Isolating Infections

Andreas Zeller

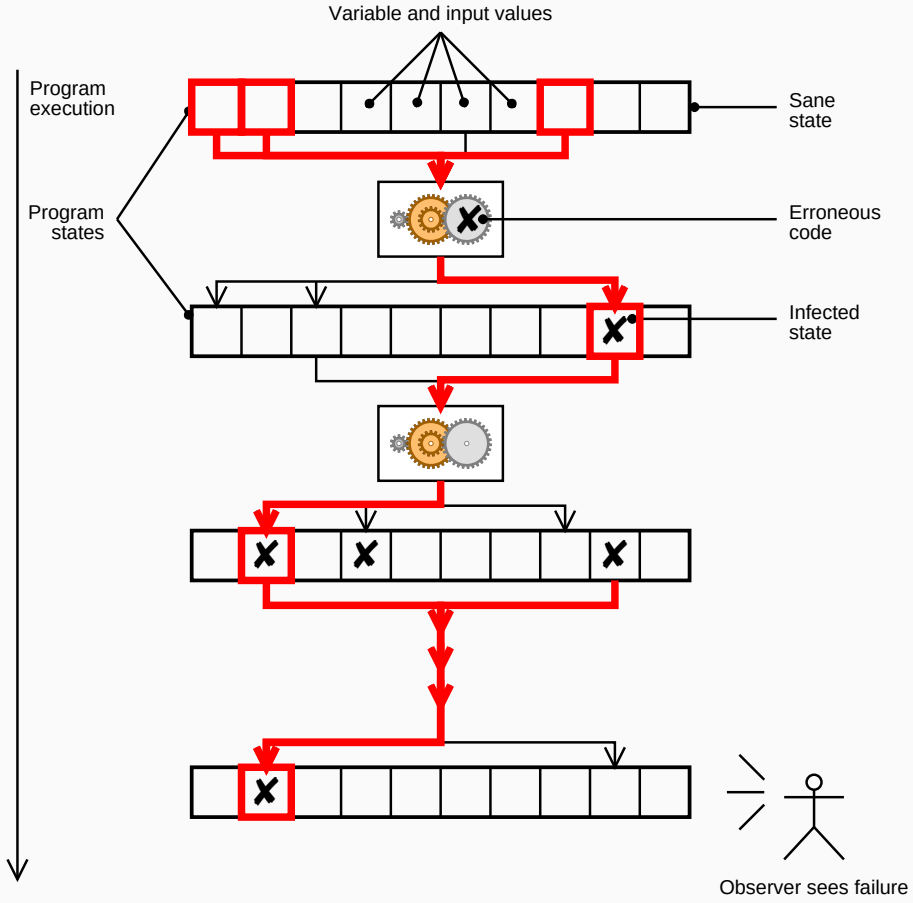
Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Isolating the Infection



Debugger Predicates

As shown before, debuggers have only *limited support for focusing:*

Type	GDB Command	Predicate
Breakpoint	`break` *location*	$PC = location$
Watchpoint	`watch` *expr*	*expr* changes
Cond. bp	`break` *location* `if` *expr*	$PC = location \wedge expr$

Does this suffice for understanding?

Answer: *yes, but only at the lowest level*

What we'd like to have

Spatial focusing (Data)

- Which variable has a value of 42?
- Which is the array x where `x[1] == null`?
- Which pointer is `null`?

Temporal focusing (Time/Code)

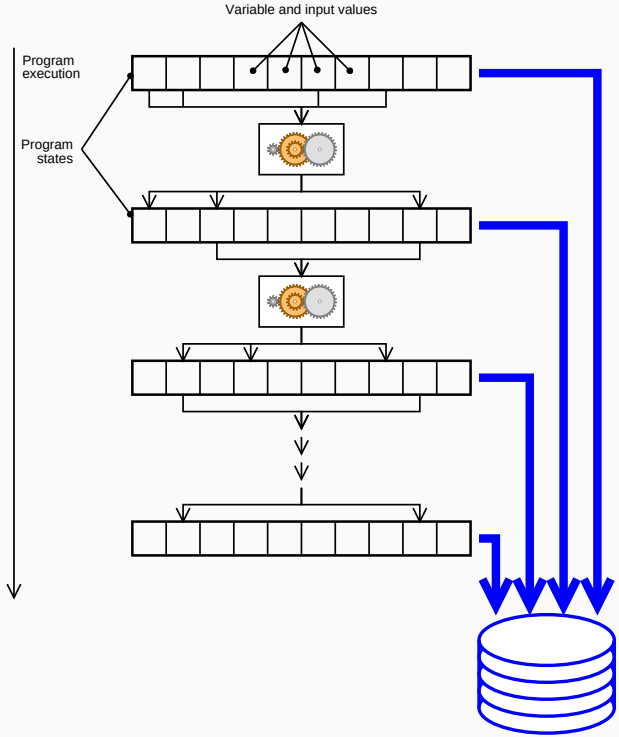
- When does `x.left` become `null`?
- When is `log_error` called by `connect`?
- When does open return `-1`?

. . . and of course, the *combination of both!*

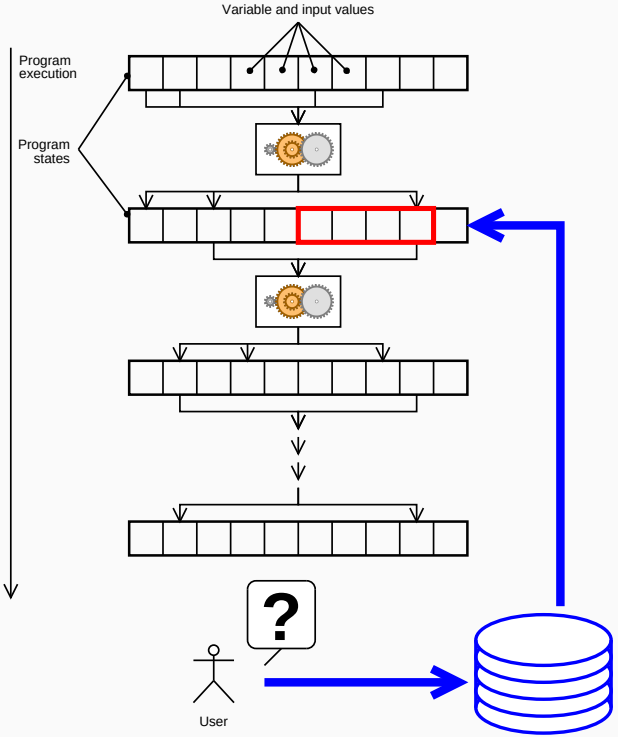
Trace Queries

Approach: *temporal and spatial queries over traces*

Store the trace...



...and _query_ the database



Realizing Trace Queries

Real databases cannot be used to store or query traces.

Reason: *too much data!*
(hundreds of thousands of variables $\times$ billions of events)

In fact, *any kind of storage* is eventually too small.

Alternative:

- Translate queries into code that is checked at run-time (via debugger or injected code)
- Execute program while query is active.







The Coca System

Coca allows for querying *events* and *data* in the trace.

Events and data are characterized by *attributes*:

Events

Attribute	Meaning
type	function/return ...
port	enter/exit
func	function name
chrono	time stamp
cdepth	call stack
line	current line
file	current file

Data

Attribute	Meaning
name	variable name
type	type
val	value
addr	address
size	size in memory
linedecl	declaration line
filedecl	declaration file







Querying Traces in Coca

To query traces, Coca uses *Prolog predicates.*

A Prolog predicate contains

- *logical variables* (starting with an upper-case letter)
- *atoms* (starting with a lower-case letter)
- *strings* and *numbers* (as usual)

Coca returns all *possible variable values* that satisfy the predicate.

Querying Examples

Spatial focusing (Data) – with `current_var`

- Which variable has a value of 42?

```
[coca] current_var(Name, val=42).
Name = x0
Name = x1
[coca] _
```

- Which is the array x where `x[1] == null`?

```
[coca] current_var(Name, val=array(-, null, ...).
Name = a1
Name = a2
[coca] _
```

All queries apply to the *current event* only.

Querying Examples (2)

Temporal focusing (Time/Code) – with fget

- When does x become null?

```
[coca] fget(func=Fun and chrono=Chr),
       current_var(x, val=null).
Fun = reset_x, Chr = 32
Fun = reset_x, Chr = 33
...
[coca] _
```

- When does open return -1?

```
[coca] fget(func=open and port=exit),
       current_var(status, val=-1).
[coca] _
```







A Debugging Session

Again, we try to understand what's going on in `shell_sort`:

```
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3 + 1;
    } while (h <= size);
    do {
        h /= 3;
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}
```







A Debugging Session (2)

- Which variables are zero?

```
[coca] fget(func=shell_sort and line=Ln),
       current_var(Name, val=0).

Name = a[2]  Ln = ⟨int i, j;⟩
Name = v     Ln = ⟨int v = a[i]⟩
Name = a[0]  Ln = ⟨a[j] = v⟩
```

- Where does a[2] come from?

```
[coca] retrace.
[coca] fget(line=Ln),
       current_var(a, val=array(-,-,0,...)).

Ln = ⟨a = malloc(...)⟩
```







Realization

Coca is built on top of an extended GNU debugger (GDB).

Queries are translated into appropriate GDB commands:

- `fget()` sets appropriate breakpoints and executes the program
- `current_var()` queries the set of variables

Drawback: General queries like

```
fget(func=shell_sort), current_var(a, val=1)
```

still require *watchpoints* or *single-stepping.*

An Alternate Approach

The Java virtual machine (JavaVM) has an interface for *monitoring* the access of object attributes:

- Interrupt execution when some attribute is read
- Interrupt execution when some attribute is written

Since the JavaVM is realized in software,

- there are no limits on the number of monitors
- there is no substantial performance loss
  - compared to debugger watchpoints
  - compared to the usual “slow” Java execution







Java Queries

The debugger of *Raimondas Lencevicius* allows querying program states for objects with specific properties:

- Which point has an x value of zero?

  ```
  Point a.  a.x = 0
  ```

- Which lexer has a token of 27?

  ```
  Lexer l; Token t.  l.token == t && t.type == 27
  ```

  This query involves a *Join* over multiple variable sets.







Java Queries (2)

Lencevicius' debugger supports static and dynamic queries:

Static Query. The query applies to *a particular event* (i.e. one moment in the execution).

Dynamic Query. The query applies to *the whole program execution*; it is checked "all the time".

This requires a number of *optimizations:*

- Installation of *JavaVM monitors* on all affected objects
- *Efficient queries* (as queries are executed with each change of an affected object)

Querying or restricting events (as in Coca) is not possible.

Assertions

So far, we have seen how to examine and query program states at various events.

The interpretation of these program states is still left to us:
We must enter appropriate queries – in retrospective

Alternative: have the computer check for sane and infected values *during exection*.

This is typically done using *assertions.*







The Assert Macro

Basic idea: A piece of code that *ensures a sane state*.

General usage:

```
assert(x);
```

- If x is true, nothing happens.
- If x is false, execution is *aborted* with a diagnostic:

```
foo.c, line 34: assertion ‘x’ failed
Aborted (core dumped)
```







Assertions for Pre- and Postconditions

Ensure that a function does the right thing

```
#include <assert.h>

void divide(int dividend, int divisor,
            int& quotient, int& remainder)
{
    assert(divisor != 0);
    // Actual computation
    ...
    assert(quotient * divisor + remainder == dividend);
}
```

Class Invariants

Goal: ensure the integrity of program state

```
 class Game {
    int n_flowers;
    Map<string, int> flower_values;
    // z.B. flower_values["rose"] == 500
    ...
public:
    bool OK() {
        assert(n_flowers >= 1);
        assert(n_flowers == flower_values.size());
        return true;
    }
}
```

Combined Assertions

A class invariant must hold at the beginning and at the end of each public class method:

```
void Game::add_flower(string name, int value)
{
    assert(OK());  // Precondition

    // Actual computation is here...

    assert(OK());  // Postcondition
}
```

Loop Invariants

Consider this simple exponentiation code:

```
long prod = 1;
for (int i = 0; i < n; i++)
    prod *= r;
```

The invariant

```
assert(prod == pow(r, i));
```

holds at the beginning of the loop, in the loop body, and after the loop.

As we see here, an assertion of non-constant complexity can *degrade the program performance*.

Therefore, assertions can be *turned off*.

Assert Definition

An actual definition of `assert(x)` looks like this:

```
#ifndef NDEBUG
#define assert(ex) \
((ex) ? 1 : (cerr << __FILE__ << ":" << __LINE__ \
                  << ": assertion ‘" #ex "’ failed\n", \
                  abort(), 0))
#else
#define assert(x) ((void) 0)
#endif
```

If NDEBUG is defined, `assert(x)` compiles to a no-op

Assertions in Production Code

In production code, assertions are typically *disabled* (by defining NDEBUG). This improves the efficiency.

As a consequence, assertions should *not be used* for checking the integrity of external data (i.e. inputs, etc.)

- The checks will be turned off in production code
- Assertion failures are very unfriendly diagnostics (to end-users, that is)







Assertions in Production Code (2)

I recommend *leaving light-weight assertions on* in production code:

- We can afford a certain amount of efficency loss for correctness sake
- Blue screens are unfriendly, but still better than wrong results

You must include a user-friendly recovery from failures, though.

Benefits of Assertions

Assertions serve three purposes:

They ensure program correctness. If the program terminates normally, its state statisfies all assertions.

They document what's going on. Assertions help in understanding the program and its assumptions.

Since they are checked at compile-time and run-time, they are consistent with the remaining program code (in contrast to comments).

They help in debugging. In contrast to log statements or debugger queries, assertions are *persistent* – they stay in the code.

There is every reason to litter your code with assertions!

Heap Assertions

In C/C++ programs, *misuse of the heap* is a frequent source for failures:

- using allocated memory after freeing it
- freeing allocated memory multiple times...

The GNU C runtime library provides *assertions* that can be enabled at run-time:

```
$ MALLOC_CHECK_=2 myprogram myargs
free() called on area that was already free'd()
Aborted (core dumped)
$ _
```

Array Assertions

In C/C++ programs, accessing an array beyond its bounds is another frequent failure source.

The *electric fence library* flags such violations automatically:

```
$ gcc -g -o sample-with-efence sample.c -lefence
$ sample-with-efence -11 14
Electric Fence 2.1
Segmentation fault (core dumped)
$ _
```







General Memory Assertions

The *Valgrind* tool checks *all* reads and writes of memory, and intercepts calls to malloc/new/free/delete.

- Use of uninitialised memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks – allocated area is not free'd
- Passing of uninitialised and/or unaddressible memory to system calls
- Mismatched use of malloc/new vs free/delete
- Some misuses of the POSIX pthreads API

A Valgrind Example


```
$ valgrind myprogram myargs
  ==25832== Invalid read of size 4
  ==25832==    at 0x8048724: BandMatrix::ReSize(int, int, int)
                  (bogon.cpp:45)
  ==25832==    by 0x80487AF: main (bogon.cpp:66)
  ==25832==    by 0x40371E5E: __libc_start_main (libc-start.c:129)
  ==25832==    by 0x80485D1: (within /home/sewardj/newmat10/bogon)
  ==25832==    Address 0xBFFFF74C is not stack'd, malloc'd or free'd
$ _
```

Process 25832 did an illegal 4-byte read of address 0xBFFFF74C.

This, as far as it can tell, is not a valid stack address, nor corresponds to any currently malloc'd or free'd blocks.

The read is happening at line 45 of bogon.cpp, called from line 66 of the same file, etc.







Valgrind and V Bits

Valgrind implements a *synthetic Intel x86 CPU*.

Every bit of data processed, stored and handled by the real CPU has, in the synthetic CPU, an associated “valid-value” bit (“V bit”).

The V bit says whether or not the accompanying bit has a legitimate value – i.e. the bit is *initialized*.

Each byte in the system therefore has a 8 V bits which follow it wherever it goes.

Example: CPU loads a word-size item (4 bytes) from memory ⇒ it also loads the corresponding 32 V bits.







How V Bits are Used

Should Valgrind flag every access to non-initialized data?

No, because many C/C++ programs routinely copy uninitialized values around in memory:

```
struct S { int x; char c; };
struct S s1, s2;
s1.x = 42;
s1.c = 'z';
s2 = s1;
```

How large is S?
Typically, 5 bytes will be initialized, 8 bytes will be copied.

How V Bits are Used (2)

Consequently, simple read accesses to not cause complaints.

Usage of uninitialized data causes complaints only if

- a value is used to generate a memory address
- a control flow decision needs to be made
- a value is passed to a system call (i.e. printed, etc.)

After a complaint, the value is regarded as well-defined (to avoid long chains of error messages).







Valgrind and A Bits

In addition to V bits, all bytes in memory have an associated “valid-address” bit (“A bit”).

The A bit indicates whether or not the program can legitimately read or write that location.

Every time the program reads or writes memory, Valgrind checks the A bits associated with the address. If any of them indicate an invalid address, an error is emitted.

Please distinguish:

- *V bits* check whether *values* are valid,
- *A bits* check whether *addresses* are valid.

How A Bits are Used

- When the program starts, all the global data areas are marked as accessible.
- malloc/new sets the A bits for the exactly the area allocated.
- Local variables are marked accessible on function entry and inaccessible on exit (by tracking the stack pointer)
- Some system calls (such as `mmap()`) cause A bits to changed appropriately.







Valgrind Benefits and Drawbacks

Benefits

- Quick detection of uninitialized memory use
- Quick detection of heap misuse

Drawbacks

- Huge increase in code size (during execution)
- Large increase in memory size (during execution)
- Some increase in execution time







Using all these Tools

Logs, debuggers, queries, assertions, and memory management tools all allow to access and assess the state.

We must still isolate the infection, though!

For this purpose, we have two general strategies:

Isolating a Specific State. Spatial focusing—across the program state.

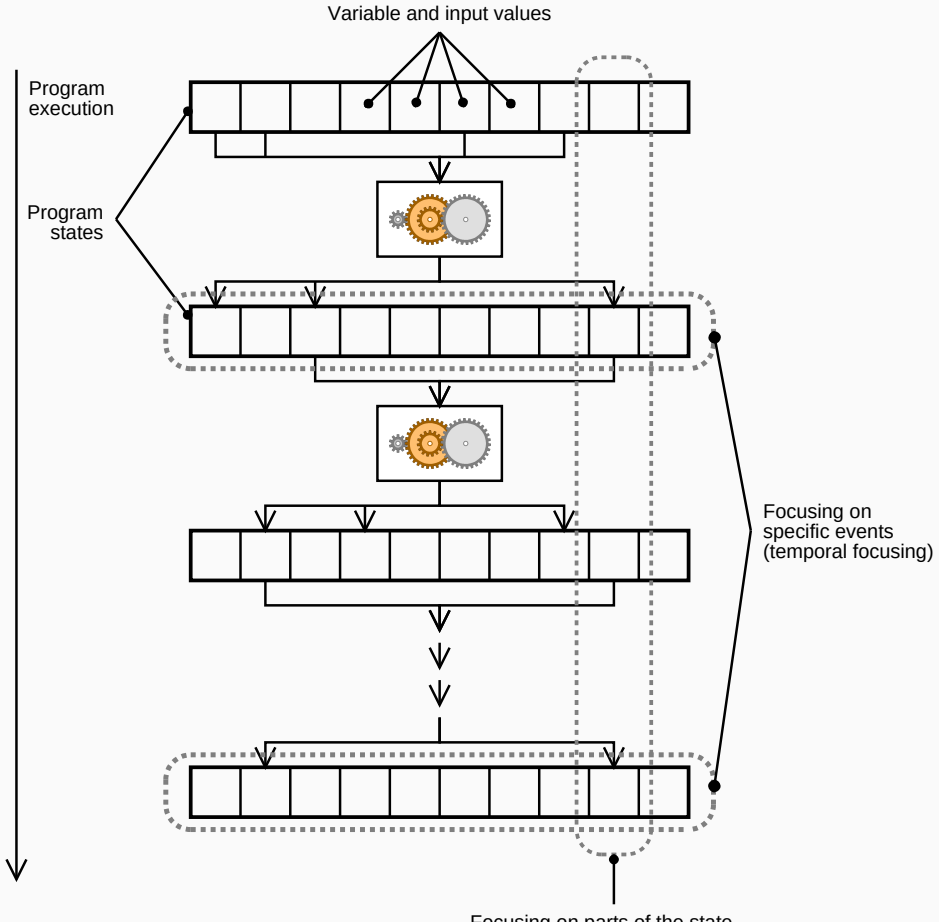
Isolating the Infection. Temporal focusing—across the program execution.







Focusing Techniques



Spatial Focusing

Basic idea: Separate *sane state* (= as intended) from *infected state* (= not as intended)

- Use logging (or a debugger) to access state
- Use *assertions* (or likewise logging/debugger techniques) to separate sane from infected state







Temporal Focusing

Basic idea: identify the *moment in time* where the state becomes infected

- Use logging (or a debugger) to access execution
- Use *assertions* (or likewise logging/debugger techniques) to separate sane from infected state
- Use binary search to find out the moment in time where the state first became infected
- Trace back possible *origins* of the infection –
  To be addressed in remainder of the course!







Concepts

- Trace queries allow *high-level interaction* with a debugger to understand what's going on
- *Assertions* are among the most useful programming tools – both expressive and persistent
- *Valgrind* and similar tools introduce assertions on the general program state
- *Spatial focusing* means to separate the state into *sane* (= as intended) and *infected*
- *Temporal focusing* means to isolate the moment in time where the infection occurs
- *All this must be (and can be) automated!*







References

- M. Ducassé: *Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events*. Proc. ICSE'99 (International Conference on Software Engineering), Los Angeles, May 1999, ACM Press.

  Preliminary version at http://www.inria.fr/RRRT/RR-3489.html

- R. Lencevicius: *Advanced Debugging Methods*. Kluwer Academic Publishers, 2000.

- Valgrind, http://developer.kde.org/~sewardj/