult == TRUE) {  
    CommandResult = ZZZ();  
}  
if (CommandResult == TRUE) {  
    // ...  
}  
return CommandResult;
```

Bad code pattern (1)

- Bad readability
- Prone to errors
- Hard for static analyzers
- Please use “early return on errors”

Bad code pattern (1)

```
CommandResult = XXX();  
if (!CommandResult) {  
    return FALSE;  
}  
CommandResult = YYY();  
if (CommandResult == FM_NAME_IS_INVALID) {  
    return FALSE;  
}  
CommandResult = ZZZ();  
if (!CommandResult) {  
    return FALSE;  
}  
// ...  
return TRUE;
```

Bad code pattern (2)

- Single global variable containing everything

```
AppData_t g;

typedef struct {
    PipeId_t CmdPipeId;
    uint16 usCmdPipeDepth;
    char cCmdPipeName[OS_MAX_API_NAME];
    int32 ulfd;
    uint32 uiRunStatus;
    // ...
    uint8 lastCmdBchErrorStatus;
} AppData_t;
```


Bad code pattern (2)

- Makes the buffer overflow analysis harder
- Please split it into different global variables

Bad code pattern (3)

- Small integers for loop counters

```
void f(uint16_t n) {  
    for (uint16_t i = 0; i < n; i++) {  
        // ...  
    }  
}
```

Bad code pattern (3)

- Small integers for loop counters

```
void f(uint16_t n) {  
    for (uint16_t i = 0; i < n; i++) {  
        // ...  
    }  
}
```

- Integer promotion rules of C

```
void f(uint16_t n) {  
    for (uint16_t i = 0;  
        (unsigned int)i < (unsigned int)n;  
        i = (uint16_t)((unsigned int)i + 1)) {  
        // ...  
    }  
}
```

Bad code pattern (3)

- Creates temporary variables in the LLVM bitcode
- Leads to imprecision of the analysis
- Please use `size_t` (or `int`) for loop indexes

Imprecision

- Initialization functions returning an error code

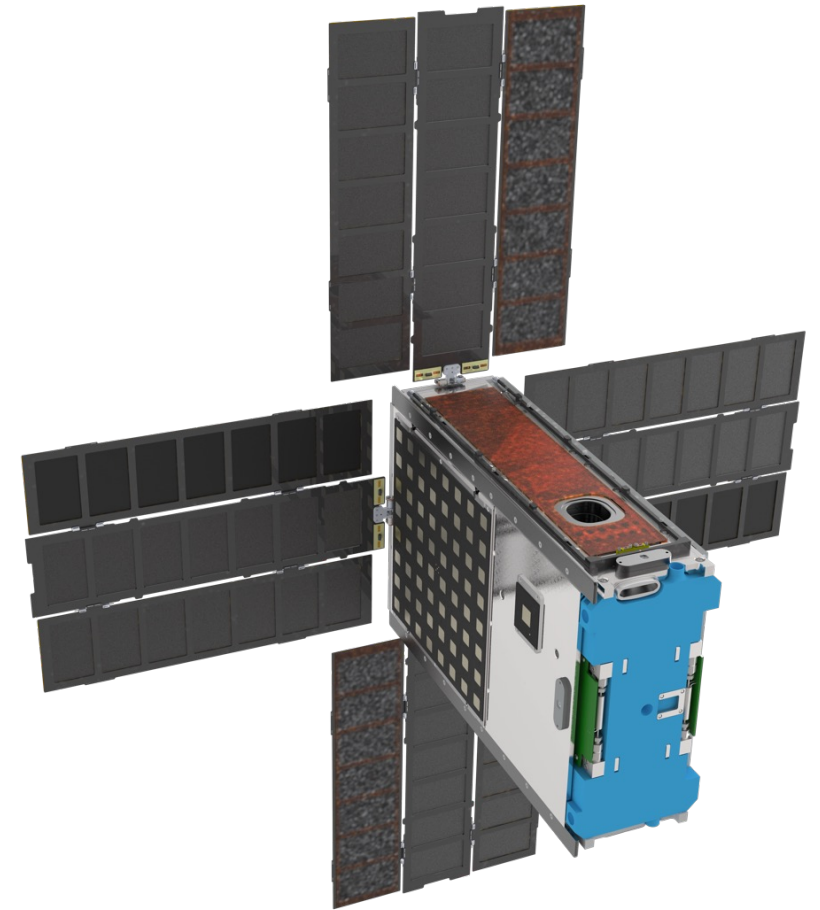
```
int Init(void) {  
  
    int status = Register();  
    if (status != SUCCESS) {  
        return status;  
    }  
  
    status = InitEvent();  
    if (status != SUCCESS) {  
        return status;  
    }  
  
    // ...  
}
```

Imprecision

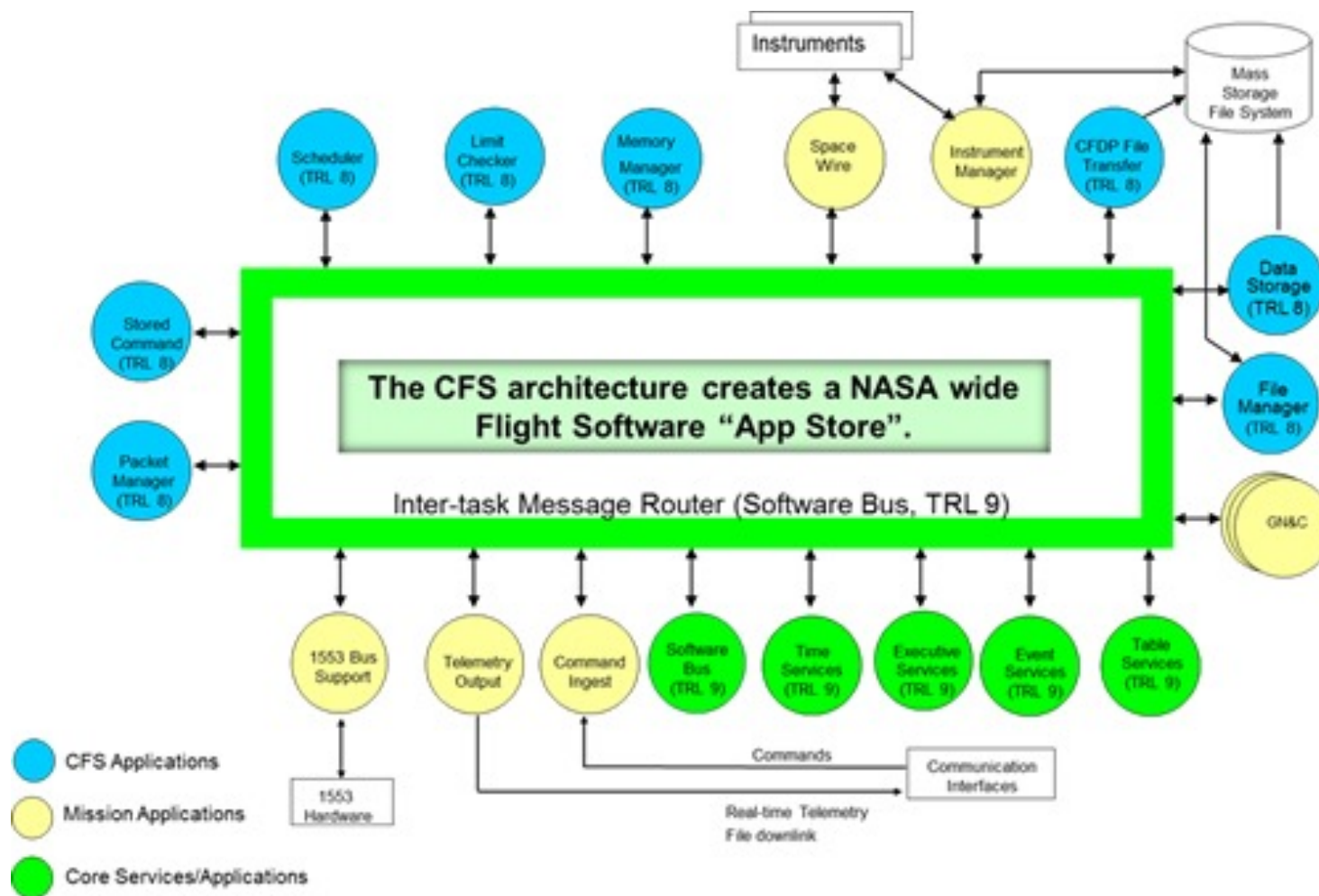
- Imprecision due to the abstract union in the analysis
- Temporary workaround: add `exit(0)` for each failure branch (ugly!)
- Proper fix in the next update – using partitioning

Success Story: BioSentinel

- Space biology mission
- CubeSat spacecraft
- Developed at NASA Ames, in collaboration with JPL, JSC, MSFC
- Flight software built on top of CFS



Success Story: BioSentinel



Success Story: BioSentinel

- Each application was analyzed with IKOS
- The CFE framework was modeled to improve the analysis (~ 1200 LOC)
- Low rate of warnings: 1.31% in average
- Found ~ 17 real bugs

Success Story: BioSentinel

Application	Abstract Domain	Time	Errors	Warnings	Warnings%	Checks
adio	var-pack-dbm	1 min 6.92 sec	0	1	0.07%	1334
brdio	var-pack-dbm	8.02 sec	0	8	0.97%	818
ci	var-pack-dbm	19.98 sec	0	6	0.65%	923
comio	var-pack-dbm	1 min 4.83 sec	0	4	0.26%	1494
epsio	var-pack-dbm	30.64 sec	0	5	0.42%	1181
letio	var-pack-dbm	24.33 sec	0	18	1.64%	1095
ms	interval	0.16 sec	0	0	0%	444
saio	var-pack-dbm	22.35 sec	0	8	0.64%	1246
sensio	var-pack-dbm	4.67 sec	0	79	9.56%	826
spe	interval	0.16 sec	0	0	0%	445
thrio	var-pack-dbm	19.33 sec	0	4	0.38%	1043
to	var-pack-dbm	2 min 18.32 sec	0	33	1.98%	1666
xactio	var-pack-dbm	22.18 sec	0	6	0.51%	1165

BioSentinel Bug Example

warning: Possible buffer overflow, pointer '&cmd[n + 2]' accesses 1 bytes at offset between 8 and 16 bytes of local variable 'cmd' of size 16 bytes

```
uint8_t cmd[16];
uint8_t n;
// ...
switch(cmd_request) {
    case CMD_OPEN:
        n = CMD_OUT + CMD_OPEN; // 6 + 8 = 14
        break;
    // ...
}
// ...
cmd[n + 2] = 0; // 14 + 2 = 16
```

Troupe1 example

- Troupe1 is another cFS application for a swarm of Rovers.
- The build being a bit complicated, running IKOS is done as follows:
 - Use `wllvm` in the build to create `.bc` files in a build tree customized for IKOS
 - Use `llvm-link` to link these `.bc` files into one `.bc` file
 - Call IKOS on that file with all the apps “main” as entry points
 - Produce a report in SARIF format so that it can be viewed in VisualStudio
- These steps were bundled into a script that’s being called by the Bamboo continuous integration box at every build
 - Note: one app (Arduover) had to be moved to a regular weekly schedule because it’s analysis by IKOS is taking over 1 hour.

```
export LLVM_COMPILER_PATH=/usr/local/ikos/llvm-9.0.0/bin/
export LLVM_COMPILER=clang
export CC=/usr/local/bin/wllvm
export CXX=/usr/local/bin/wllvm++
make SIMULATION=native O=build-ikos prep
make O=build-ikos

bc_files=$(find . -name "*.bc" | grep "/apps/" | grep
'cam.dir\|SLAM_APP\|locio_app\|sch.dir')

curdir=$(pwd)

analysis_name=ikos_analysis

llvm-link ${bc_files} -o ${curdir}/${analysis_name}.bc

object_file=${curdir}/${analysis_name}.bc

entry_point="CAM_AppMain,slam_AppMain,LOCIO_AppMain,SCH_Ap
pMain"

ikos "${object_file}" -e="${entry_point}" -d=var-pack-dbm
-f text --partitioning=return

ikos-report -f sarif -o ${analysis_name}.sarif output.db
```

Guidelines

- Use a lightweight static analyzer first: `cppcheck`, `clang-tidy`, `pvs-studio`, etc.
- Use `ikos-scan` to generate the llvm bitcode (`.bc`): `ikos-scan make`
- Use `ikos` on the llvm bitcode (`.bc`): `ikos program.bc`
- Try different abstract domains: `ikos -d=var-pack-dbm program.bc`
- Use `ikos-view` to examine the results: `ikos-view output.db`
- (Optional) Model key library functions
- (Optional) Annotate the code
- (Optional) Avoid “bad” patterns
- (Optional) Add ikos in your continuous build system?

IKOS at a glance

- IKOS is a **static analyzer** for **C/C++** targeting **safety critical** software
- IKOS is **open source**: <https://github.com/NASA-SW-VnV/ikos>
- Contact: ikos@lists.nasa.gov

Thank you.

Questions?