Automatically Detecting Variability Bugs Through Hybrid Control and Data Flow Analysis

Kelly Kaoudis
Trail of Bits
New York, USA
kelly.kaoudis@trailofbits.com

Henrik Brodin
Trail of Bits
New York, USA
henrik.brodin@trailofbits.com

Evan Sultanik
Trail of Bits
New York, USA
evan.sultanik@trailofbits.com

***Abstract*—Subtle bugs that only manifest in certain software configurations are notoriously difficult to correctly trace. Sometimes called Heisenbugs, these runtime variability flaws can result from invoking undefined behavior in languages like C and C++, or from compiler flaws. In this paper, we present a novel analysis technique for detecting and correctly diagnosing variability bugs' impact on a program through comparing *control-affecting data flow* across differently compiled program variants. Our *UBet* prototype dynamically derives a runtime control flow trace while tracing universal data flow for a program processing a given input, operating at a level of tracing completeness not achievable through similar dynamic instrumentation means. Sans compiler bugs or undefined behavior, every compile-time program configuration (*i.e.*, compiler flags vary) should be semantically equivalent. Thus, any input for which a program variant produces inconsistent output indicates a variability bug. Our analysis compares control-affecting data flow traces from disagreeing program version runs to identify related input bytes and determine where in the program the processing variability originates. Though we initially demonstrate our technique on C++ variability bugs in Nitro, the American Department of Defense NITF (National Imagery Transmission Format) reference implementation parser, our approach applies equally to other programs and input types beyond NITF parsers. Finally, we sketch a path toward completing this work and refining our analysis, including evaluating parsers of other input formats.**

***Index Terms*—application security, dynamic analysis, dynamic tainting, validation, testing and debugging**

Introduction

Environmental, configuration, and compilation variance can cause unexpected deviations in program behavior. Such *variability bugs* encompass the colloquial class of "Heisenbugs" [1]—so called because they frustratingly seem to vanish when one attempts to reproduce them. Parser developers may also implement subtly different variations on an input file specification, which can cause inconsistent behavior between parsers (known as *file format schizophrenia* [2]), but this problem is not our focus. In this work we concentrate on the problem of tracing, triaging, and explaining variability bugs in C and C++ programs. We focus on bugs that certain compiler and build configurations exacerbate, or that only manifest in certain configurations [3], relating to programming error(s) like invoking undefined behavior, or to compiler-level flaws. Identifying and eliminating variability bugs reduces program complexity, which provides security value by shrinking potential attack surface.

Today, there are few options for debugging variability bugs which do not require preexisting background knowledge of the program source or additional domains. Consumer tools like UBSan [4] and similar compiler-supported sanitizers help detect subcategories of variability bugs [5] without specialized knowledge, but even if all such sanitizers could instrument the same binary simultaneously, they cannot diagnose all possible causes of runtime program variability [3]. To apply DFSan (the most comprehensive, thus most likely to expose relevant program details, of such sanitizers) directly, the programmer must already have identified potential areas of interest in the codebase, since DFSan does not track enough of a program's data flow [6] to capture diagnostically significant information about the bug. Other common methodologies and tools (*e.g.*, fuzzing [7], binary disassembly and manual assembly snippet comparison, dynamically stepping through the program [8], or applying other dynamic data flow analysis tools [9]) also require significant domain expertise, and can be unacceptably slow. Much cutting-edge dynamic dataflow analysis work [10]–[13] focuses on creating tools for production deployment (a non-goal, as we are most interested in aiding offline bug reproduction and diagnosis), and primarily evaluates them against more consistently appearing bugs, rather than Heisenbugs.

We hypothesize variability bugs could become uniquely visible when diffing control flows extracted from sufficiently comprehensive runtime dataflow records of input bytes' paths through a program to output, which we term *control-affecting data flow*. Our prototype, *UBet*, applies differential analysis over control-affecting data flow traces obtained from program configuration variants to confirm and diagnose runtime variability bugs. UBet also can surface the extent of effects of compiler bugs (such as incorrect branch merging during optimization) at runtime.

The main challenges we face relate to variability bug detection and correct diagnosis. *Detection* means we must know how a parser should actually evaluate a particular input [14], [15]. *Correct diagnosis* means we must filter benign differences between runtime traces from differently compiled program variants (*e.g.*, optimization pass function inlining or basic block reordering), and also be able to trace back to the start of the buggy behavior in source. Our contributions are the following:

- a unique program-run representation built from granular whole-program runtime data flow, which we term *control-affecting data flow*
- a differential analysis method which surfaces actionable, true-positive differences between control-affecting data flows for purposes of diagnosing program variability bugs

We envision applications of this work not only as a debugging aid for programmers tracing subtle runtime issues without a good idea of where in memory or code to start looking, but also in reproducing precise effects of malicious inputs on parser data and control flow.

Motivation and Related Work

Our goal is to help programmers more easily understand and debug undesirable variability in complex programs like parsers for underspecified formats, reducing such programs' potential attack surface.

Parser Bugs Obscure Specification Divergence

Flaws in parsers make it more difficult to judge whether a given parser correctly follows the implemented input format specification. While some formats have a clear context-free grammar or ASN.1 encoding, many popular formats are unfortunately defined with natural language specifications or reference implementations. Worse (or, arguably, *better* for an attacker) a parser implemented from an unclear or not-backward-compatible specification may unintentionally consider malicious input benign, leading to cascading application component compromise [16] or even system level consequences [17]–[19]. This is exacerbated by the relative lack of runtime memory safety guarantees and parser compartmentalization in languages like C and C++ [20].

Rewriting existing C and C++ parsers in a safer language to reduce exploitability may also be infeasible due to codebase size, lack of modularity, or constraints that dependencies impose [21]. Even a clean parser implementation which conforms to relevant specification may behave subtly differently from other conforming implementations [22], making it difficult to determine the source of an issue afflicting a particular parser from simply comparing its results to those of an assumed correct reference implementation or another parser for the same format. Comparing results of *many* parsers which implement a particular specification as in [23], [24] provides an averaged ground truth for that specification, but will not necessarily yield specifics useful for fixing configuration-to-configuration misbehavior of any one of the compared parsers.

Example: Undefined Behavior

Consider the following (buggy) toy parser, which accepts at least one command-line argument.

Listing 1 According to the C++20 specification, a bitwise left shift operation (as on line 3) results in undefined behavior if the right operand is negative.

```
1  int main(int argc, char* argv[]) {
2      if(argc > 1) {
3          return (int)*argv[argc - 1] << -2;
4      } else {
5          return 0;
6      }
7  }
```

With optimizations *disabled*, Clang / LLVM 15.0.0 run on the code in Listing 1 will produce a binary that operates consistently dependent on input. With optimizations *enabled*, Clang will optimize out the first branch of the conditional, which invokes undefined behavior; the `-O3` optimized program always returns zero.

Listing 2 Assembly of the program in Listing 1 when compiled with Clang's `-O3` optimization level. All of the control flow has been silently elided, and the program will always return 0.

```
main:
        xor    %eax, %eax
        retq
```

The clear difference in the simple examples of program control flow in Listings 1 and 2 above reflect potential effects of optimization passes run on code invoking undefined behavior.

Example: Configuration Variability

Consider now Listing 3, for which we assume there are distinct debug and production build configurations.

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 }
```

Suppose, as above, a particular contributor writes control flow relying on an assertion to check the user-provided value of `shift` is within size bounds of the container `int` type, but another contributor later adds the `NDEBUG` macro to prevent assertion usage (as on Listing 3 lines 1–3) when the code is compiled with `-DPRODUCTION`.

Now suppose a third programmer without knowledge of the source code observes their deployment of the production build allows a `shift` value of 63 (causing integer overflow), though all tests pass. Our third (debugging) programmer runs the debug binary version to reproduce the issue locally, where the `assert()` fails and integer overflow does not occur. If the binary is then instrumented at compile time with a common sanitizer such as UBSan [4] and the program receives `63` as its argument, UBSan will warn that a shift exponent of 63 is too large for the 32-bit `int` type, but will neither show that the `NDEBUG` macro redefines the `assert()` implementation to a no-op, nor show that an assertion guards a risky computation accepting unsanitized user input, where a conditional should be instead. While a static analyser could potentially provide some of this information, particularly if the codebase under analysis were more complex, it would be buried in a large "maybe" state space of potentially dangerous flows to analyse [3], and would not take into account the context that the input value `63` is problematic. If an ordinary programmer without significant knowledge of the codebase beforehand debugging a similar issue aims to quickly fix the real cause in a more complex codebase, neither of these common methods applies cleanly.

Example: Compiler Flaw

Compiled with Clang with optimizations enabled, when run, Listing 4 immediately exits after printing "Hello world!" [25].

Listing 4 A C++ program that one would expect to either enter an infinite busy loop, or immediately exit with code zero. A bug in the latest version of Clang/LLVM (15.0.0) causes this program to erroneously print "Hello world!" when compiled with optimizations enabled.

```cpp
#include <iostream>

int main() {
    while(true);
}

void unreachable() {
    std::cout << "Hello world!" << std::endl;
}
```

When optimizing out the infinite loop (an operation the C++ standard allows), Clang fails to add an implicit `return` at the end of `main()`. Execution thus falls through to the code directly after `main()`: `unreachable()`. A binary built without optimizations does run the infinite `while` loop as expected; Listing 4's execution and output only change when optimizations are enabled. This begs the question of whether a commodity sanitizer such as UBSan could potentially expose such an UB-adjacent issue. Yet when built with Clang (with and without optimizations) and UBSan, via the `-fsanitize=undefined` option (which includes the `-fsanitize=return` check intended to alert when the end of a value-returning function is reached without returning a value) the missing `return` is not caught. Such bugs and their full effects on program control and data flow are difficult to diagnose, particularly in complex programs. This paper proposes a technique to automatically trace back to the source lines most closely related to the origins of such bugs.

Related Work

As mentioned in the previous examples, a debugging programmer today might confirm the presence of a variability bug with sanitizers [5] like UBSan or TSan [26] integrated as Clang or GCC flags. Although some such tools can apply to the same binary build, adding *all* available LLVM or GCC-compatible sanitizers simultaneously to the same binary at compilation time would cause instrumentation conflicts or even obscure the bug, since these sanitizers' instrumentation methodologies group to an extent, but are largely incompatible with each other [27].

A programmer might disassemble or dynamically step through each binary version to identify divergent representations [28] of source variables (a hallmark of variability bugs and undefined behavior), but such an approach can be slow even for an experienced programmer and requires specialized knowledge to succeed. They might also black-box compare differently optimized parser version *outputs* run on the same inputs [29] or compare entirely different parsers which accept the same input format [24] as previously mentioned as a technique to average format ground truth over several parsing implementations, but such an approach would not help a programmer trace a subtle bug which only surfaces under the right build and runtime conditions in a single parser.

Our programmer might also try fuzzing [30] but this is more likely to expose further discrepancies and vulnerabilities in the code rather than help trace the precise origins of an issue relating to processing an already-known particular input. Interestingly, a fuzzing-based approach to control and data flow tracing as in [31], [32] will try to *expand* the aspects of control flow covered to a broad subset of the possibilities one might otherwise obtain through static analysis, in contrast to a dynamic taint tracking approach, which aims to reduce control and data flow under consideration to just what is relevant at runtime for a particular input.

Dynamic Information Flow Tracking (DIFT), also known as Dynamic Taint Analysis (DTA), is a technique which tracks the flow of information through a program at runtime for later replay and analysis [10]. DIFT is a challenging technique to implement precisely; many modern approaches suffer from high implementation overhead, low accuracy, and/or low fidelity [11]. Prior work establishes precedent for using control-flow to complete the data-flow representation of a program's execution [10], and exploring data-flow which affects control flow [12], [13], but primarily improves DIFT representation accuracy and technique overhead, instead of applying and improving on DIFT tooling to help programmers debug runtime program variance issues.

To prevent undertainting, *i.e.* not capturing enough of the program data flow to enable the programmer to accurately draw conclusions about runtime behavior, ReCFI and many similar works attempt to complete the program representation which DIFT tooling obtains through including control flow [10], [12], [13], [27], [33]. These approaches often incorporate fuzzing or even short applications of slower techniques like static analysis or symbolic execution at control flow points to then avoid *overtainting*, *i.e.* creating false positive data relations in the resulting program trace by including many branches the tainted data flow did not actually pass through at runtime, in the quest for achieving complete-enough program data flow coverage. Yet such approaches introduce significant runtime overhead and can still lack precision.

Many such DIFT tools generally are not *universal*, and only track a subset of relevant data through the program [34]. Whole-program, all-the-data DIFT tools *e.g.* Taintgrind [9] do exist, but most lack the performance [35] an universal DIFT approach needs to help a programmer more quickly debug code which accepts real-world-sized input without a good prior idea of what particular parts of that input could interact with buggy portions of the program, or are not directly applicable to C and C++ programs [36].

Further work applies a hybrid control- and data-flow approach over static program representations specifically to detect variability bugs [37]. However, since this approach does not execute the underlying program variants, it is unfortunately prone to false positives.

To reduce overtainting when producing dynamic program traces, Conflux, DTA++, and others limit aspects of control flow considered instead of applying a full dominance-based control flow definition [33], [38], [39]. We find that reducing the examined aspects of control flow correspondingly limits the possible granularity of analysis too much to be useful for detecting variability bugs. Therefore, we use a more traditional dominance-relationship definition of control flow [40].

While our analysis has superficial similarity to some control flow integrity (CFI) schemes, these generally check only indirect jumps at runtime, and abort the running program if anything unexpected happens, since many low-level exploits require control flow modification [41]–[43].

There is additionally interesting and recent precedent for tracing attack origins in C and C++ programs with DIFT [13], [35], but DRTaint only evaluates performance compared to libdft when recording the effects of WannaCry on a Windows machine and does not evaluate accuracy, and ReCFI only evaluates the accuracy and runtime overhead of instrumenting a live old-version NGINX instance to detect exploits of a single buffer overflow vulnerability, CVE-2013-2028. In contrast to these works' goal (which many DIFT works share) of eventually improving data flow tracking and analysis performance to a suitable level for deployment in live production distributed systems, our goals and therefore eventual performance considerations differ: we simply want to give a programmer better local debugging tools for variability bug tracing or attack reproduction.

PolyTracker [44], [45] and SymCC [46] produce some of the most complete program representations over a given input available today through dynamic analysis, but SymCC's evaluation indicates that tool is best used in a tight feedback loop with a fuzzer, without having any particular known-bad input. PolyTracker is intended to make runtime dataflow exploration via DFSan [6] not only simpler but more comprehensive, goals which are attractive for an analysis intended to help ordinary debugging programmers. We currently base our analysis on top of PolyTracker, though to improve program coverage we may also take cues from SymCC's usage of fuzzing for input generation as we refine our approach.

Design

Our approach to detecting and diagnosing variability bugs is based on differential analysis over dynamic data flow program traces. The key factors our design must balance are: obtaining sufficiently detailed data for detection purposes, and not getting lost in fine-grained details which differ between program variants' runtime traces but are unrelated to the variability bug we seek, for correct diagnosis when we compare the traces.

Approach Overview

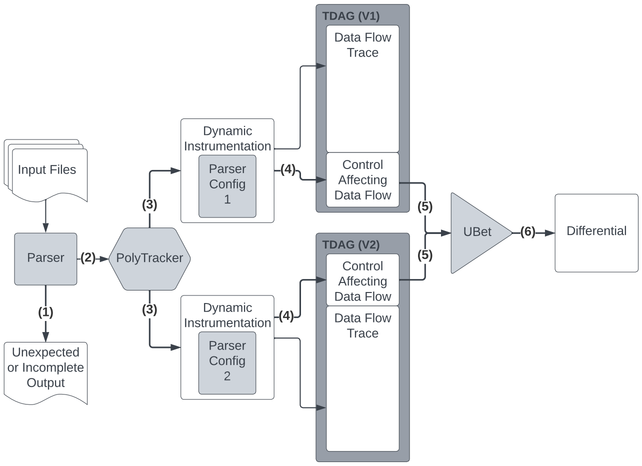

Fig. 1: A high-level overview of the control-affecting data flow differential process. First, we discover a program input that causes unexpected output. Then, we produce at least two different program compile-time configurations, and instrument them with PolyTracker and an additional pre-IR-optimization LLVM pass (PolyTracker instruments following IR optimization). We run each program variant on the same input that we found to initially produce the incorrect output, and PolyTracker's instrumentation plus our additional pass produce a TDAG (Tainted Directed Acyclic Graph) binary file containing a full data flow trace and also the much simpler control-affecting data flow trace that the additional pass produces. UBet's analysis phase then merges (compares) these control-affecting data flow logs to produce the final differential.

For this early version of UBet, we exploit a handy property of parsers: many only perform operations involving input, and such operations are often at least sketched in specification, thus the majority of a parser's functionality should be exercised given input close enough to specification to not cause an early processing abort. Finding *one* input producing an output differential is the minimum prerequisite to try our analysis prototype, which should provide enough information to create regression test case(s) and also address the issue. Since we are primarily concerned in this paper with reproducing and understanding the effects of a particular *known-bad* input on program behavior, the process of finding or generating inputs is out of scope here.

Diffing program-run outputs, or diffing outputs of distinct parsers implementing the same specification as in [24], [29] enables better variability bug *detection*, but does not help with achieving our debugging programmer's goal: identifying the line(s) of source where the unwanted effects originate. Differences between our more granular DIFT program traces, in contrast, are likely to either be due to the methodology used to generate the program variants (*e.g.*, differing optimization pass sets) thus possible to elide, or actually related to the bug, and can be mapped back to specific lines of IR. With some additional runtime information, specific IR markers can be mapped back to source-level symbols.

Control-Affecting Data Flow Log

A program trace TDAG (Tainted Directed Acyclic Graph) [45] produced through PolyTracker's dynamic information flow tracking contains significant runtime dataflow detail. Early experiments that led to the UBet design compared the effects of *blind spots*, which are the set of input bytes whose data flow never influences either control flow that leads to an output or an output itself (don't-cares in the Karnaugh [47] sense) between TDAGs generated from distinct parsers run on the same input using the technique from [45] to try to debug a particular parser. Unfortunately, we found that blind-spots differential analysis, while useful for understanding differences in distinct parsers' implementations of the same format, obscured the run-to-run single-program variability we wanted to diagnose.

We iterated on our program representation, taking inspiration from the Not So Fast [29] program optimization study, which compares versions of the same program differing only in compile-time options to find ROP gadgets that optimization passes create or make more effective for an attacker. Sans compiler bugs or undefined behavior, every compile-time program configuration (*i.e.*, compiler flags vary) should be semantically equivalent. Encountering an input which the program seems to handle differently run-to-run, we produce at least two compile-time configuration parser variants with *e.g.*, different optimization levels (`-O0` vs `-O3`) and instrument those variants to produce TDAGs when run on the same input.

But optimization passes can inline functions, reorder basic blocks, and remove conditionals. Assertions can add conditionals and system calls to control flow. Unexported functions' symbols could change. To account for these benign changes, we match up not only input bytes between runtime traces, but also high-level features of the same control flow, similar to the way we previously tried comparing execution at different parsers' blind spots. We set possible program waypoints via instrumentation before optimization to allow for the greatest possible waypoint similarity across compiled program versions. These higher-level waypoint traces help us interpret the finer detail from a given pair of TDAGs' data flow traces.

Each entry E in a higher-level trace is constructed from information relative to a particular *taint label*, t_j. We add an entry for any t_j that is an operand in a branch condition. During program execution we record additional events for entering and leaving functions, information that later allows us to reconstruct a call stack at each entry. E consists of the identifier $f()_{id}$ of that function, and the list of input byte offsets $b_i...b_n$ which influenced t_j prior to that point:

$$t_j \leftarrow \text{taint label descended from } b_i...b_n$$
$$b_i...b_n \leftarrow \text{input byte offset(s) } i...n$$
$$f()_{id} \leftarrow \text{nearest function identifier}$$
$$E := < b_{i...n}, f()_{id} >$$

We call the hybrid control and data flow program-trace representation consisting of such entries *control-affecting data flow*. This representation naturally cannot include all possible paths through a program like a static analyser would produce; rather, it consists of only the dynamic (unique to a particular input) *subset* of program control flow paths that tainted values affect.

Differential Analysis

The next stage of our process following control-affecting data flow log creation builds the *differential* between control-affecting data flow logs. The first question we must answer to compare two control-affecting data flow logs is whether they generally include the same input byte offsets. A variability bug could cause even the sets of input bytes each parser-run actually evaluated to differ, even if we created both TDAGs with the same program input. Considering each entry in the control-affecting data flow logs of $TDAG_A$ and $TDAG_B$ in input-byte evaluation order, if E_A and E_B do not match, we construct a *differential* entry. If input-byte offsets *match*, the corresponding differential entry has the form

$$< b_iA...b_nA,\ f()_{id}A,\ b_iB...b_nB,\ f()_{id}B >$$

where $f()_{id}A$ does not equal $f()_{id}B$. This means the same taint label, implicitly represented by the set of input byte offsets $b_i...b_n$ which influenced it for ease of comparison as previously mentioned, was an operand to different function identifiers at the same approximate point in control flow. If input-byte offsets in E_A and E_B do *not* match, we include either a differential entry comprised only of E_A or only of E_B to avoid creating a snowballing series of unintentional differential entries where input-byte offsets are *incorrectly* matched; further details on accounting for optimization-introduced differences will be discussed in the Implementation section.

At this point, we map symbols to function identifiers (including recorded call stack) in the differential output to aid human readability. If after this step it is still unclear whether there are any remaining benign differences, to confirm our evaluation of data-affecting control flow we can easily obtain another set of TDAGs from running the same program variants over the same input, or obtain another set of TDAGs to diff from *another substantially similar input*, and then compare across differentials.

IMPLEMENTATION

In this section, we describe how our *UBet* prototype addresses the implementation-level challenges of gathering enough information to sufficiently cover a variability bug's effects, and eliding false positives to allow for diagnosis.

Universal Taint Analysis

For two program variants which differ in output, we recompile each with added instrumentation for *universal taint analysis*. This form of DIFT tracks *all* input bytes throughout the execution of a program, mapping inputs to outputs. To create such runtime records, we use PolyTracker [22], [44], [45]: an LLVM-based dynamic analysis tool that automatically instruments a program to produce a dataflow trace as the program executes over a particular input. Treating each input byte offset as a source of taint, PolyTracker tags the output of any computation involving an input byte as (or part of) an operand as tainted by that particular operand, until a sink (an output such as `printf`, or ultimately even the end of the `main` function) is reached. The resulting trace contains labels representing the results of all program computations descending from any input byte. In this way, each taint label in a runtime dataflow trace can be considered the effective composition of every input byte offset which interacted with that label, or parent operands of that label.

TDAG Modifications

PolyTracker is built on the DFSan dynamic data flow framework; but DFSan requires [6] the debugging programmer to already have at least an idea of what regions of memory and source they would like to explore, since it only provides eight labels, which users are responsible for managing. PolyTracker overcomes DFSan's input-tracking limitations primarily by replacing DFSan's dense matrix representation for taint unions with a new graph data structure, the *TDAG* [45]. Each taint label t_j which PolyTracker assigns at runtime has a fixed-size entry in the TDAG, which links to t_j's parent labels, and eventually back to particular input bytes $b_i ... b_n$.

We augment PolyTracker's instrumentation (which follows IR compile-time optimization) with an additional pre-optimization instrumentation pass which will record all control flow decisions that tainted values affected at runtime. Both these compatible sets of instrumentation write to a single TDAG. We select function identifiers prior to IR optimization, so they have the greatest chance of matching across program variants which differ in optimization level. Yet we do not record every function in the source like a static analyser would; rather, as we are tracing the complete data flow record of the program at runtime, if a tainted value affects control flow, as previously mentioned in the Design section, we add an entry to the control flow log in the TDAG representing the related taint label and nearest-scope function identifier.

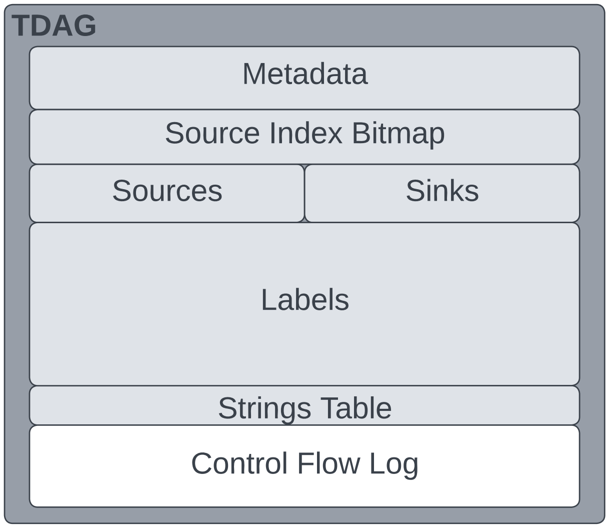

Fig. 2: The modified TDAG format including control-affecting data flow. Control-affecting data flow consists of taint labels that provide a valid index into the overall program dataflow trace, mapped to the nearest-scope function identifier. We add an entry to the Control Flow Log section if at least one of the values in a conditional within function scope was tainted; or, if control flow enters or leaves a function. Briefly, the Metadata section describes the overall layout of the file; the source index bitmap links the sources section and the labels sections together; the strings table contains symbols from the instrumented binary. From the tainted data flow propagation record and this additional control-affecting data flow trace, our UBet analysis phase reconstructs each tainted conditional, falling back to callstack information as needed.

If we assign a particular identifier $f()_{id}$ for a function operating on taint labels descended from particular input bytes in an unoptimized program variant $TDAG_A$, in general, $f()_{id}B$ will handle taint labels descended from the same input bytes in an *optimized* program variant $TDAG_B$ that $f()_{id}A$ handled in $TDAG_A$, even if the actual labels $f()_{id}$ is associated with differ between the control flow logs of $TDAG_A$ and $TDAG_B$. The primary exception to the above is if a function is elided completely by optimization; the other exception is in the case of function inlining, in which case we fall back to comparing the call stack information. This reduces false positive differences due to standard compiler optimizations when comparing program variant TDAGs. A further refinement to the differential format in future work will be to automatically associate and replace function identifiers with the nearest-scope symbol.

Our updated version of the TDAG format includes both PolyTracker's data flow log and a new section which stores the information generated by our additional instrumentation pass. This enables filtering for relevancy without loss of ability to trace each control flow affecting label back to its source input byte-set via PolyTracker's full data flow graph.

Observer Effect

Recording data flow from instrumentation placed before and after LLVM optimizations run has a secondary benefit: we can check if the pre and post-optimization sections in a TDAG from a program run are consistent. If these records are consistent, this provides some evidence our instrumentation was transparent [15], meaning that the compiler did not transform locations where we inserted pre-optimization instrumentation in a way resulting in our instrumentation changing program control and data flow, and also meaning our post-optimization instrumentation pass did not affect execution. We can additionally check an instrumented program variant's regular output against the uninstrumented program output for the same input to prove our dynamic instrumentation *in aggregate* does not observably influence the expected program result. If these checks succeed, we conclude our instrumentation did not change the instrumented program variant binary in a semantically meaningful way.

Differential Analysis

Generally, PolyTracker-assigned labels *will* differ between program variant traces, but the input bytes those labels stem from should be identical for semantically equivalent executions. We therefore initially check that input bytes (but not taint value labels) recorded as control-flow-affecting match up across the compared TDAGs. Then, if the function reference at a given point in $TDAG_A$ matches that in $TDAG_B$, we can reasonably compare those points in control-affecting data flow across the TDAGs.

Comparing optimized and unoptimized variants might result in a differential similar to Table I in the Initial Results section, where we clearly see functions in the unoptimized program variant (left) encountered by same input bytes' tainted data flows more times than in the optimized variant, but our representation allows a $TDAG_A$ entry which does not map to a $TDAG_B$ entry if the previous entry has the same function name or input bytes (respectively, a B-side entry can be included without having a direct A-side mapping). Since input-byte evaluation guides the order in which we consider and compare taint labels (therefore, entries in the control-affecting data flow logs), only producing entries with unequal control-flow-point identifiers would result in a differential with clearly staggered discrepancies, as optimization might result in fewer branching operations or function calls on the same values, for example.

Though we select function and basic-block identifiers sequentially when we instrument the program prior to IR optimization, in optimized program variants, some function identifiers may not appear due to function inlining. To avoid this staggering, we include two other types of differential entry:

- if a branch condition occurs on a taint label in A but *not* in B, we include an entry of the form $< b_iA...b_nA,\ f()_{id}A >$
- respectively, if a branch condition occurs on a taint label in B but *not* in A, we include an entry of the form $< b_iB...b_nB,\ f()_{id}B >$

This helps us cleanly represent optimization and inlining-related differences where both variants process the same input bytes, but the TDAG diff after that point might otherwise be misaligned.

Now, we can see precisely the input bytes whose tainted data flows pass through problematic areas of program control flow. Our method for differentially comparing TDAGs can not only trace output-influencing behavior which causes differences in control flow back to the sections of source code which likely caused it, it can also help a programmer debug largely internal program behavior with security impact such as integer overflow or underflow which typically occurs only in the presence of certain inputs. To date, to check we account reasonably for inlining, we have experimented with constructing extra differentials (such as `-O0` vs `-O1`, `-O1` vs `-O2`) and then differentiating between *those*, though as our work matures, we may turn off all inlining at compile time when constructing program variants to better expose true-positive differences. Our additions to PolyTracker to support UBet analysis are hosted on GitHub[1].

INITIAL RESULTS

Applying our UBet prototype to the US Department of Defense National Imagery Transmission Format (NITF) [48] reference parser Nitro [49], we obtain preliminary results which demonstrate that our approach is worthy of further development and refinement.

NITF

NITF is a binary image file format. Each NITF packages one or more visual data representations (video, fingerprints, CAT scan, JPEG, *etc.*) with extra metadata and other conditionally included information *e.g.*, captions, information for rendering visual redactions, or geo-reference data. Nitro parses multiple mutually incompatible versions of the NITF specification. To simulate the effects of running in production and encountering a particular bad input we would like reproduce the effects of, we apply Nitro instrumented with UBet to a corpus of 148 valid and known-invalid NITF files.

Differential Analysis

Table I contains an excerpt of the differential between two TDAGs obtained respectively from the instrumented Debug and Release versions of Nitro run on a single NITF 2.0 [50] file. The byte offsets are the locations within the program input stream that affected the control flow within each function in the trace. Note that each variant has differing function sequences, *e.g.*, due to optimization and function inlining. For example, the debug build does not make the final call to `std::__1::basic_stringbuf<...>::overflow(int)`.

UBet resolves these differences and matches equivalent function calls based off of data-flow similarities and function symbols. It is immediately evident from this UBet output that the causal variability bug has something to do with byte offset 756 of the input file, since this was the last offset affecting control flow before the variants' traces ultimately diverged. The Debug trace (left side) stops after reaching labels 42 and 1860, which were affected by input byte 756. Since the function log each TDAG contains is ordered by data flow over the course of program execution, from the Debug Nitro UBet function log we easily obtain the last

[1] https://github.com/trailofbits/polytracker/

TABLE I: Excerpt of a data-flow differential between two program traces. Offsets are the locations within the program input stream that affected control flow within each function. It is clear from this differential that its causal variability bug has something to do with byte offset 756 of the input file, since it was the last offset before the traces ultimately diverge.

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}
X Debug trace diverges here	`std::__1::basic_stringbuf<...>::overflow(int)`	{764}
		{764}
		{772}
		{772}
		{774}
		{774}
		{775}
		{775}
		{777}
	⋮	⋮

function influenced by input byte 756 during execution of `showImages(nitf::Record const&)` (and can cross check it against the Release Nitro log) before the program aborted:

```
TRY_SHOW( imsub.imageRepresentation() );
```

Not only does UBet help us find the approximate final function executed in Debug Nitro, we can now determine the exact problematic input byte's value: “Y”.

UBet's control-affecting data flow differential results show why Nitro incorrectly processes some NITFs in its Release build configuration (with compiler optimizations), but aborts during processing when built with Debug options (compiled with `-O0`), as shown in Table I. This was accomplished with only a superficial understanding of the software and NITF specification and did not require interactive debugging or reverse engineering. We then referenced the UBet-indicated function definitions in the Nitro source code and annotated our test NITF file with the NITF 2.0 specification to double-check our instrumented Nitro results against specification and determine if our input file was actually valid. To contrast our results with what can be achieved with commodity sanitizers, we also instrumented the Nitro binary using both UBSan [4] and ASan [51] and processed the offending file. Neither sanitizer produced any error.

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
```

Debugging Outcome

The offending header field starts at offset 756 in our test input file. UBet data flow associations tell us it is 8 bytes long; manual analysis confirmed that “`Y`” is an acceptable first byte value for this field. The NITF 2.0 `IREP` (Image Representation) header field describes the color system and other bands of attribute values to use in interpreting the embedded image data. Supported by all publicly available NITF specification versions [48], [50], [52], the `YCbCr601` JPEG representation records signal brightness in the `Y` band, blue chrominance in the `Cb` band, and red chrominance in the `Cr` band. All three MIL-STD-2500x NITF specifications allow IREP to contain the value `YCbCr601`, but Nitro evidently does not. Our test NITF file embeds a JPEG compressed in the CCIR 601 color space, therefore using the `YCbCr601` representation.

Nitro uniformly uses an `ImageRepresentation` enumeration to describe and extract from the `IREP` field. The relevant lines of Nitro source code are reproduced in Listing 5. Note that the enum definition is missing an entry for `YCbCr601`. Running our instrumented builds on further NITF files which embed `YCbCr601` JPEGs, we confirmed Nitro Debug and Release versions consistently fail on such files when processing the IREP field. We disclosed this bug to the Nitro developers [53].

Conclusions and Future Directions

In this paper, we introduced our *UBet* prototype, a dynamic analysis tool to help programmers debug subtle issues related to or resulting in undefined behavior, run-to-run, and build-to-build program behavior differences. This approach has already successfully identified variability bugs in the Nitro NITF parser [53]. Nothing in UBet is specific to the NITF format itself. UBet can immediately be applied to any C or C++ program compilable with Clang/LLVM.

We see a number of possibilities for automating the final manual specification-checking step of our test process. Firstly, for an input that we have identified as triggering a differential, we could produce an abstract syntax tree annotated with lexical source information, *e.g.*, using a tool like PolyFile [54]. We could then incorporate PolyFile into UBet’s analysis phase to automatically map offending input bytes back to specification fields, similar to the approach used in [22]. This will eliminate manually cross-checking byte offsets and function calls against the parser source and specification, making UBet’s results more digestible. UBet’s diffing method, while functional, could also be improved to resolve more control-flow edge cases. We plan on replacing our initial proof-of-concept efforts with Graphtage [55] to produce an optimal matching between program traces. In addition to exploring Nitro and other binary file format parsers further, we plan to expand our analysis efforts to other historically difficult-to-parse formats like ELF, X.509, XML, and multimedia processors. The *UBet* approach is embarrassingly parallel, insofar as each input can be run on each variant independently. We are also in the process of integrating *UBet* into the Format Analysis Workbench[2] [23], [24] to exploit this parallelism and provide automated differential analysis of parsers over large corpora.

I. Acknowledgments

This research was supported in part by the DARPA SafeDocs program as a subcontractor to Galois under HR0011-19-C-0073. Many thanks to our shepherd Sergey Bratus, to our anonymous reviewers, and to Marek Surovic, Nathan Dautenhahn, Michael Brown, Peter Goodman, Dominik Czarnota, and Lisa Overall for invaluable discussion and feedback.

References

[1] M. Grottke and K. Trivedi, “A classification of software faults,” in *Proceedings of the IEEE Reliability Society Series on Rethinking Software Fault Tolerance*, Jan. 2005.

[2] A. Albertini, “Abusing file formats; or, Corkami, the novella,” *The International Journal of Proof of Concept or GTFO*, no. 0x07, pp. 18–41, Mar. 2015.

[3] A. Mordahl, J. Oh, U. Koc, S. Wei, and P. Gazzillo, “An empirical study of real-world variability bugs detected by variability-oblivious tools,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 5061.

[4] “UndefinedBehaviorSanitizer,” accessed: January 13, 2023. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[5] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, “Sok: Sanitizing for security,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1275–1295.

[6] P. C. et al., “DataFlowSanitizer design discussion,” 2013, accessed: January 12, 2020. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2013-June/062877.html

[7] M. Zalewski, “American fuzzy lop,” http://lcamtuf.coredump.cx/afl/, 2014, accessed: January 12, 2020.

[8] “GDB: The GNU project debugger,” https://sourceware.org/gdb/, accessed: March 16, 2023.

[9] “Taintgrind: a Valgrind taint analysis tool,” https://github.com/wmkhoo/taintgrind, accessed: March 2, 2020.

[2] https://github.com/galoisinc/faw

[10] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP 10. USA: IEEE Computer Society, 2010, pp. 317–331.

[11] C. Brant, P. Shrestha, B. Mixon-Baca, K. Chen, S. Varlioglu, N. Elsayed, Y. Jin, J. Crandall, and D. Oliveira, “Challenges and opportunities for practical and effective dynamic information flow tracking,” *ACM Computing Surveys*, vol. 55, no. 1, November 2021.

[12] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy, “Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 490–504.

[13] O. Braunsdorf, S. Sessinghaus, and J. Horsch, “Compiler-based attack origin tracking with dynamic taint analysis,” in *International Conference on Information Security and Cryptology*. Springer, 2022, pp. 175–191.

[14] W. M. McKeeman, “Differential testing for software,” *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[15] D. Bruening, Q. Zhao, and S. Amarasinghe, “Transparent dynamic instrumentation,” in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 133–144.

[16] H. Lefeuvre, V.-A. Bădoiu, Y. Chien, F. Huici, N. Dautenhahn, and P. Olivier, “Assessing the impact of interface vulnerabilities in compartmentalized software,” in *Proceedings of 30th Network and Distributed System Security (NDSS’23)*. Internet Society, 2022.

[17] CAPEC-113: Interface manipulation. Available from MITRE. An adversary manipulates the use or processing of an interface (e.g. Application Programming Interface (API) or System-on-Chip (SoC)) resulting in an adverse impact upon the security of the system implementing the interface. This can allow the adversary to bypass access control and/or execute functionality not intended by the interface implementation, possibly compromising the system which integrates the interface. Interface manipulation can take on a number of forms including forcing the unexpected use of an interface or the use of an interface in an unintended way. [Online]. Available: https://capec.mitre.org/data/definitions/113.html

[18] CAPEC-554: Functionality bypass. Available from MITRE. An adversary attacks a system by bypassing some or all functionality intended to protect it. Often, a system user will think that protection is in place, but the functionality behind those protections has been disabled by the adversary. [Online]. Available: https://capec.mitre.org/data/definitions/554.html

[19] CAPEC-33: HTTP request smuggling. Available from MITRE. An adversary abuses the flexibility and discrepancies in the parsing and interpretation of HTTP Request messages using various HTTP headers, request-line and body parameters as well as message sizes (denoted by the end of message signaled by a given HTTP header) by different intermediary HTTP agents (e.g., load balancer, reverse proxy, web caching proxies, application firewalls, etc.) to secretly send unauthorized and malicious HTTP requests to a back-end HTTP agent (e.g., web server). [Online]. Available: https://capec.mitre.org/data/definitions/33.html

[20] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, “Clean application compartmentalization with SOAAP,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1016–1031.

[21] A. Bikineev, M. Lippautz, and H. Payer, “Retrofitting temporal memory safety on C++,” https://v8.dev/blog/retrofitting-temporal-memory-safety-on-c++, Jun. 2022, accessed: January 13, 2023.

[22] C. Harmon, B. Larsen, and E. Sultanik, “Toward automated grammar extraction via semantic labeling of parser implementations,” in *Proceedings of the Sixth Workshop on Language-Theoretic Security (LangSec)*. IEEE Symposium on Security and Privacy, 2021, pp. 276–283.

[23] S. Cowger, Y. Lee, N. Schimanski, M. Tullsen, W. Woods, R. Jones, E. Davis, W. Harris, T. Brunson, C. Harmon, B. Larsen, and E. Sultanik, “Research report: Icarus: Understanding de facto formats by way of feathers and wax,” in *2020 IEEE Security and Privacy Workshops (SPW)*, 2020, pp. 327–334.

[24] “FAW: Galois Format Analysis Workbench,” https://github.com/GaloisInc/FAW/, accessed: January 12, 2023.

[25] FrankHB, “Over-aggressively optimization [sic] on infinite loops,” https://github.com/llvm/llvm-project/issues/60622, 2023, accessed: March 8, 2023.

[26] “ThreadSanitizer,” accessed: January 10, 2023. [Online]. Available: https://clang.llvm.org/docs/ThreadSanitizer.html

[27] Y. Rong, P. Chen, and H. Chen, “Integrity: Finding integer errors by targeted fuzzing,” in *International Conference on Security and Privacy in Communication Systems*. Springer, 2020, pp. 360–380.

[28] A. Kellas, “Look out! divergent representations are everywhere!” https://blog.trailofbits.com/2022/11/10/divergent-representations-variable-overflows-c-compiler/, Nov. 2022, accessed: January 6, 2023.

[29] M. D. Brown, M. Pruett, R. Bigelow, G. Mururu, and S. Pande, “Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[30] M. Zalewski, “Automatically inferring file syntax with afl-analyze,” https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html, Feb. 2016, accessed: January 12, 2020.

[31] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, “ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2019, pp. 769–786.

[32] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, “GRIMOIRE: Synthesizing structure while fuzzing,” in *Proceedings of the 28th USENIX Security Symposium*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1985–2002.

[33] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, “Dta++: dynamic taint analysis with targeted control-flow propagation.” in *NDSS*, 2011.

[34] “DataFlowSanitizer,” accessed: January 12, 2020. [Online]. Available: https://clang.llvm.org/docs/DataFlowSanitizer.html

[35] P. Yang, F. Kang, Y. Zhao, and H. Shu, “DRTaint: A dynamic taint analysis framework supporting correlation analysis between data regions,” *Journal of Physics: Conference Series*, vol. 1856, no. 1, p. 012013, April 2021. [Online]. Available: https://doi.org/10.1088/1742-6596/1856/1/012013

[36] M. Höschele and A. Zeller, “Mining input grammars from dynamic taints,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 720–725.

[37] A. V. Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, “Variability-aware static analysis at scale: An empirical study,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, nov 2018.

[38] K. Hough and J. Bell, “A practical approach for dynamic taint tracking with control-flow relationships,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–43, 2021.

[39] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, “Strict control dependence and its effect on dynamic information flow analyses,” in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA ’10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1831708.1831711

[40] F. E. Allen, “Control flow analysis,” *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[41] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[42] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[43] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, “Exploit programming: From buffer overflows to “weird machines” and theory of computation,” *;login:*, vol. 36, 2011.

[44] “PolyTracker: An LLVM-based instrumentation tool for universal taint tracking, dataflow analysis, and tracing.” https://github.com/trailofbits/polytracker, accessed: January 12, 2023.

[45] H. Brodin, E. Sultanik, and M. Surovič, “Blind spots: Identifying exploitable program inputs,” in *Proceedings of the Ninth Workshop on Language-Theoretic Security (LangSec)*. IEEE Symposium on Security and Privacy, 2023.

[46] S. Poeplau and A. Francillon, “Symbolic execution with symcc: Don’t interpret, compile!” in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 181–198.

[47] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 72, no. 5, pp. 593–599, 1953.

[48] "National Imagery Transmission Format (NITF version 2.1) for the national imagery transmission format standard, MIL-STD-2500C," https://web.archive.org/web/20210918070130/https://gwg.nga.mil/ntb/baseline/docs/2500c/2500C.pdf, Geospatial Intelligence Standards Working Group, Reston, VA, Standard, May 2006, accessed: January 12, 2023.

[49] "Nitro: a C cross-platform, full-fledged, extensible library solution for reading and writing the National Imagery Transmission Format (NITF), a U.S. DoD standard format," https://github.com/mdaus/nitro, accessed: February 2, 2023.

[50] "National Imagery Transmission Format (NITF version 2.1) for the national imagery transmission format standard, MIL-STD-2500B," https://web.archive.org/web/20201025132514/https://gwg.nga.mil/ntb/baseline/docs/2500b/2500b.pdf, Geospatial Intelligence Standards Working Group, Reston, VA, Standard, Aug. 1997, accessed: January 12, 2023.

[51] "AddressSanitizer," accessed: March 17, 2023. [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html

[52] "National Imagery Transmission Format (NITF version 2.0) for the national imagery transmission format standard, MIL-STD-2500A," https://web.archive.org/web/20130217004018/https://gwg.nga.mil/ntb/baseline/docs/2500a/2500a.pdf, Geospatial Intelligence Standards Working Group, Reston, VA, Standard, Oct. 1994, accessed: January 12, 2023.

[53] "Missing support for `YCbCr601`," accessed: February 3, 2023. [Online]. Available: https://github.com/mdaus/nitro/issues/528

[54] "PolyFile: a pure Python cleanroom implementation of libmagic, with instrumented parsing from Kaitai struct and an interactive hex viewer." https://github.com/trailofbits/polytracker, accessed: January 12, 2023.

[55] E. Sultanik, "Graphtage: A new semantic diffing tool," https://blog.trailofbits.com/2020/08/28/graphtage/, August 28, 2020, accessed: February 3, 2023.