# 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 *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 != 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.

```
1  #if defined(PRODUCTION)
2    #define NDEBUG
3  #endif
4
5  #include <cassert>
6  #include <cstdlib>
7
8  int main(int argc, char* argv[]) {
9      if(argc > 1) {
10         int shift = std::atoi(argv[1]);
11         assert(shift > 0 && shift < 32);
12         return 0xff << shift;
13     } else {
14         return 0;
15     }
16 }
```

**PROD**

```
$ clang++ -Wall -o toy -std=c++20
-DPRODUCTION -O2 toy.cpp
$ ./toy 63
$ echo $?
0
```

**DEBUG**

```
$ 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
```

**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.

```
1  #if defined(PRODUCTION)
2    #define NDEBUG
3  #endif
4
5  #include <cassert>
6  #include <cstdlib>
7
8  int main(int argc, char* argv[]) {
9      if(argc > 1) {
10         int shift = std::atoi(argv[1]);
11         assert(shift > 0 && shift < 32);
12         return 0xff << shift;
13     } else {
14         return 0;
15     }
16 }
```

**DEBUG**

```
$ 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
```

**UBSAN**

```
$ 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.

```
1  #if defined(PRODUCTION)
2    #define NDEBUG
3  #endif
4
5  #include <cassert>
6  #include <cstdlib>
7
8  int main(int argc, char* argv[]) {
9      if(argc > 1) {
10         int shift = std::atoi(argv[1]);
11         assert(shift > 0 && shift < 32);
12         return 0xff << shift;
13     } else {
14         return 0;
15     }
16 }
```

DEBUG

```
$ 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
```

PROD

```
$ clang++ -Wall -o toy -std=c++20 -O2
-DPRODUCTION -fsanitize=undefined toy.cpp
$ ./toy 63
toy.cpp:12:21: runtime error: shift
exponent 63 is too large for 32-bit type
'int'
SUMMARY: UndefinedBehaviorSanitizer:
undefined-behavior toy.cpp:12:21
```

**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.

```
1  #if defined(PRODUCTION)
2    #define NDEBUG
3  #endif
4
5  #include <cassert>
6  #include <cstdlib>
7
8  int main(int argc, char* argv[]) {
9      if(argc > 1) {
10         int shift = std::atoi(argv[1]);
11         assert(shift > 0 && shift < 32);
12         return 0xff << shift;
13     } else {
14         return 0;
15     }
16 }
```

**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.

```
1  #if defined(PRODUCTION)
2    #define NDEBUG
3  #endif
4
5  #include <cassert>
6  #include <cstdlib>
7
8  int main(int argc, char* argv[]) {
9      if(argc > 1) {
10         int shift = std::atoi(argv[1]);
11         assert(shift > 0 && shift < 32);
12         return 0xff << shift;
13     } else {
14         return 0;
15     }
16 }
```

## ↓ This would be awesome ↓

```
       DEBUG      | fn | PROD
1| {0,1,2,3}      | f0 | {0,1,2,3}
2| {8,9,10,11}    | f1 | {8,9,10,11}
3| {8,9,10,11}    | f2 |
4| {8,9,10,11}    | f2 |
5|                | f0 | {8,9,10,11}
```

**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.

```
1  #if defined(PRODUCTION)
2    #define NDEBUG
3  #endif
4
5  #include <cassert>
6  #include <cstdlib>
7
8  int main(int argc, char* argv[]) {
9      if(argc > 1) {
10         int shift = std::atoi(argv[1]);
11         assert(shift > 0 && shift < 32);
12         return 0xff << shift;
13     } else {
14         return 0;
15     }
16 }
```

```
        DEBUG     | fn | PROD
1| {0,1,2,3}      | f0 | {0,1,2,3}
2| {8,9,10,11}    | f1 | {8,9,10,11}
3| {8,9,10,11}    | f2 |
4| {8,9,10,11}    | f2 |
5|                | f0 | {8,9,10,11}
```

↓ This would be *really* awesome ↓

```
        DEBUG     |        function symbol           | PROD
1| {0,1,2,3}      | int main(int argc, char* argv[]) | {0,1,2,3}
2| {8,9,10,11}    | int std::atoi(const char* str)   | {8,9,10,11}
3| {8,9,10,11}    | void assert(int expression)      |
4| {8,9,10,11}    | void assert(int expression)      |
5|                | int main(int argc, char* argv[]) | {8,9,10,11}
```

./toy 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.

```
1  #if defined(PRODUCTION)
2    #define NDEBUG
3  #endif
4
5  #include <cassert>
6  #include <cstdlib>
7
8  int main(int argc, char* argv[]) {
9      if(argc > 1) {
10         int shift = std::atoi(argv[1]);
11         assert(shift > 0 && shift < 32);
12         return 0xff << shift;
13     } else {
14         return 0;
15     }
16 }
```

?

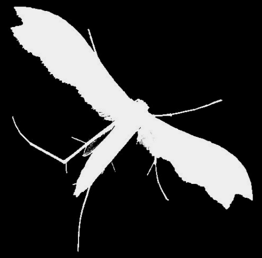
| | DEBUG | fn | PROD |
|---|---|---|---|
| 1 | {0,1,2,3} | f0 | {0,1,2,3} |
| 2 | {8,9,10,11} | f1 | {8,9,10,11} |
| 3 | {8,9,10,11} | f2 | |
| 4 | {8,9,10,11} | f2 | |
| 5 | | f0 | {8,9,10,11} |

?

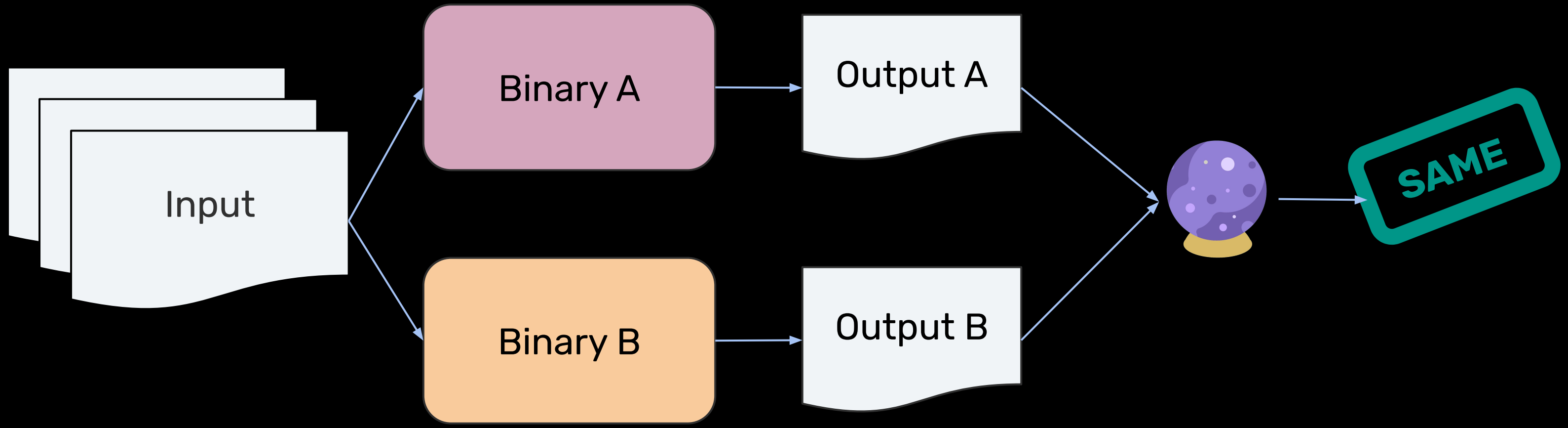
↓ This would be *really* awesome ↓

| | DEBUG | function symbol | PROD |
|---|---|---|---|
| 1 | {0,1,2,3} | int main(int argc, char* argv[]) | {0,1,2,3} |
| 2 | {8,9,10,11} | int std::atoi(const char* str) | {8,9,10,11} |
| 3 | {8,9,10,11} | void assert(int expression) | |
| 4 | {8,9,10,11} | void assert(int expression) | |
| 5 | | int main(int argc, char* argv[]) | {8,9,10,11} |

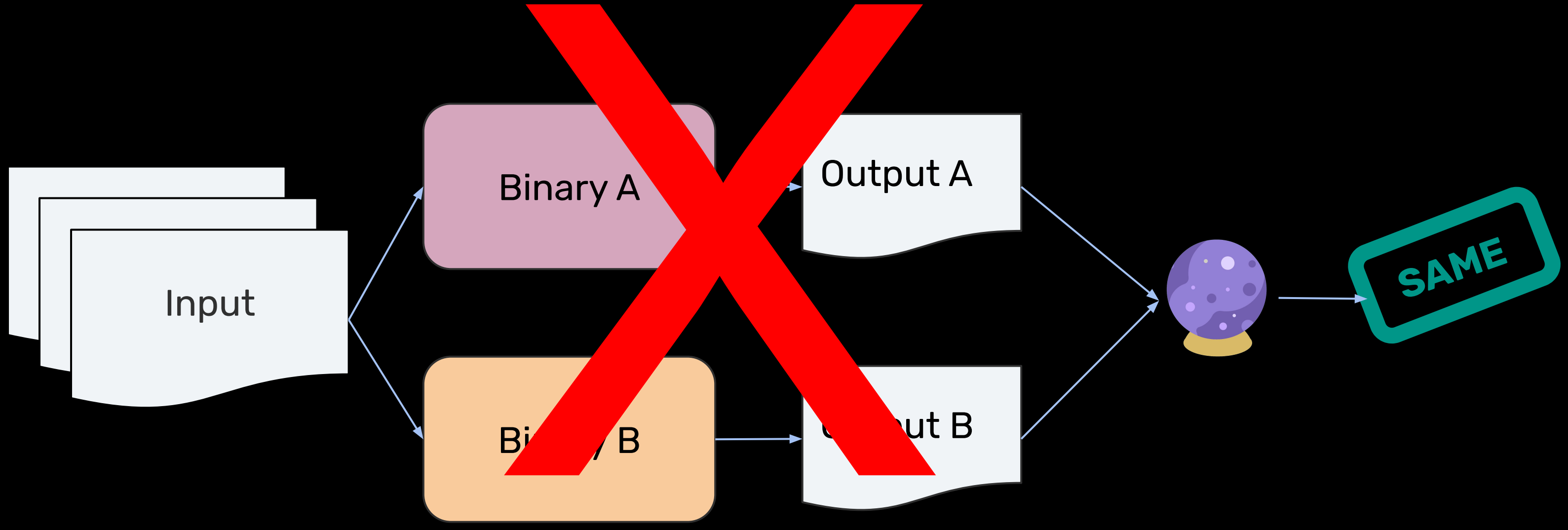
## Challenge #1
## How to successfully detect?

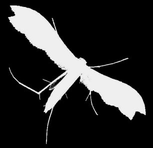


# Parser differential basics



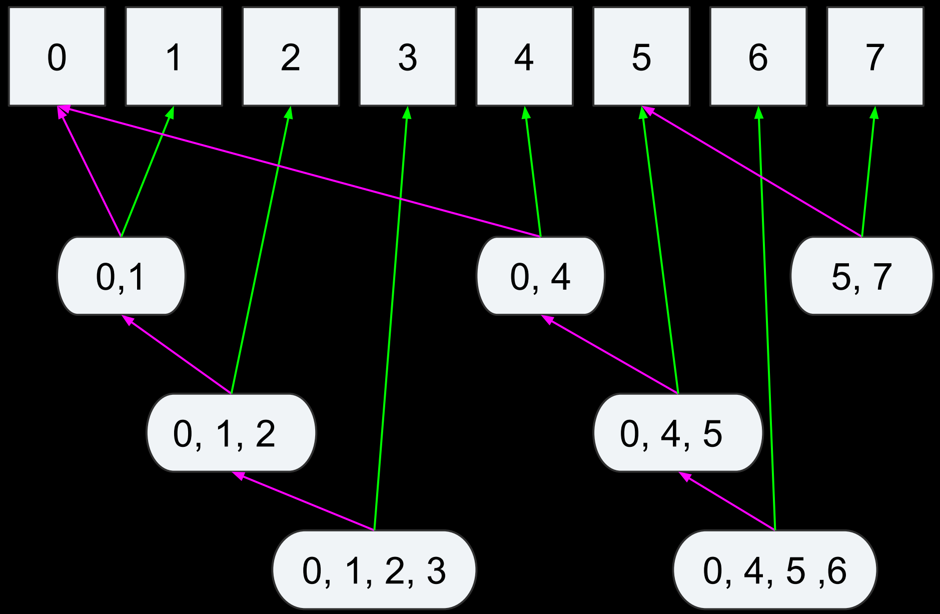
# ~~Program output~~ differential basics





**Challenge #2**
**Can we "rewind" execution (enough) to correctly diagnose the contributing factors?**

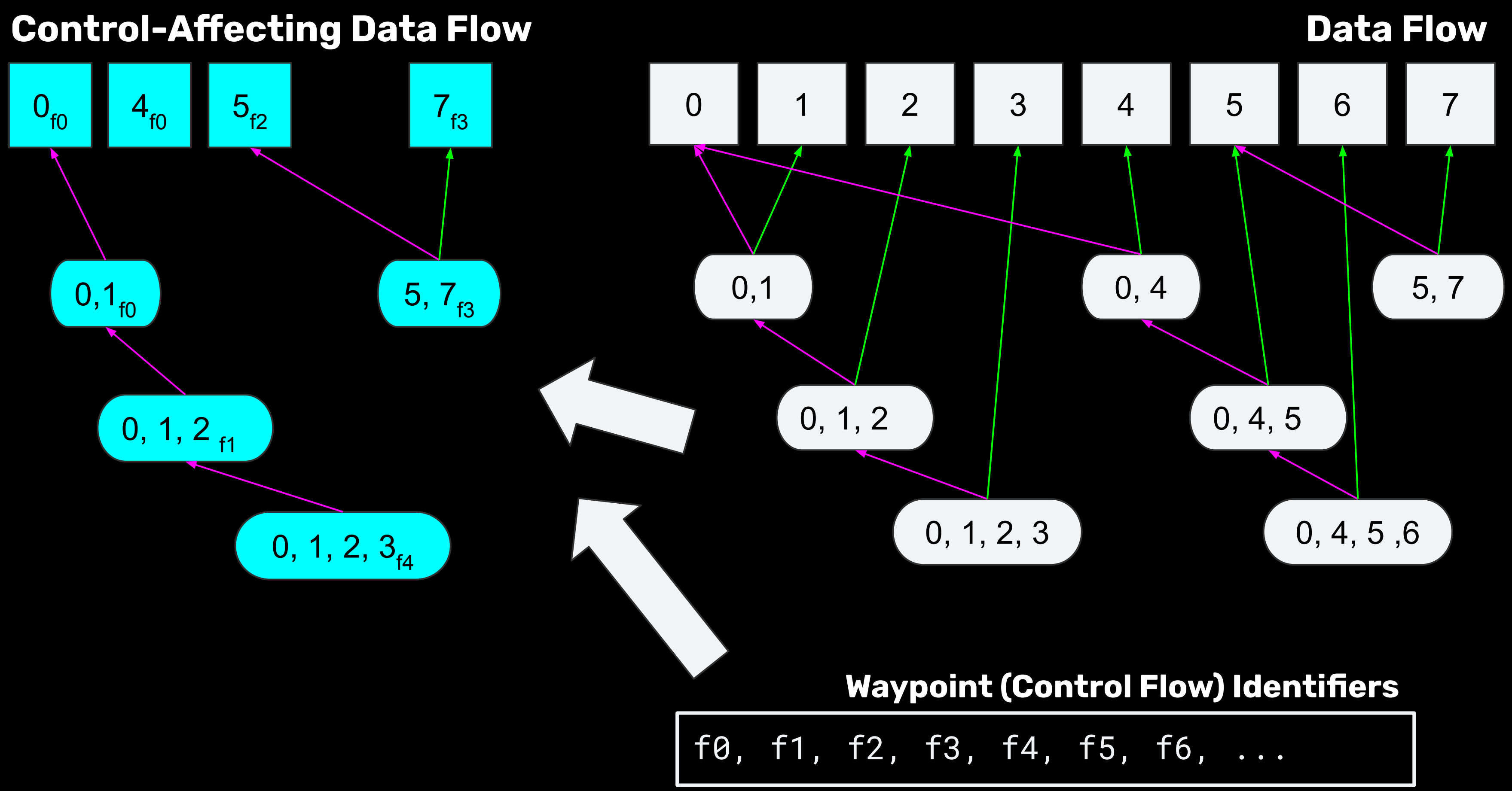
# PolyTracker’s Data Flow Representation



# 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
0 f0
4 f0
5 f2
7 f3
0,1 f0
5, 7 f3
0, 1, 2 f1
0, 1, 2, 3 f4
Data Flow
0
1
2
3
4
5
6
7
0,1
0, 4
5, 7
0, 1, 2
0, 4, 5
0, 1, 2, 3
0, 4, 5 ,6
Waypoint (Control Flow) Identifiers
f0, f1, f2, f3, f4, f5, f6, ...

# 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) $b_i \ldots b_n$
  - Can check instrumentation is transparent!!
- Compare $f()_{id}$s at matching input byte sets $b_i \ldots b_n$
- 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!

direction of execution

| Debug Offsets | Function | Release Offsets |
| --- | --- | --- |
| ⋮ | ⋮ | ⋮ |
| {360, 361, 362} | DBG: int Gsl::details::narrow2_(...) != REL: showImages(...) | {360, 361, 362} |
| {360, 361, 362} | nitf::INVALID_NUM_SEGMENTS(unsigned int) | Functions optimized out of the Release build |
| {360, 361, 362} | int Gsl::details::narrow1_<int, unsigned int>(int, unsigned int) | |
| {360, 361, 362} | int Gsl::details::narrow<int, unsigned int>(int, unsigned int) | |
| {360, 361, 362} | int Gsl::details::narrow2_<int, unsigned int>(int, unsigned int) | |
| {360, 361, 362} | nitf::Record::getNumImages() const | |
| ⋮ | ⋮ | ⋮ |
| {717} | showImages(nitf::Record const&) | {717} |
| {717} | | {717} |
| {737} | | {737} |
| {737} | | {737} |
| {745} | | {745} |
| {745} | | {745} |
| {753} | | {753} |
| {753} | | {753} |
| {756} | | {756} |
| {756} | | {756} |
| | std::__1::basic_stringbuf<...>::overflow(int) | {764} |
| | | {764} |
| | | {772} |
| | | {772} |
| | | {774} |
| | | {774} |
| | | {775} |
| | | {775} |
| | | {777} |
| | ⋮ | ⋮ |

| Debug Offsets | Function | Release Offsets |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| {360, 361, 362} | DBG: int Gsl::details::narrow2_(...) != REL: showImages(...) | {360, 361, 362} |
| {360, 361, 362} | nitf::INVALID_NUM_SEGMENTS(unsigned int) | Functions optimized out of the Release build |
| {360, 361, 362} | int Gsl::details::narrow1_<int, unsigned int>(int, unsigned int) | |
| {360, 361, 362} | int Gsl::details::narrow<int, unsigned int>(int, unsigned int) | |
| {360, 361, 362} | int Gsl::details::narrow2_<int, unsigned int>(int, unsigned int) | |
| {360, 361, 362} | nitf::Record::getNumImages() const | |
| ⋮ | ⋮ | ⋮ |
| {717} | showImages(nitf::Record const&) | {717} |
| {717} | | {717} |
| {737} | | {737} |
| {737} | | {737} |
| {745} | | {745} |
| {745} | | {745} |
| {753} | | {753} |
| {753} | | {753} |
| {756} | | {756} |
| {756} | | {756} |
| X Debug trace diverges here | std::__1::basic_stringbuf<...>::overflow(int) | {764} |
| | | {764} |
| | | {772} |
| | | {772} |
| | | {774} |
| | | {774} |
| | | {775} |
| | | {775} |
| | | {777} |
| | ⋮ | ⋮ |

direction of execution

# 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`

**Listing 5** Lines 68–72 of `ImageSubheader.hpp` in the Nitro codebase as of git commit `466534fd`. The `ImageRepresentation` enumeration is missing an entry for `YCbCr601`.

```
enum class ImageRepresentation {
  MONO,
  RGB,
  RGB_LUT,
  MULTI,
  NODISPLY
};

NITF_ENUM_define_string_to_enum_begin(
ImageRepresentation
)
// need to do this manually because of "RGB/LUT"
{ "MONO", ImageRepresentation::MONO },
{ "RGB", ImageRepresentation::RGB },
{ "RGB/LUT", ImageRepresentation::RGB_LUT },
{ "MULTI", ImageRepresentation::MULTI },
{ "NODISPLY", ImageRepresentation::NODISPLY }
NITF_ENUM_define_string_to_end
```

# 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 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**