Automatically Detecting Variability Bugs Through Hybrid Control and Data Flow Analysis

Kelly Kaoudis, Henrik Brodin, Evan Sultanik

LangSec Workshop at IEEE S&P, 25 May 2023
Variability bug: run-to-run (over the same input) software execution divergence due to build configuration or environment
Goal: detect \textit{and} correctly diagnose runtime C and C++ variability bugs (with multiple causes)
“That should be easy to figure out with UBSan, right?”

- UBSan helps detect some but not all types of UB
- UBSan cannot detect all types of variability bug
- Detection $\neq$ correct diagnosis
- Tells you there is a bug (detection) and roughly where, but does not help with further diagnosis actions
- May not help at all, due to build configuration
Listing 3 A bitwise left shift operation in the following toy program results in undefined behavior if shift is greater than the data type’s max bitwise capacity. Undefined behavior on line 12 occurs dependent on user input and build configuration.

```c
#include <cassert>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if (argc > 1) {
        int shift = std::stoi(argv[1]);
        assert(shift > 0 && shift < 32);
        return 0xff << shift;
    } else {
        return 0;
    }
}
```
Listing 3 A bitwise left shift operation in the following toy program results in undefined behavior if `shift` is greater than the data type’s max bitwise capacity. Undefined behavior on line 12 occurs dependent on user input and build configuration.

```cpp
#include <cassert>
#include <cstdlib>

int main(int argc, char* argv[]) {
  if (argc > 1) {
    int shift = std::atoi(argv[1]);
    assert(shift > 0 && shift < 32);
    return 0x0ff << shift;
  } else {
    return 0;
  }
}
```
$ clang++ -Wall -o toy0 -std=c++20 -O0 toy.cpp
$ ./toy0 63
toy0: toy.cpp:11: int main(int, char **):
Assertion `shift > 0 && shift < 32'
failed.
[1] 500225 abort ./toy0 63

$ clang++ -Wall -o toy0 -std=c++20 -O0 -fsanitize=undefined toy.cpp
$ ./toy0 63
toy0: toy.cpp:11: int main(int, char **):
Assertion `shift > 0 && shift < 32'
failed.
[1] 500413 abort ./toy0 63

Listing 3 A bitwise left shift operation in the following toy program results in undefined behavior if shift is greater than the data type’s max bitwise capacity. Undefined behavior on line 12 occurs dependent on user input and build configuration.

```cpp
#include <cassert>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if(argc > 1) {
        int shift = std::atoi(argv[1]);
        assert(shift > 0 && shift < 32);
        return 0xff << shift;
    } else {
        return 0;
    }
}```
Listing 3 A bitwise left shift operation in the following toy program results in undefined behavior if `shift` is greater than the data type’s max bitwise capacity. Undefined behavior on line 12 occurs dependent on user input and build configuration.

```cpp
#include <cassert>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if (argc > 1) {
        int shift = std::atoi(argv[1]);
        assert(shift > 0 && shift < 32);
        return 0xff << shift;
    } else {
        return 0;
    }
}
```
Listing 3 A bitwise left shift operation in the following toy program results in undefined behavior if shift is greater than the data type's max bitwise capacity. Undefined behavior on line 12 occurs dependent on user input and build configuration.

```c
#include <cassert>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if(argc > 1) {
        int shift = std::atoi(argv[1]);
        assert(shift > 0 && shift < 32);
        return 0x0f0 << shift;
    } else {
        return 0;
    }
}
```

↓ This would be awesome ↓

<table>
<thead>
<tr>
<th>DEBUG</th>
<th>fn</th>
<th>PROD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{0, 1, 2, 3}</td>
<td>f0</td>
</tr>
<tr>
<td>2</td>
<td>{8, 9, 10, 11}</td>
<td>f1</td>
</tr>
<tr>
<td>3</td>
<td>{8, 9, 10, 11}</td>
<td>f2</td>
</tr>
<tr>
<td>4</td>
<td>{8, 9, 10, 11}</td>
<td>f2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>f0</td>
</tr>
</tbody>
</table>
Listing 3 A bitwise left shift operation in the following toy program results in undefined behavior if shift is greater than the data type’s max bitwise capacity. Undefined behavior on line 12 occurs dependent on user input and build configuration.

```c
#include <cassert>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if(argc > 1) {
        int shift = std::atoi(argv[1]);
        if(shift > 0 && shift < 32)
            return 0x0f << shift;
        else {
            assert(shift > 0 && shift < 32);
            return 0;
        }
    }
    return 0;
}
```

↓ This would be *really* awesome ↓

<table>
<thead>
<tr>
<th>DEBUG</th>
<th>fn</th>
<th>PROD</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0,1,2,3}</td>
<td>f0</td>
<td>{0,1,2,3}</td>
</tr>
<tr>
<td>{8,9,10,11}</td>
<td>f1</td>
<td>{8,9,10,11}</td>
</tr>
<tr>
<td>{8,9,10,11}</td>
<td>f2</td>
<td>{8,9,10,11}</td>
</tr>
<tr>
<td>{8,9,10,11}</td>
<td>f2</td>
<td>{8,9,10,11}</td>
</tr>
<tr>
<td>{8,9,10,11}</td>
<td>f0</td>
<td>{8,9,10,11}</td>
</tr>
</tbody>
</table>
This would be *really* awesome

<table>
<thead>
<tr>
<th>DEBUG</th>
<th>fn</th>
<th>PROD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 1, 2, 3</td>
<td>f0</td>
<td>{0, 1, 2, 3}</td>
</tr>
<tr>
<td>8, 9, 10, 11</td>
<td>f1</td>
<td>{8, 9, 10, 11}</td>
</tr>
<tr>
<td>8, 9, 10, 11</td>
<td>f2</td>
<td>{8, 9, 10, 11}</td>
</tr>
<tr>
<td>8, 9, 10, 11</td>
<td>f2</td>
<td></td>
</tr>
<tr>
<td>8, 9, 10, 11</td>
<td>f0</td>
<td>{8, 9, 10, 11}</td>
</tr>
</tbody>
</table>

```
Listing 3 A bitwise left shift operation in the following toy program results in undefined behavior if shift is greater than the data type’s max bitwise capacity. Undefined behavior on line 12 occurs dependent on user input and build configuration.

# if defined(PRODUCTION)
#define NDEBUG
#endif

#include <cassert>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if(argc > 1) {
        int shift = std::atoi(argv[1]);
        assert(shift > 0 && shift < 32);
        return 0xff << shift;
    } else {
        return 0;
    }
}
```
Challenge #1
How to successfully detect?
Parser differential basics

Input

Binary A → Output A

Binary B → Output B

SAME
Program output differential basics

Input

Binary A

Output A

Binary B

Output B

SAME
Challenge #2
Can we “rewind” execution (enough) to **correctly diagnose** the contributing factors?
PolyTracker’s Data Flow Representation
Avoid FPs and reduce extra detail

- Start from too much, reduce to helpful representation
- Control flow (function, BB identifiers, ...) as waypoints
- Label all waypoints by nearest function identifier, $f()_{id}$
- When data flow passes through a waypoint, create a control-affecting data flow log entry mapped to $f()_{id}$
- Map $f()_{id}$s to human-readable program symbols
Program representation:
Hybrid control and data flow
Control-affecting data flow
Control-Affecting Data Flow

Data Flow

Waypoint (Control Flow) Identifiers

\[ f_0, f_1, f_2, f_3, f_4, f_5, f_6, \ldots \]
Method summary

- For each program variant, build the program representation
  - 2x llvm dynamic instrumentation passes
    - Before front end optimization (new!)
    - After front end optimization (PolyTracker original)
  - When data flow passes through a waypoint $f_{id}$, map $f_{id}$ to parent input byte(s) $b_1...b_n$
  - Can check instrumentation is transparent!!
- Compare $f_{id}$s at matching input byte sets $b_1...b_n$
- Map opaque $f_{id}$s to de-mangled symbols (from the pre-opt llvm pass)
Preliminary Evaluation
Example: Nitro

- Reference parser for public NITF specifications
  - **NITF**: visual data (mp4, jpeg, fingerprints, ...) + text (captions, ...) in a binary file format package
  - Implements the mutually incompatible MIL-STD-2500\{A, B, C\}
  - Bespoke stdlib fn implementations baked into build system
- Small known-valid and known-invalid input corpus (148 NITFs) to start with
- Found and diagnosed 3 bugs in Nitro; more to come!
<table>
<thead>
<tr>
<th>Debug Offsets</th>
<th>Function</th>
<th>Release Offsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>{360, 361, 362}</td>
<td>DBG: int GCS::details::narrow2(...) != REL: showImages(...)</td>
<td>{360, 361, 362}</td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>nitf::INVALID_NUM_SEGMENTS(unsigned int)</td>
<td></td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>int GCS::details::narrow1_&lt;int, unsigned int&gt; (int, unsigned int)</td>
<td></td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>int GCS::details::narrow&lt;int, unsigned int&gt; (int, unsigned int)</td>
<td></td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>int GCS::details::narrow2_&lt;int, unsigned int&gt; (int, unsigned int)</td>
<td></td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>nitf::Record::getNumImages () const</td>
<td></td>
</tr>
</tbody>
</table>

Functions optimized out of the Release build

---

<table>
<thead>
<tr>
<th>Debug Offsets</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>{717}</td>
<td>showImages(nitf::Record const&amp;)</td>
</tr>
<tr>
<td>{737}</td>
<td></td>
</tr>
<tr>
<td>{745}</td>
<td></td>
</tr>
<tr>
<td>{753}</td>
<td></td>
</tr>
<tr>
<td>{756}</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Debug Offsets</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>{764}</td>
<td>std::__1::basic_stringbuf&lt;&gt;::__overflow(int)</td>
</tr>
<tr>
<td>{772}</td>
<td></td>
</tr>
<tr>
<td>{774}</td>
<td></td>
</tr>
<tr>
<td>{775}</td>
<td></td>
</tr>
<tr>
<td>{777}</td>
<td></td>
</tr>
<tr>
<td>Debug Offsets</td>
<td>Function</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>DBG: int Gel::details::narrow2(...) != REL::showImages(...)</td>
</tr>
<tr>
<td></td>
<td>nif::INVALID_NUM_SEGMENTS(unsigned int)</td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>int Gel::details::narrow1&lt;int, unsigned int&gt;(int, unsigned int)</td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>int Gel::details::narrow&lt;int, unsigned int&gt;(int, unsigned int)</td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>int Gel::details::narrow2&lt;int, unsigned int&gt;(int, unsigned int)</td>
</tr>
<tr>
<td>{360, 361, 362}</td>
<td>nif::Record::getNumImages() const</td>
</tr>
</tbody>
</table>

_functions optimized out of the Release build_

| {717} | showImages(nif::Record const&) |
| {717} | |
| {737} | |
| {745} | |
| {745} | |
| {753} | |
| {753} | |
| {756} | |

Debug trace diverges here

std::__1::basic_stringbuf<>::overflow(int)
Result: Nitro

- Last byte offset affecting control flow before divergence: 756 ‘Y’
- Nearest identifier: `showImages(nitf::Record const&)`
- Last thing Nitro runs: `TRY_SHOW(imsub.imageRepresentation());`
- Manual (for now) mapping back of byte offset to NITF specification fields: **IREP** (Image Representation)
- Field value in input: **YCbCr601**
Future directions :D

- Evaluate different types of binary file or image format parsers
- Better differential metrics - graph similarity clustering
- More experiments evaluating Nitro, too
- Integrate other Trail of Bits tools into our analysis
  - Graphtage for improved control-affecting data flow matching up
  - Polyfile for mapping back last related input byte offset to spec
  - Maybe: run PolyTracker over an MLIR (from VAST) instead of bitcode?
- Integrate our analysis into Galois’ Format Analysis Workbench (FAW)?
- What else would you like to see? We are open to ideas
Summary

- Learned the limits of existing compiler-rt sanitizers!
- New program representation enabling variability bug analysis!
- We found that following the control flow input bytes exercised helps trace back to the root(s) of a divergence!
- Detected and diagnosed variability bugs in real software!

Thank you!

Special thanks to our shepherd Sergey, our awesome reviewers, and our colleagues Nathan, Marek, Peter, Dominik, Lisa, Jay, and Michael.

Code: github.com/trailofbits/polytracker
Contact: kelly.kaoudis@trailofbits.com, henrik.brodin@trailofbits.com, evan.sultanik@trailofbits.com