Tools for correctness, part 2

We are here

- Part 1: How computers works
  - Boolean logic, integers
  - Instructions
  - Memory
- Part 2: Software development
  - Compiling (clang, make, …)
  - Architectures, portability (ABIs, …)
  - Code management (regex, git)
- Part 3: Correctness
  - Specifications
  - Documentation, testing
  - Static & dynamic analysis, debugging $\leftarrow$ TODAY
- Part 4: Performance
  - CPU pipelines, caches
  - Data structures
  - Parallel computation

Static code analysis

- **Static** analysis operates on the source code

  (before any assembly or executable code is produced)
- Compilers do advanced case analysis on the code

  (in order to produce faster code)
- The same analysis can be used to find (potential) bugs

- Not an exact science
  - Relies on heuristics to detect hazardous code
  - Suffers from false negatives and false positives

Clang’s static analyzer

If you use a `Makefile`, run

```
scan-build make
```

> result

Python linters

- A “linter” is a static analyzer
- Typically, linters enforce a specific coding style

Examples:

- Pylint
- flake8
- mypy (adds static type checking)

```python
def fib(n):
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

```python
def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

Dynamic code analysis

- **Dynamic** analysis operates on the running executable
  (during testing)
- by adding runtime checks
- can find more bugs than static analysis…
- … but only for those bugs are triggered by some test!

Sanitizers

With **sanitizers**, runtime checks are added by the **compiler**.

UBSan

- The “undefined behavior sanitizer” detects many types of undefined behavior (at runtime)
- triggers an immediate crash (with an explanation message)
- Pass “`-fsanitize=undefined`” to `gcc` or `clang`

```c
#include <stdio.h>
#include <stdlib.h>

int f(int a, int b)
{
    printf("a = %d, b = %d\n", a, b);

    int r = a / b;

    printf("We survived!\n");

    return r;
}

int main(int argc, char **argv)
{
    int i = (argc < 2) ? 5 : strtol(argv[1], NULL, 0);
    int r = f(10, i);
    printf("r = %d\n", r);
}
```

Without UBSan:

```
gcc -O3 -o timetravel timetravel.c
./timetravel 0
```

```
a = 10, b = 0
We survived!
Floating point exception (core dumped)
```

With UBSan:

```
clang -O3 -fsanitize=undefined -o timetravel timetravel.c
./timetravel 0
```

```
a = 10, b = 0
timetravel.c:8:12: runtime error: division by zero
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior timetravel.c:8:12 in
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==3245281==ERROR: UndefinedBehaviorSanitizer: FPE on unknown address 0x00000042b43d (pc 0x00000042b43d bp 0x7ffdb30690f0 sp
    #0 0x42b43d in f /home/poirrier/courses/softeng/code/std/timetravel.c:8:12
    #1 0x42b43d in main /home/poirrier/courses/softeng/code/std/timetravel.c:18:10
    #2 0x7fd43af4db89 in __libc_start_call_main (/lib64/libc.so.6+0x27b89) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a9442adcd
    #3 0x7fd43af4dc4a in __libc_start_main@GLIBC_2.2.5 (/lib64/libc.so.6+0x27c4a) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a9
    #4 0x4033d4 in _start (/home/poirrier/courses/softeng/code/std/timetravel+0x4033d4) (BuildId: a42ae4bf9188c9d93ff828ccd

UndefinedBehaviorSanitizer can not provide additional info.
SUMMARY: UndefinedBehaviorSanitizer: FPE /home/poirrier/courses/softeng/code/std/timetravel.c:8:12 in f
==3245281==ABORTING
```

```c
#include <stdlib.h>
#include <stdio.h>

static int (*function_pointer) ();

static int erase_all_files()
{
    return printf("Deleting all your files\n");
}

void this_function_is_never_called()
{
    function_pointer = erase_all_files;
}

int main() {
    return (*function_pointer) ();
}
```

```
clang -O3 -fsanitize=undefined -o ub_del ub_del.c
./ub_del
```

```
Deleting all your files
```

Pros

- Fixes (in some cases) the anything-can-happen problem with undefined behavior.

  We get a crash with an explanation instead.
- No false positives

Cons

- Not all types of undefined behavior detected (most are)
- Does not always stop the compiler from exploiting undefined behavior
- Overhead (~3x slowdown)
- Needs good tests (good in combination with fuzzing)

ASan

- The “address sanitizer” detects many types memory access errors (at runtime)
- Separate from UBSan because it uses different mechanisms
- triggers an immediate crash (with an explanation message)
- Pass “`-fsanitize=address`” to `gcc` or `clang`

```c
#include <stdio.h>

char *f()
{
    char buffer[16];

    snprintf(buffer, sizeof(buffer), "Hello");

    return buffer;
}

int main()
{
    char *s = f();

    printf("Here is the return value of f():\n");
    printf("%s\n", s);
    return 0;
}
```

```
clang -O3 -fsanitize=address -o bug bug.c
./bug
```

```
Here is the return value of f():
=================================================================
==3245688==ERROR: AddressSanitizer: stack-use-after-scope on address 0x7f604b800020 at pc 0x00000043cd41 bp 0x7ffd5bb0da70
READ of size 1 at 0x7f604b800020 thread T0
    #0 0x43cd40 in puts (/home/poirrier/courses/softeng/code/std/bug+0x43cd40) (BuildId: fd60803d545d3b62b6353b1498d16e17a
    #1 0x4f39d1 in main (/home/poirrier/courses/softeng/code/std/bug+0x4f39d1) (BuildId: fd60803d545d3b62b6353b1498d16e17a
    #2 0x7f604d60db89 in __libc_start_call_main (/lib64/libc.so.6+0x27b89) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a9442adc
    #3 0x7f604d60dc4a in __libc_start_main@GLIBC_2.2.5 (/lib64/libc.so.6+0x27c4a) (BuildId: 3ebe8d97a0ed3e1f13476a02665c5a
    #4 0x41d324 in _start (/home/poirrier/courses/softeng/code/std/bug+0x41d324) (BuildId: fd60803d545d3b62b6353b1498d16e1

Address 0x7f604b800020 is located in stack of thread T0 at offset 32 in frame
    #0 0x4f393f in main (/home/poirrier/courses/softeng/code/std/bug+0x4f393f) (BuildId: fd60803d545d3b62b6353b1498d16e17a

  This frame has 1 object(s):
    [32, 48) 'buffer.i' <== Memory access at offset 32 is inside this variable

. . .
```

ASan detects (1)

- Out-of-bounds accesses to heap, stack and globals

```
int a[10];

printf("%d\n", a[20]);
```

- Use-after-free

```
free(pointer);

printf("%d\n", *pointer);
```

ASan detects (2)

- Use-after-return

```
int *f()
{
    int a[10];
    return a;
}

void g()
{
    int *pointer = f();
    printf("%d\n", pointer[0]);
}
```

- Use-after-scope

```
void g()
{
    int *pointer;

    if (1) {
        int a[10];
        pointer = a;
    }

    printf("%d\n", pointer[0]);
}
```

ASan detects (3)

- Double-free, invalid free

```c
void *other_pointer = pointer;

free(pointer);
free(other_pointer);
```

```c
int a[10];
free(a);
```

- Memory leaks

```c
void f()
{
    void *ptr = malloc(10);
}
```

Pros

- Detects most memory issues
- No false positives

Cons

- Not every memory issue detected (many are)
- Speed overhead (~2x slowdown)
- Memory overhead (~2x memory usage)
- Needs good tests (good in combination with fuzzing)

Valgrind

- Valgrind adds runtime checks on already-compiled executable.
- It is a hybrid interpreter / JIT compiler for machine code.
- It adds checks around all memory accesses.
  - Detects uses of invalid pointers (incl. uninitialized memory)
  - Detects memory leaks (at exit)

For readable debug messages, Valgrind requires compiling with the “-ggdb” option (gcc / clang)

```
valgrind --leak-check=full ./truthtable all ../data/parse_04.cnf
```

```
==3244248== Memcheck, a memory error detector
==3244248== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3244248== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==3244248== Command: ./truthtable all ../data/parse_04.cnf
==3244248==
../data/parse_04.cnf: -3 is out of bounds (n = 2)
==3244248==
==3244248== HEAP SUMMARY:
==3244248==     in use at exit: 262,144 bytes in 1 blocks
==3244248==   total heap usage: 3 allocs, 2 frees, 266,712 bytes allocated
==3244248==
==3244248== 262,144 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3244248==    at 0x484182F: malloc (vg_replace_malloc.c:431)
==3244248==    by 0x4023EF: di_push (parse.c:94)
==3244248==    by 0x4023EF: dimacs_parse_f (parse.c:215)
==3244248==    by 0x402541: dimacs_parse (parse.c:268)
==3244248==    by 0x401201: run (main.c:12)
==3244248==    by 0x401201: main (main.c:62)
==3244248==
==3244248== LEAK SUMMARY:
==3244248==    definitely lost: 262,144 bytes in 1 blocks
==3244248==    indirectly lost: 0 bytes in 0 blocks
==3244248==      possibly lost: 0 bytes in 0 blocks
==3244248==    still reachable: 0 bytes in 0 blocks
==3244248==         suppressed: 0 bytes in 0 blocks
==3244248==
==3244248== For lists of detected and suppressed errors, rerun with: -s
==3244248== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Pros

- Detects almost all memory issues (that happen at runtime)

Cons

- Large overhead (~10x slowdown)
- Needs good tests (good in combination with fuzzing)

Debugging techiques

Instrumentation

- The basic approach to debugging is:

  Check that what we **think is true** is **actually true**.

- Narrow down the precise point at which execution deviates from our assumptions
- We can use
  - assertions: `assert` / `assert()`
  - debugging messages: `print()` / `printf()`
  - machine-readable output

Crash instrumentation example

```c
void perform_actions(struct state *s)
{
    action_a(s);
    action_b(s);
    action_c(s);
    action_d(s);
    action_e(s);
}
```

```c
void perform_actions(struct state *s)
{
    printf("Action A...\n");
    action_a(s);
    printf("Action B...\n");
    action_b(s);
    printf("Action C...\n");
    action_c(s);
    printf("Action D...\n");
    action_d(s);
    printf("Action E...\n");
    action_e(s);
    printf("Actions done.\n");
}
```

```
Action A...
Action B...
Action C...
Segmentation fault
```

$\longrightarrow$ crash in `action_c()` assuming no time-traveling UB.

Machine-readable output example

```python
def matrix_inverse(mtx):
    ...
    return result
```

```python
def matrix_inverse(mtx):
    ...

    error_matrix = mtx * result - matrix_identity()
    matrix_write(mtx, "mtx.m")
    matrix_write(result, "result.m")
    assert matrix_norm(error_matrix) < 1e-5

    return result
```

How to handle large test cases?

- assume our `matrix_inverse()` code has a bug
- we find a wrong result for a specific 2000x2000 matrix
- how do we proceed?

- we would like to instrument `matrix_inverse()` by printing the matrix at each step,
- but a 2000x2000 matrix is too large to visualize

Testcase reduction

- Input: $A \in \mathbb{R}^{n \times n}$
- Step 1: construct $B \in \mathbb{R}^{m \times m}$ by selecting an arbitrary square submatrix of A
- Step 2: test `matrix_inverse()` on B
- Step 3: if `matrix_inverse(`B`)` fails again, then $A := B$
- Step 4: go back to Step 1

Example approach:

- at first we can try removing a random half of the rows and columns of A
- if it fails repeatedly, we try to remove fewer rows and columns of A
- if it fails again, we remove a single row and column of A

This process can be automated!

Useful tool: the bisection method

Given an alphabetically-ordered list of n words

```
acre
airlock
embassy
helicopter
iron
log
olive
puddle
skeleton
trouble
virus
whey
zoology
```

determine if the word

```
pen
```

is part of the list.

Checking every word one by one until a word is alphabetically after pen: $O(n)$

Bisection: $O(\log(n))$ expected complexity

```
acre
airlock
embassy
helicopter
iron
log
olive       <-- midpoint, before pen
puddle
skeleton
trouble
virus
whey
zoology
```

```
acre
airlock
embassy
helicopter
iron
log
olive       _______________________
puddle
skeleton
trouble     <-- midpoint, after pen
virus
whey
zoology     _______________________
```

```
acre
airlock
embassy
helicopter
iron
log
olive       ________________________
puddle      <-- midpoint, after pen
skeleton    ________________________
trouble
virus
whey
zoology
```

```
acre
airlock
embassy
helicopter
iron
log
olive       ________________________   (pen not found)
puddle
skeleton
trouble
virus
whey
zoology
```

Code bisection

```c
void perform_actions(struct state *s)
{
    action_000(s);
    action_001(s);
    action_002(s);
    . . .
    action_998(s);
    action_999(s);
}
```

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    action_001(s);
    action_002(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    action_998(s);
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Segmentation fault
```

$\longrightarrow$ crash between 500 and 999 (assuming no time-traveling UB).

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    printf("Action 750...\n");
    action_750(s);
    . . .
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Action 750...
Segmentation fault
```

$\longrightarrow$ crash between 750 and 999.

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    printf("Action 750...\n");
    action_750(s);
    . . .
    printf("Action 875...\n");
    action_875(s);
    . . .
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Action 750...
Segmentation fault
```

$\longrightarrow$ crash between 750 and 875.

```c
void perform_actions(struct state *s)
{
    printf("First action...\n");
    action_000(s);
    . . .
    printf("Action 500...\n");
    action_500(s);
    . . .
    printf("Action 750...\n");
    action_750(s);
    . . .
    printf("Action 812...\n");
    action_812(s);
    . . .
    printf("Action 875...\n");
    action_875(s);
    . . .
    action_999(s);
    printf("Actions done.\n");
}
```

```
First action...
Action 500...
Action 750...
Action 812...
Segmentation fault
```

⟶ crash between 812 and 875.

Version bisection

```
git log --oneline
9e9e6fc (HEAD -> main, origin/main) Added perf version check.
ff3c21b Changed branch mispredict ratio displayed.
fd49f78 Silently ignore branch events.
85afe03 Support new perf-script brstack format with added spaces.
77f8759 Made perf script output parsing more lenient.
637f374 Version bump.
47b578b Fixed erroneous use of atime, should have been mtime.
1dadd0f Moved objdump cache to /tmp.
60a534a Added caching of objdump output.
6f3c377 Some debugging code.
b2daa9b Updated version.
```

Version bisection

```
git log --oneline
9e9e6fc (HEAD -> main, origin/main) Added perf version check. ← test this
ff3c21b Changed branch mispredict ratio displayed.
fd49f78 Silently ignore branch events.
85afe03 Support new perf-script brstack format with added spaces.
77f8759 Made perf script output parsing more lenient.
637f374 Version bump. ← test this
47b578b Fixed erroneous use of atime, should have been mtime.
1dadd0f Moved objdump cache to /tmp.
60a534a Added caching of objdump output.
6f3c377 Some debugging code.
b2daa9b Updated version. ← test this
```

Debuggers

- A debugger is a tool that allows us to run our code step-by-step (e.g. line by line)
- Between each step, we can examine
  - program output
  - program state (i.e. variables)
- Debuggers for interpreted languages are language-specific
- Debuggers for compiled languages work at the assembly level