Data Flows in You: Benchmarking and Improving Static Data-flow Analysis on Binary Executables

Nicolaas Weideman*, Sima Arasteh†, Mukund Raghothaman†, Jelena Mirkovic‡, Christophe Hauser§
*USC Information Sciences Institute, Los Angeles, CA, USA
nhweideman@gmail.com
†University of Southern California, Los Angeles, CA, USA
{arasteh, raghotha}@usc.edu
‡USC Information Sciences Institute, Los Angeles, CA, USA
mirkovic@isi.edu
§Dartmouth College, Hanover, NH, USA
christophe.hauser@dartmouth.edu

***Abstract*—Data-flow analysis is a critical component of security research. Theoretically, accurate data-flow analysis in binary executables is an undecidable problem, due to complexities of binary code. Practically, many binary analysis engines offer some data-flow analysis capability, but we lack understanding of the accuracy of these analyses, and their limitations. We address this problem by introducing a labeled benchmark data set, including $215,072$ microbenchmark test cases, mapping to $277,072$ binary executables, created specifically to evaluate data-flow analysis implementations. Additionally, we augment our benchmark set with dynamically-discovered data flows from 6 real-world executables. Using our benchmark data set, we evaluate three state of the art data-flow analysis implementations, in angr, Ghidra and Miasm and discuss their very low accuracy and reasons behind it. We further propose three model extensions to static data-flow analysis that significantly improve accuracy, achieving almost perfect recall (0.99) and increasing precision from 0.13 to 0.32. Finally, we show that leveraging these model extensions in a vulnerability-discovery context leads to a tangible improvement in vulnerable instruction identification.**

I. Introduction

With the ever-increasing reliance of the modern world on software systems, the cat-and-mouse game of finding and mitigating software vulnerabilities before they are exploited has reached critical levels. This is exacerbated by the growing complexity of software, which increases the burden of ensuring its security. To assuage this burden, program analysis for system security has gained more traction. Data-flow analysis is a critical component in program analysis for security, as understanding how information flows between the instructions of a program is often necessary to evaluate the program's security. Unfortunately, static data-flow analysis is undecidable in the general case. Moreover, when evaluating the security of a program, analyzing the source code is often considered to be insufficient due to the What You See is not What You Execute phenomenon [1]. This creates the necessity for *accurate data-flow analysis* designed to operate *on the binary instructions* of programs.

Fortunately, while data-flow analysis may be undecidable in a general case, there are many practical scenarios in which it is possible to perform data-flow analysis on small code segments (e.g., within a function), by implementing certain approximations in binary analysis engines [2], [3], [4]. These approximations manifest as assumptions made in the model and heuristics incorporated into the implementation of the binary analysis engine. The approximations may at times sacrifice correctness in order to achieve guaranteed termination (and indeed scalability). Incorrect data-flow analysis manifests as false positives (non-existing data flows identified) and false negatives (existing data flows missed). Unfortunately, there is currently no systematic approach to evaluate the size and scope of inaccuracies in binary engines' implementations of data flow analysis, and to identify opportunities for improvements.

This paper aims to benchmark data-flow approaches in binary analysis engines, to produce comprehensive, quantitative and insightful findings about their strengths and limitations. The benchmark dataset should not only measure accuracy of a data-flow analysis approach, but it should also identify root causes for inaccuracies (false positives and false negatives), which can be tackled by future researchers. We demonstrate how the insights from our evaluation led us to propose three improvements to angr's data-flow analysis model, which improved recall from 40% to 99% and precision from 13% to 32%.

Improving data-flow analysis accuracy is an essential step towards improving binary program analysis for system security. This paper demonstrates such an improvement in the context of vulnerability discovery, showing that our improvements to angr lead to higher identification of vulnerable data flows in three case studies.

In summary, this paper makes the following contributions.

- We introduce *alias classes*, a novel concept for categorizing data flows.
- We define and implement an open source framework for generating microbenchmark test cases, labeled with ground truth, that can be used to evaluate static data flow analysis models with respect to our alias classes.
- We define and implement a framework for extracting data flows from real-world programs using dynamic analysis,

to generate ground-truth information about existing data flows.

- With the provided frameworks, we generate a data-flow evaluation benchmark data set, consisting of $215,072$ microbenchmark test cases mapping to $277,072$ unique binary executables, as well as data flows from 6 real-world executables.
- We show the twofold utility of this data set:
  1) We evaluate three state of the art static data-flow analysis implementations in angr [2], Ghidra [3] and Miasm [4]. Our evaluation yields insights about how accurate these analysis engines are with respect to different categories of data flows, and enables us to identify areas for improvement. To the best of our knowledge, this is the first evaluation of its kind.
  2) We propose three novel data-flow model extensions, implement them on top of angr, and evaluate them on real-world executables. These extensions achieve nearly perfect recall, while simultaneously improving precision of data-flow analysis on our benchmarks.
  3) We show that leveraging our model extensions in a vulnerability-discovery context leads to an improved recovery of vulnerability-related instructions in three case studies.

II. Binary-level data-flow analysis

To perform data-flow analysis on a program is to reason about the flow of information between its instructions. Since data flow between two instructions requires executing these instructions in order, data-flow analysis is often built on top of control-flow analysis. Consequently, data-flow analysis inherits the complexities of control-flow analysis, such as *context sensitivity* – the notion that the next instruction in an execution path may be determined by the preceding instructions. For example, the instruction following a return instruction, is determined by the preceding, matching call instruction. This complexity is part of a larger problem caused by indirect branches, in which case the control-flow depends on the data flow. Due to the intermixed nature of control-flow and data-flow analysis, in the general case, both problems are undecidable [5]. Moreover, performing these analyses on binary code, as opposed to source code, adds an extra layer of complexity due to the loss of high-level semantic information, such as control-flow structures, data types and data structures. However, binary code is the most faithful program representation when it comes to reasoning about a range of security properties, and it is therefore our target.

One of the root causes of the undecidability of data-flow analysis lies in the *aliasing problem*, i.e., determining if two instructions access the same data (*i.e.*, if two different pointers point to the same location). In binary program analysis, this means determining if a memory write instruction and a subsequent memory read instruction access memory at the same address. A data flow exists between these instructions if a control flow exists and if they must (or may) access the same data, such that the data written by the first instruction is read by the second instruction, and has not been changed in the meantime. Stated differently, a data flow exists between two instructions if they form a link in a def-use chain [6].

In spite of data-flow analysis being undecidable, a number of theoretical algorithms [7], [8] and implementations of binary-level data-flow analysis [2], [3], [4] have been created to yield approximate solutions. While the theoretical data-flow analysis algorithms make well-defined assumptions in order to guarantee termination, in the implementations it is necessary for developers to deviate from these theoretical models in order to achieve scalability in addition to termination. This deviation manifests as additional, often undocumented, *approximations* (assumptions and heuristics), which may lower analysis accuracy. We discuss examples of such approximations in Section II-C. In this paper we use our benchmarks to quantify the impact on accuracy from various approximations, which enables us to pinpoint areas for improvement, and to implement and demonstrate benefit of several improvements (Section VI).

A. Definitions

In order to rigorously define our approach, we extend the existing concept of data flow with the following definitions.

1) Degree of data flow: We define three different degrees of data flow between any pair of instructions in a program: *unconditional data flow*, *possible data flow* and *impossible data flow*.

A pair of instructions have an *unconditional data flow*, if on every execution of the program, in which these instructions are executed in order, there is a data flow from the first instruction to the second. We show an example of an unconditional data flow in Listing 1.

```
1 mov [rdi], dl ; Write
2 mov al, [rdi] ; Read
```

Listing 1. The instruction pair on lines 1 and 2 have an unconditional data flow.

A pair of instructions have a *possible data flow*, if on at least one execution of the program there is a data flow from the first instruction to the second. We show an example of a possible data flow in Listing 2. In Listing 2, the data written to memory by the instruction on line 1 will be read from memory by the instruction on line 2 if and only if the value in register `rsi` is 0. If this value is dependent on input to the program, then in some executions there will be a data flow, while in others not.

```
1 mov [rdi], dl      ; Write
2 mov al, [rdi+rsi] ; Read
```

Listing 2. The instruction pair on lines 1 and 2 have a possible data flow.

A pair of instructions have an *impossible data flow*, if on every execution of the program, there is no data flow from the first to the second. We show an example of an impossible data flow in Listing 3. In Listing 3, the 1 byte written to memory by the instruction on line 1 will never be read from memory

by the instruction on line 2. Regardless of the value in the register `rdi` (used as the address in Line 1), this value can never be equal to 8 less than itself (used as the address in Line 2).

```
1 mov [rdi], dl   ; Write
2 mov al, [rdi-8] ; Read
```

Listing 3. The instruction pair on lines 1 and 2 have an impossible data flow.

2) Data flow scope: We define the scope of a data flow as being either *intra-procedural*, *inter-procedural* or both. A data flow is considered intra-procedural if it occurs in a single execution of a function. Conversely, a data flow is inter-procedural if its instructions span multiple functions, or multiple executions of a single function.

3) Data flow channel: Given that a data flow is caused by a pair of instructions writing and reading the same data location, we define the *channel of a data flow* as the register or memory that is accessed by both instructions. We indicate this channel by using a token matching the name of the register or `mem` for memory.

We show an example of data flow channels in Listings 4 and 5. In Listing 4, the instructions on lines 1 and 2 have the data flow channel `rbx` as they both access this register. In Listing 5, the instructions on lines 1 and 2 write and read memory at the same address respectively, so they have the data flow channel `mem`.

```
1 mov rbx, 0   ; Write
2 mov rax, rbx ; Read
```

Listing 4. The instruction pair on lines 1 and 2 have a data flow channel `rbx`.

```
1 mov [rbp-0x8], rdi ; Write
2 mov rax, [rbp-0x8] ; Read
```

Listing 5. The instruction pair on lines 1 and 2 have a data flow channel `mem`.

B. Our Scope

In this paper, we separate the intermixed nature of control flow and data flow in order to focus exclusively on the latter. We posit that the main challenge of data flow analysis, independently of control flow, is approximating a solution to the aliasing problem. Therefore, in all our test cases, we focus on data flows between pairs of memory write and read instructions, i.e., with `mem` in the data-flow channel. We do not explicitly determine if a data-flow analysis is capable of identifying data flows between instructions writing and reading the same CPU register. We argue that solving the aliasing problem in this particular case is trivial, since the location where the data is accessed (the register), is identifiable from the encoding of the instruction itself. We further focus on intra-procedural data-flow analysis, and leave inter-procedural data flows for future work. We do this, because most data-flow analysis implementations limit their scope to intra-procedural analyses (in large part because of scalability issues caused by the limitations of existing pointer aliasing analysis models).

C. Data-flow analysis implementations

Due to the scale and complexity of modern software, developers of data-flow analysis implementations are required to introduce approximations in order to make the analysis usable on real-world executables. This includes implementing heuristics to determine if a pair of instructions access the same memory, and if the data written by the first instruction is overwritten prior to being read by the second instruction.

A heuristic to determine if two instructions access the same memory must handle cases where at least one of the addresses is undefined in the scope of analysis. We provide an example of this with Listing 2. We illustrate the complexity of determining whether or not a memory value is overwritten with function calls. Refer to Listings 6 and 7. In both these listings the `f_target` functions are identical, writing to stack memory (labeled as `Write`) and subsequently reading from stack memory (labeled as `Read`). However, these `Write` and `Read` instructions are interrupted by a function call to `f_callee`. In Listing 7 the callee function on line 8 overwrites the same memory written by the `Write` instruction. Therefore this listing shows an *impossible data flow*. On the other hand, in Listing 6 the callee function on line 8 only reads this memory. Therefore this listing shows an *unconditional data flow*. Since the effects of `f_callee` are out of scope for an intra-procedural analysis of function `f_target`, a data-flow analysis implementation must use a single heuristic to handle both these cases. Note that any heuristic will therefore lead to inaccuracies for some `f_target` implementations. While in this case it would be trivial to include the instructions of `f_callee` in the analysis (and perform inter-procedural data-flow analysis), in real-world programs functions tend to form large call graphs, with data-flows spanning multiple functions. Any scalable data-flow analysis will necessarily need to limit its scope of analysis and use heuristics to reason about out-of-scope behavior.

In Sections V and VI we investigate how data-flow analysis implementations approach these types of complexities and we propose alternative approaches that improve analysis accuracy.

```
1 f_target:
2   mov [rsp+0x8], 0   ; Write
3   lea rdi, [rsp+0x8]
4   call f_callee
5   mov rax, [rsp+0x8] ; Read
6   ret
7 f_callee:
8   mov rax, [rdi]
9   ret
```

Listing 6. The instructions on line 2 and line 5 have an unconditional data flow.

```
1 f_target:
2   mov [rsp+0x8], 0   ; Write
3   lea rdi, [rsp+0x8]
4   call f_callee
5   mov rax, [rsp+0x8] ; Read
6   ret
7 f_callee:
8   mov [rdi], 0
9   ret
```

Listing 7. The instructions on line 2 and line 5 have an impossible data flow.

III. Designing a Data Set to Benchmark Static Data-Flow Analysis

Since data flow is undecidable in general [9], [10] the utility of any data-flow model can only be evaluated experimentally. We propose a benchmark data set for evaluation of data-flow models, discussed in Section III-A. We further implement an

automated framework for evaluating data-flow models against our benchmarks, discussed in Section III-B.

A. Data Set

To gain a fine-grained insight into the efficacy of a data-flow analysis model, we break down data flows into a number of categories, which we refer to as *alias classes*, discussed in Section III-A1.

Using these alias classes as a guide, we create a framework for generating a data set of test cases. We divide these test cases into two categories: microbenchmark test cases, discussed in Section III-A2, and real-world test cases (Section III-A3). The microbenchmark test cases are synthesized at the source-code level and then compiled; the compilation process enables us to label them with ground truth data flows. The real-world test cases are extracted from real-world programs. We use a dynamic analysis approach to label them with ground truth about existing data flows (we mitigate side effects of the incompleteness of dynamic analysis coverage in Section V-B).

After generation, this data set is a collection of binaries, each paired with information regarding its functions and data flows. Each data flow is also assigned a degree of data flow as ground truth and an alias class.

1) Alias classes: We categorize an intra-procedural data flow between a pair of instructions by how the pointers – dereferenced by these instructions – are introduced in the function at the source code level. We refer to these categories as *alias classes* and the introduction method of a pointer as the *pointer origin*. We define four such pointer origins, *stack*, *heap*, *foreign* and *global*. For the *foreign* pointer origin, the pointer is defined outside the function and introduced via a function argument. For the *stack* pointer origin, the pointer is introduced as an offset of the stack pointer register. For the *heap* pointer origin, the pointer is introduced via the return value of a memory allocation function. Finally, *global* pointers are allocated upon program initialization and are accessed either as a constant address, or an address relative to the instruction pointer. We show a source-code level illustration of each of the pointer origins in Listing 8. Additionally, we show examples of data flows with alias classes `(Stack, Stack)` and `(Global, Foreign)` in Listings 9 and 10, respectively.

```
1 char global_pointer;
2 void f(char *foreign_pointer) {
3   char stack_pointer;
4   char *heap_pointer = malloc(1);
5 }
```

Listing 8. Pointers with each of the pointer origins.

```
1 char f(char c) {
2   char stack_ptr;
3   stack_ptr = c;
4   return stack_ptr;
5 }
```

Listing 9. An unconditional data flow (line 3 to 4) with alias class `(Stack, Stack)`.

```
1 char global_ptr;
2 char f(char *foreign_ptr) {
3   global_ptr = c;
4   return *foreign_ptr;
5 }
```

Listing 10. A possible data flow (line 3 to 4) with alias class `(Global, Foreign)`.

2) Microbenchmark test cases: The purpose of our microbenchmark test cases is to span a wide variety of intra-procedural data flows that can occur in programs, thus enabling comprehensive evaluation of data-flow analysis. Each test case is designed to be minimalistic, testing a single target data flow in a target function, focused solely on executing this data flow.

At the heart of each test case lie a pair of instructions writing and reading memory addresses, forming the target data flow. The ground truth of each test case – whether there exists an unconditional, possible or impossible data flow between these instructions – depends on the parameters of its construction. The source code of the test cases are generated by enumerating these parameters. Finally, each instance of the source code is compiled in multiple ways, enumerating compiler options, and producing one binary per option set.

a) Pointer creation: The first phase of test case creation is to create the pointers that will be used by the memory access instructions. This phase involves *pointer definition* and *pointer expansion*. In creating the data set, we enumerate a set of properties that make up the pointer definition – a pointer origin, data type, size and length – and expand the pointer into a write and read pointer. The data type is a native data type supported by the compiler and underlying architecture, e.g. integer or floating point. The size attribute of a pointer indicates the number of bits comprising the data type. Finally, the length describes the number of adjacent data types in memory the pointer points to. For a length greater than 1, the pointer points to an array. These properties of a pointer are used to define it in the source code of the test case, including allocating the necessary amount of space on the stack or heap.

In the next phase, *pointer expansion*, we either use a single pointer for both the write and read instruction, or two distinct pointers for each instruction. We refer to the pointer used in the write and read instruction as the *write pointer* and *read pointer*, respectively, regardless of whether or not they are the same. We use a single pointer for both the write and read pointer in order to construct unconditional data flows.

b) Pointer transformation: To increase test case complexity and variety, we introduce an optional pointer transformation. This transformation adds an offset to the pointer, which can either be a constant value, or a variable undefined within the target function. The length attribute of the pointer is used to select an offset within the bounds of the pointer array. The transformed pointers are ultimately used in the target data flow.

c) Callee interruption: As discussed in Section II-C, an intra-procedural data-flow analysis must use approximations in order to handle function calls in the target function. In order to expose these approximations, we create a counterpart for each test case where the target data flow is interrupted by a function call. We show an example of such a pair of test cases in Section VI in Listings 11 and 12.

d) Test case compilation: We increase test case diversity by compiling the source code with a variety of compiler flags. This is useful, because compiler flags such as optimization options have a significant impact on how the source code is converted into binary code. Additionally, options to include or omit the frame pointer have an effect on how stack

variables are represented in binary code. Different binary code representations can may impact accuracy of data flow analysis in different binary analysis engines.

e) Ground truth: With respect to ground truth, we divide the microbenchmark test cases into two subcategories, the *fully-specified* test cases and the *underspecified* test cases.

In *fully-specified test cases* all information regarding the existence of the target data flow is within the target function. In every fully-specified test case, the target data flow either occurs on every execution of the program (unconditional data flow) or does not occurs on any execution of the program (impossible data flow). Therefore, any result reported by a data-flow analysis that contradicts the ground truth can be confirmed as an error. The goal of fully-specified test cases is to uncover the concrete strengths and weaknesses of a particular approach. Listings 1 and 3 are examples of fully-specified test cases.

Underspecified test cases include a target data flow that may or may not occur, depending on data outside of the target function. Listing 2 shows an example of an underspecified test case. Underspecified test cases have *possible* data flows, because by definition, the information missing from the scope of analysis can be defined specifically to cause a data flow. In the example shown in Listing 2, a data flow exists when register `rsi` is equal to 0. Note, however, that in some cases a possible data flow is unlikely. For example, between an instruction writing to global memory and an instruction reading from stack memory. For a data flow to exist, the stack pointer would need to point to global memory. The purpose of the underspecified test cases, is to evaluate how data-flow analysis works in scenarios where perfect information is unavailable. Many real-world binaries contain instances of functions with underspecified data flows.

3) Real-world Test Cases: Our microbenchmark test cases are intentionally simple and unambiguous in order to reflect important program properties. However, these are not meant to reflect the complexity of real-world programs. For this reason, we also enrich our benchmarks with test cases built from real-world binaries. To accomplish this we must address three challenges. Firstly, we need to establish ground truth, i.e. which data flows exist within a given real-world binary. Secondly, we need to construct intra-procedural data-flow graphs composed of these ground-truth data flows. Thirdly, we need to determine the alias class of each of these data flows.

a) Establishing ground truth: Unlike the microbenchmark test cases, where we construct the ground truth in the source code and compile, in real-world test cases the ground truth must be inferred. Dynamic analysis allows us to establish that a data flow exists in some executions of the binary. Therefore, dynamic analysis is capable of uncovering possible data flows in a binary, but it cannot identify unconditional or impossible data flows. By definition, dynamic analysis considers a target program one program path at a time. Since real-world program are large, they potentially have an unbounded number of program paths, making dynamic analysis incomplete. Consequently, our ground truth includes *a number of* possible data flows, which may be incomplete. We address this in Section V-B by computing lower and upper bound approximations for our real-world test cases.

In order to establish ground truth for a target binary, we perform dynamic instrumentation and log all information necessary to recover the data flows. This information includes the instruction address, data access type (write or read), the data location accessed (either a register identifier, or memory address) and the context (the function call in which the instruction executes). Whenever a data write access is encountered, a map is used to associate the data location with the instruction that performed the access. Whenever a data read access occurs, this map is consulted to identify the matching write access. The result is a pair of instructions, for which the first writes to the same data location that is read by the second, thus forming a data flow.

b) Creating the data-flow graphs: The dynamically-collected data flows are consolidated into an inter-procedural data-flow graph, where instruction addresses are the nodes and directed edges indicate a data flow. We annotate the edges in this graph with information pertaining to the scope and channel of the data flow.

In order to separate the inter-procedural data-flow graph into an intra-procedural subgraph we extract a node-induced subgraph, using the instructions of the target functions as the node set. We also eliminate all edges in this subgraph that correspond to a data flow with only an inter-procedural scope.

c) Handling special cases: We apply the following modifications to the dynamically-generated intra-procedural data-flow graph to handle common special cases found in binary programs. A common pattern in binary code is to preserve registers across function calls, by writing the register to memory at the start of the function and then reading it back again into the register at the end. In our dynamically-generated intra-procedural data-flow graphs, this appears as undefined registers used after the function call. We identify these save-restore patterns in the inter-procedural data-flow graph and reconnect the instruction addresses in the target function with an intra-procedural data-flow edge. We show an example of this in Figure 1. The register `rbx` is saved across the function call `f_callee` and therefore, we reconnect the definition and use of this register in `f_target`.

The second modification we make, is to identify and remove data flows leading into an instruction used to clear a register. We show an example of this in Figure 2. The `xor` instruction clears the value in the register `rbx`. Therefore, even though this instruction writes and reads this register, no data flow exists through this instruction. If such an instruction is used to clear a register at the start of the function, in the intra-procedural data-flow graph, any subsequent instruction that reads this register will appear to use an undefined register. Instead, the register-clearing instruction should be interpreted as setting the register to 0, with no incoming data flows.

We make these modifications, because correctly identifying registers undefined in the target function forms the basis

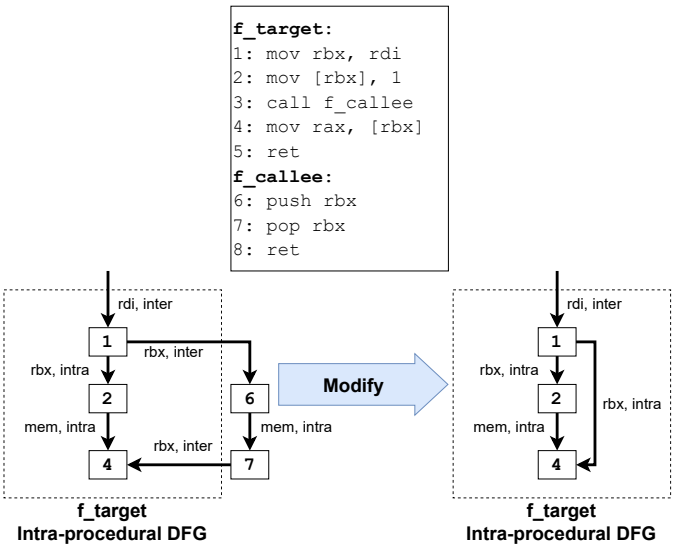

Fig. 1. We modify the intra-procedural data-flow graph to reconnect registers across save-restore edges.

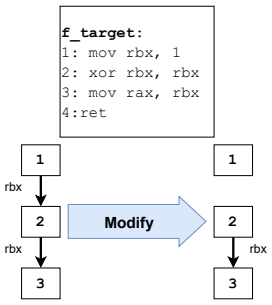

Fig. 2. We modify the intra-procedural data-flow graph to eliminate data flows into register clearing instructions.

of how we identify alias classes in real-world binaries. We explain this next.

d) Alias class identification: Recall that the alias class of a data flow is defined by the origin of the pointers dereferenced in data write and read instructions.

Identifying pointer origins in real-world program is challenging, because between its introduction and use, the pointer may be assigned to a different register and also saved (restored) to (from) memory. Therefore, a simple syntactic analysis of the pointer is insufficient to correctly determine its origin. Indeed, it is necessary to trace the data flow of a pointer back from its use to its introduction into the function. We identify the introduction point of an address by following its intra-procedural data flow backwards until an inter-procedural data flow is encountered. The undefined registers upon which this address depends is the union of the data-flow channels of these inter-procedural edges.

For a stack pointer origin, the pointer is introduced via the stack pointer register, which is undefined within a function (as its value depends on the call stack). Foreign pointers are passed to the target function as arguments. How this is implemented in binary code, depends on the calling convention. In this paper, we focus on the calling convention specified in the System V AMD64 Binary Application Interface. In this calling convention, pointer arguments are passed in the registers `rdi`, `rsi`, `rdx`, `rcs`, `r8`, `r9` (additional pointer arguments are passed via the stack). Therefore, a foreign pointer is introduced via one of these undefined registers. Heap pointers are most often obtained as the return value of a memory allocation function (e.g., `malloc`), which is passed via the `rax` register. Therefore, heap pointers are introduced via an undefined `rax` register. In the case of global pointers, the address does not depend on any undefined registers, because the address is constant or it is an offset of the instruction pointer.

B. Evaluation Framework

We evaluate a data-flow analysis approach by applying it on the binaries, functions and data flows in our benchmark data set. We refer to the binary and function under evaluation as the *target binary* and *target function*, respectively. Similarly, the data flow under test is is the *target data flow*. For each target function, we generate a data-flow graph using the binary analysis engine (e.g., angr) under evaluation. This data-flow graph is examined to determine the presence (or absence) of the edge between the two instructions constituting the target data flow. We compare this information to the ground truth to establish correctness.

IV. Implementation

A. Selected approaches

We evaluate the three state-of-the-art data-flow analysis implementations found in the binary program analysis engines angr [2] (version 9.2.39), Ghidra [3] (version 10.2.2) and Miasm [4] (version 0.1.3.dev447). We refer to these as our *selected approaches*. These binary analysis engines are popular in both academia and the industry, and were selected for being open source, and for providing an implementation of a general-purpose, best-effort data-flow analysis. In Section V, we conduct our evaluation using these selected approaches to uncover the similarities and differences between these approaches, as well as their strengths and weaknesses.

B. Microbenchmark test cases

We implement an open source framework[1] to generate our microbenchmark test cases automatically by following the approach in Section III-A2. For the pointer definition properties, we use the native data types and sizes: 8-bit `int`, 16-bit `int`, `float` and `double`. We also include a defined `struct` data type comprising an integer and pointer value. For the length property of pointers, we assign one of two constant values, 1 and 2, or one of two variable values. These variables are passed to the target function via function parameter. We incorporate these variable lengths to test undefined offset transformations.

After generating the source code, we generate the test case binaries by compiling the source code with the GNU Compiler Collection (GCC, version 11.3.0) using one of the

[1]Anonymized for review

6 optimization options (`O0-O3`, `Os` and `Ofast`) as well varying the inclusion of the function stack frame pointer with `-fomit-frame-pointer`. All pointers are tagged as `volatile` in the source code to prevent the compiler optimizing away the target data flow. Every target binary is compiled with DWARF debug symbols, using the flag `-gdwarf-4`. The debug symbols are used to identify the memory access instructions of the target data flow in the compiled binary, using the `pyelftools` Python library [11], and to establish ground truth for data flows in our microbenchmarks. Finally, given the set of compiled binaries, a fingerprint is calculated for each target function by computing the MD5-hash of the bytes comprising its machine code instructions. The MD5-hash is used to identify and eliminate duplicate functions. The final set of microbenchmark test cases is a set of binaries, each containing a unique target function. In total, we have $215,072$ source code level test cases, mapping to $277,072$ unique target functions.

C. Real-world test cases

For the real-world test cases, we select binaries from the following projects: `chmod`, `cp`, `ls` (from the Coreutils package [12], version v9.1-98-g8613d35be) as well as the `Apatche-Httpd` server [13] (version 2.5.1-dev), the `Mujs` javascript interpreter [14] (version 1.3.3) and `CJson` parser [15] (version v1.7.15-14-gb45f48e). We select these binaries to cover a range of types of application domains and complexities of software.

To dynamically collect execution traces for a target binary, we instrument it using Intel's PIN framework [16], while executing test cases in the project repository. Since the purpose of test cases is to cover a broad range of program functionality, these will allow us to recover data flows across many execution paths. We normalize instruction addresses, to compensate for address space layout randomization (ASLR).

From each of the real-world binaries, we select a subset of target functions, for which we will compute static data-flow graphs in our evaluation. Specifically, we select the 5 functions for which our dynamic analysis approach identified the highest number of memory data flows. We show these functions for each target binary in Table I, with the number of target data flows per alias class. We denote the alias class as *unknown* in the cases where our alias class identification for real-world binaries fails (see Section III-A3). We use these memory data flows as ground truth for the static data-flow analyses.

While we could perform our evaluation on all functions in the target binary, this is often a computationally expensive procedure. We wanted to allocate sufficient time for selected static-analysis approaches to compute the data-flow graph, while still keeping the overall run time manageable. To this end, we select the above-mentioned 5 functions from each target binary and allow a 5 hour time limit per target function.

TABLE I
THE SELECTED TARGET FUNCTIONS FOR EACH REAL-WORLD BINARY WITH THE NUMBER OF IDENTIFIED DATA FLOWS PER ALIAS CLASS.

Target Function	(F, F)	(G, G)	(H, H)	(S, S)	Unknown
`chmod`					
`main`	0	16	1	29	0
`quotearg_buffer_restyled`	0	0	0	37	0
`fts_build`	0	0	0	23	3
`rpl_fts_open`	0	0	1	4	0
`quotearg_n_options`	0	0	0	4	0
`cp`					
`copy_internal`	0	2	0	354	4
`sparse_copy`	1	0	0	44	5
`main`	0	1	0	47	0
`backupfile_internal`	0	0	0	39	0
`make_dir_parents_private`	3	0	2	27	2
`ls`					
`main`	0	66	0	57	0
`quotearg_buffer_restyled`	0	0	0	52	0
`mpsort_with_tmp.part.0`	1	0	0	37	8
`canonicalize_filename_mode`	0	0	1	34	2
`__strftime_internal.isra.0`	0	0	0	28	0
`Apache-Httpd`					
`trie_node_link`	9	0	0	53	0
`ap_add_module`	0	0	0	39	2
`trie_node_alloc`	0	0	1	36	0
`register_filter`	1	0	0	36	0
`ap_setup_prelinked_modules`	0	11	0	24	0
`Mujs`					
`jsR_run`	4	0	0	380	6
`cstm`	0	0	0	323	0
`jsC_cexp`	0	0	0	256	0
`js_gc`	2	0	0	197	0
`statement`	2	0	0	177	0
`CJson`					
`cJSON_Delete`	0	0	0	23	0
`get_object_item`	0	0	0	18	0
`add_item_to_object`	0	0	0	18	0
`add_item_to_array`	0	0	0	16	0
`UnityAssertEqualString`	0	0	0	15	0

D. Evaluation framework implementation

We implement the evaluation framework in an open source repository[2], by creating a Python wrapper script for each selected approach (angr, Miasm, Ghidra) to generate a data-flow graph for a specified target function in a target binary. Each of the selected approaches provide a number of settings to fine-tune the data-flow analysis for a particular use case. We make a best-effort approach to apply the settings that will maximize performance on our data set. This is achieved by performing an analysis with a variety of settings, and selecting the best results per binary analysis engine to report. Next, we discuss these settings as well as the process of converting the approach-specific data-flow information into an unified-representation comparable to ground truth.

a) angr.: By default, angr builds an inter-procedural data-flow graph on top of a control-flow graph (CFG), augmented with symbolic execution. In order to focus this data-flow analysis on the target function, we reduce the scope of CFG generation to only this function. We achieve this by disabling context sensitivity and setting the call-depth parameter to 0. angr produces a data-flow graph over the statements of the Vex IR. Each of the statements in this IR is associated with a machine code instruction via an `IMark`[3] statement, which contains the instruction address of this machine code

[2]Anonymized for review

[3]An `IMark` statement is a special statement in Vex IR that does not describe the behavior an instruction, but instead its address and size in bytes.

instruction. We use these `IMark` statements to convert the data-flow graph produced by angr, to one over machine code instructions.

b) Ghidra.: Ghidra performs its data-flow analysis in the process of converting the machine code instructions to its P-code IR. In this process, the inputs and output of P-code operations are linked. Each of these P-code operations are associated with a machine code instruction address, which we use to create a data-flow graph. Per its default settings, Ghidra performs a number of simplification steps on the produced P-code, such as consolidating some operations. This simplification eliminates the mapping between some machine code instructions and their corresponding P-code operations. To prevent this, we set the simplification style to `firstpass` instead.

c) Miasm.: Similar to angr, for Miasm we first create a CFG for the target function over its intermediate representation. Then, we generate a dependency graph, while instructing Miasm explicitly to consider memory dependencies, while disregarding function calls.

V. Evaluation

We evaluate the selected approaches on both the microbenchmark test cases and real-world test cases of our data set.

A. Microbenchmark test cases

Our microbenchmarks consist of $277,072$ target binaries, each paired with information regarding its target function, target data flow and ground truth. We use our evaluation framework to extract a static data-flow graph for the target function, using each of the selected approaches. We inspect the produced data-flow graphs to determine if the target data flow is reported or not by each approach. We compare this report with the ground truth for the test case. We show the performance of the selected approaches on our microbenchmark test cases in Table II. Here, we only show the results of the fully-specified test cases because these have a clear ground truth. In Section VI, we show angr's performance on some underspecified test cases and we show how we leverage this to improve performance.

In Table II we observe that angr is capable of identifying every unconditional data flow in each alias class. However, it also reported a data flow for each of the impossible data flows in the `(Foreign, Foreign)` and `(Heap, Heap)` alias classes and some (20.19%) of the impossible data flows in the `(Stack, Stack)` alias class. As all these data flows are impossible, each of these cases represents a false positive. In Section VI we conduct an investigation on these false positives and show how they indicate an opportunity for improvement. We also investigate the few cases in which Ghidra reports a data flow. We conclude that Ghidra does not perform alias analysis, it assumes all memory addresses are unequal. There are a few exceptions in which it equates memory addresses in global memory, but we leave the deep-dive into Ghidra's source code to establish the reason for this as future work. Finally, we observe that Miasm sporadically reports unconditional and impossible data flows. Miasm performs alias analysis by syntactically comparing the memory access instructions. This is a reasonable heuristic, but it is also sensitive to how the compiler implements the access instructions, especially with respect to register allocation.

TABLE II
Performance of selected approaches on the microbenchmark test cases

Alias Class	Ground Truth	Edge	Edge %	No Edge	No Edge %	Total
			angr			
(F, F)	unconditional	158	100.00%	0	0.00%	158
(F, F)	impossible	72	100.00%	0	0.00%	72
(G, G)	unconditional	170	100.00%	0	0.00%	170
(G, G)	impossible	0	0.00%	3,726	100.00%	3,726
(H, H)	unconditional	376	100.00%	0	0.00%	376
(H, H)	impossible	12,988	100.00%	0	0.00%	12,988
(S, S)	unconditional	115	100.00%	0	0.00%	115
(S, S)	impossible	717	20.19%	2,835	79.81%	3,552
			Ghidra			
(F, F)	unconditional	0	0.00%	158	100.00%	158
(F, F)	impossible	0	0.00%	72	100.00%	72
(G, G)	unconditional	10	5.88%	160	94.12%	170
(G, G)	impossible	210	5.64%	3,516	94.36%	3,726
(H, H)	unconditional	0	0.00%	376	100.00%	376
(H, H)	impossible	0	0.00%	12,988	100.00%	12,988
(S, S)	unconditional	0	0.00%	115	100.00%	115
(S, S)	impossible	0	0.00%	3,552	100.00%	3,552
			Miasm			
(F, F)	unconditional	106	67.09%	52	32.91%	158
(F, F)	impossible	24	33.33%	48	66.67%	72
(G, G)	unconditional	52	30.59%	118	69.41%	170
(G, G)	impossible	232	6.23%	3,494	93.77%	3,726
(H, H)	unconditional	376	100.00%	0	0.00%	376
(H, H)	impossible	2462	18.96%	10,526	81.04%	12,988
(S, S)	unconditional	107	93.04%	8	6.96%	115
(S, S)	impossible	150	4.22%	3,402	95.78%	3,552

B. Real-world test cases

Our real-world test cases consist of 6 real-world binaries, each paired with information regarding 5 target functions and dynamically recovered data flows. For each target function, we compute the static data-flow graph using each of the selected approaches. We show the run time of this process in Table III. Miasm fails to produce a data-flow graph for 6 target functions, due to exceeding a memory limit of 100GB. By using a profiler, we establish that the excessive memory usage is related to its task list of states to process. These states are differentiated by code location and data flows present at that state. Consequently, the size of this task list may grow disproportionately larger than the size of the target function.

We combine the data flows of each target function in a particular binary and show the total number per alias class in Table IV. Additionally, this table shows how many of the dynamic data flows are discovered by the selected approaches and the number of data flows reported by static analysis only. We see that Ghidra surprisingly identifies data flows in more alias-classes than during the microbenchmark evaluation (Table II). However, after manual analysis, we conclude that these data flows exist between synthetic P-code operations inserted by Ghidra and do not reflect the target data flow.

We use the numbers in Table IV to compute an aggregated performance score for each selected approach on real-world binaries. For each selected approach α, we consolidate all

TABLE III
EXTRACTION TIME OF THE DATA-FLOW GRAPH FOR THE TARGET FUNCTION IN EACH REAL-WORLD BINARY.

Target Function	angr	Ghidra	Miasm
chmod			
`main`	14.63s	12.59s	OOM
`quotearg_buffer_restyled`	60.79s	13.08s	OOM
`fts_build`	16.29s	12.21s	OOM
`rpl_fts_open`	10.60s	12.11s	580.53
`quotearg_n_options`	9.38s	11.90s	OOM
cp			
`copy_internal`	87.95s	16.32s	OOM
`sparse_copy`	17.36s	14.62s	OOM
`main`	16.67s	14.92s	54.52s
`backupfile_internal`	15.90s	14.60s	OOM
`make_dir_parents_private`	16.26s	14.27s	OOM
ls			
`main`	30.81s	18.98s	OOM
`quotearg_buffer_restyled`	66.94s	17.99s	OOM
`mpsort_with_tmp.part.0`	17.37s	16.75s	249.54s
`canonicalize_filename_mode`	18.85s	17.03s	OOM
`__strftime_internal.isra.0`	77.55s	17.53s	OOM
Apache-Httpd			
`trie_node_link`	112.62s	43.09s	2.40s
`ap_add_module`	112.51s	44.51s	6.91s
`trie_node_alloc`	111.97s	43.42s	1.11s
`register_filter`	112.21s	43.97s	1.65s
`ap_setup_prelinked_modules`	112.47s	43.51s	1.66s
Mujs			
`jsR_run`	50.05s	22.39s	0.73s
`cstm`	18.90s	21.94s	0.48s
`jsC_cexp`	24.92s	21.17s	0.44s
`js_gc`	17.45s	21.03s	OOM
`statement`	21.69s	21.70s	19.81s
CJson			
`cJSON_Delete`	5.05s	10.63s	0.78s
`get_object_item`	4.72s	10.81s	0.63s
`add_item_to_object`	4.85s	10.80s	0.55s
`add_item_to_array`	4.64s	11.01s	0.41s
`UnityAssertEqualString`	4.91s	10.56s	0.62s

TABLE IV
THE NUMBER OF DATA FLOWS DISCOVERED DYNAMICALLY PER ALIAS CLASS AND THE NUMBER OF THESE DISCOVERED STATICALLY. ADDITIONALLY, WE SHOW THE NUMBER OF DATA FLOWS DISCOVERED STATICALLY ONLY.

Alias Class	Dyn	angr	angr %	Ghidra	Ghidra %	Miasm	Miasm %
chmod							
(G, G)	16	0	0.00%	0	0.00%	0	0.00%
(H, H)	2	0	0.00%	0	0.00%	1	50.00%
(S, S)	97	24	24.74%	2	2.06%	4	6.19%
unknown	3	1	33.33%	0	0.00%	0	0.00%
Static-only	-	1,331	-	363	-	22	-
cp							
(F, F)	4	0	0.00%	0	0.00%	0	0.00%
(G, G)	3	0	0.00%	0	0.00%	0	0.00%
(H, H)	2	0	0.00%	0	0.00%	0	0.00%
(S, S)	511	128	25.05%	9	1.76%	13	2.54%
unknown	11	2	18.18%	2	18.18%	0	0.00%
Static-only	-	1,555	-	654	-	56	-
ls							
(F, F)	1	0	0.00%	0	0.00%	0	0.00%
(G, G)	66	7	10.61%	4	6.06%	0	0.00%
(H, H)	1	0	0.00%	0	0.00%	0	0.00%
(S, S)	208	96	46.15%	10	4.81%	37	17.79%
unknown	10	1	10.00%	0	0.00%	1	10.00%
Static-only	-	3,030	-	572	-	94	-
Apache-Httpd							
(F, F)	10	9	90.00%	0	0.00%	9	90.00%
(G, G)	11	8	72.73%	0	0.00%	0	0.00%
(H, H)	1	1	100.00%	0	0.00%	1	100.00%
(S, S)	188	133	70.74%	19	10.11%	188	100.00%
unknown	2	1	50.00%	0	0.00%	0	0.00%
Static-only	-	125	-	70	-	36	-
Mujs							
(F, F)	8	2	25.00%	0	0.00%	2	25.00%
(S, S)	1,333	519	38.93%	25	1.88%	217	16.28%
unknown	6	3	50.00%	0	0.00%	0	0.00%
Static-only	-	976	-	363	-	57	-
CJson							
(S, S)	90	79	87.78%	7	7.78	90	100.00%
Static-only	-	70	-	31	-	26	-

data flows discovered dynamically and statically into a sets D and S_α, respectively. We consider the intersection ($D \cap S_\alpha$) the true positive data flows discovered by α. As discussed in Section III-A3, dynamic analysis is incomplete. Every data flow reported only by static analysis ($S_\alpha \setminus D$) may either be a false positive, or a true positive for which we do not have dynamic evidence. We assume each such data flow is a false positive. Consequently, the number of true positives $|D \cap S_\alpha|$ and false positives $|S_\alpha \setminus D|$ we report is a lower and upper bound approximation, respectively. To estimate the number of false negatives, we identify the dynamic data flows not discovered statically ($D \setminus S_\alpha$). This is a lower-bound approximation, because there may be data flows unreported by both dynamic and static analysis. We show these approximations in Table V along with an approximation of the precision, recall an F_1 score. Table V gives us an insight into how each of the selected approaches perform on real-world binaries. We see that Miasm has the highest F_1 score estimation, but as we have seen from Table III it also has scalability issues with respect to memory consumption. We see that angr misses a significant number of data flows and also reports a very large number of assumed false positives. We use this as an opportunity to improve the state of the art in data-flow analysis.

TABLE V
PERFORMANCE OF SELECTED APPROACHES OVER ALL TARGET REAL-WORLD BINARIES.

	angr	Ghidra	Miasm
True positives (lower bound)	1,014	78	563
False positives (upper bound)	7,087	2,053	291
False negatives (lower bound)	1,570	2,506	2,021
Precision (lower bound)	0.1252	0.0366	0.6593
Recall (estimation)	0.3924	0.0302	0.2179
F_1 score (estimation)	0.1898	0.0331	0.3275

VI. IMPROVING THE STATE OF THE ART

In this section, we leverage the results of our evaluation to improve the state of the art in static data-flow analysis. For this purpose, we select angr as the data-flow analysis implementation to investigate and improve. In Section VI-A, we highlight specific behavior of angr that reveals opportunities for improvement. In Section VI-B we introduce three novel data-flow model extensions that serve as alternatives to angr's behavior and show in Section VI-C that these do indeed yield better results. Finally, in Section VI-D, we show that leveraging these model extensions in a vulnerability-discovery context leads to an improved recovery of vulnerability-related instructions.

A. Identifying improvement opportunities

In our evaluation of the selected approaches, we see in Table V that angr has a significant number of false negatives and assumed false positives. This matches what we see in

TABLE VI
THE CHANGE (IN BOLDFACE) INTRODUCED BY $\mathcal{C}_1$ AND $\mathcal{C}_2$ IN HOW ANGR REPORTS DATA FLOWS INTERRUPTED BY A CALLEE FUNCTION.

			angr		angr$_{\mathcal{C}}$	
Alias Class	Ground Truth	Callee	Edge	Edge %	Edge	Edge %
(F, F)	Unconditional	No	158	100.00%	158	100.00%
(F, F)	Under-specified	Yes	0	0.00%	**180**	**100.00%**
(G, G)	Unconditional	No	170	100.00%	170	100.00%
(G, G)	Under-specified	Yes	0	0.00%	**168**	**100.00%**
(H, H)	Unconditional	No	376	100.00%	376	100.00%
(H, H)	Under-specified	Yes	0	0.00%	**376**	**100.00%**
(S, S)	Unconditional	No	115	100.00%	115	100.00%
(S, S)	Under-specified	Yes	0	0.00%	**135**	**100.00%**

TABLE VII
THE CHANGE (IN BOLDFACE) INTRODUCED BY $\mathcal{F}$ WITH RESPECT TO HOW ANGR REPORTS DATA FLOWS BETWEEN POINTERS TRANSFORMED BY EQUAL OR DISTINCT OFFSETS.

			angr		angr$_{\mathcal{F}}$	
Alias Class	Ground Truth	Equal Offset	Edge	Edge %	Edge	Edge %
(F, F)	Unconditional	Yes	158	100.00%	158	100.00%
(F, F)	Impossible	No	72	100.00%	**0**	**0.00%**
(G, G)	Unconditional	Yes	170	100.00%	170	100.00%
(G, G)	Impossible	Yes	0	0.00%	0	0.00%
(G, G)	Impossible	No	0	0.00%	0	0.00%
(H, H)	Unconditional	Yes	376	100.00%	376	100.00%
(H, H)	Impossible	Yes	8,480	100.00%	**3,402**	**40.12%**
(H, H)	Impossible	No	4,508	100.00%	**475**	**10.54%**
(S, S)	Unconditional	Yes	115	100.00%	115	100.00%
(S, S)	Impossible	Yes	717	38.88%	**0**	**0.00%**
(S, S)	Impossible	No	0	0.00%	0	0.00%

Table II, where angr reports many impossible data flows (false positives). These false positives and negatives are a clear indication that angr can be improved by fine-tuning its restrictions for when to report data flows.

In order to show the specific improvement opportunities, we present two additional categorizations of our microbenchmarks in Tables VI and VII. Table VI shows all unconditional data flows, paired with their underspecified counterparts, where the target data flow is interrupted by a function call. We show an example of such a pair of data flows in Listings 11 and 12. From Table VI, we clearly see that the presence of a callee function causes angr to not report the target data flow. Since the callee function introduces out-of-scope modifications to the program state, the ground truth is under-specified. Therefore, one could argue that disrupting all data flows that cross the callee function is an acceptable approach for an intra-procedural data-flow analysis to take. However, in Section VI-B we propose an alternative approach and in Section VI-C we show that this reflects real-world behavior more accurately and therefore yields better results.

```
1 mov BYTE PTR [rsp-0x1],dil ; Write
2 mov al,BYTE PTR [rsp-0x1]  ; Read
```

Listing 11. A fully-specified unconditional data flow between exists between lines 1 and 2.

```
1 sub  rsp,0x10
2 mov  BYTE PTR [rsp+0xf],dil ; Write
3 call 11e9
4 mov  al,BYTE PTR [rsp+0xf]  ; Read
5 add  rsp,0x10
```

Listing 12. The counterpart of Listing 11. The data flow is interrupted by a function call (line 3), resulting in a underspecified data flow between lines 2 and 4.

We highlight our second opportunity for improvement with Table VII. In this table, we select all fully-specified data flows and categorize these with respect to ground truth, and whether or not an offset transformation was applied to both the write and read pointer with equal offsets (discussed in Section III-A2). We show an example of such a test case in Listing 13. In Table VII, we see that angr reports fully-specified, impossible data flows with distinct offsets. Since these are fully-specified test cases, we can confirm angr's behavior as erroneous. In order to understand how to correct this behavior, we investigate angr's source code. Shortly, angr exhibits this behavior due to how it treats undefined memory addresses. Such an address is artificially concretized to a constant specified in angr's source code. This effectively assumes all undefined memory addresses alias. We propose an alternative approach in Section VI-B and in Section VI-C we show that this reflects real-world behavior more accurately and therefore yields better results.

```
1 lea    rax,[rsi+0x1]
2 mov    QWORD PTR [rsp-0x8],rax
3 mov    BYTE PTR [rsi],dil      ; Write
4 mov    rax,QWORD PTR [rsp-0x8]
5 mov    al,BYTE PTR [rax]       ; Read
```

Listing 13. A fully-specified impossible data flow exists between instructions 3 and 5 due to the distinct offsets of register `rsi` that are accessed: `[rsi+0x0]` vs. `[rsi+0x1]`.

B. Data-flow model extensions

Our novel model extensions focus on addressing the limitations discussed in Section VI-A. Two extensions introduce more precise handling of the state of a caller function upon return from a callee function, discussed in Section VI-B1. The third model extension introduces a simple, but effective, way to improve field sensitivity of static data-flow analysis, as described in Section VI-B2. We show how these three model extensions vastly improve the accuracy of static data-flow analysis in Section VI.

1) Handling function calls:

a) $\mathcal{C}_1$: Leveraging calling convention: A calling convention defines how function parameters and return values are passed between a caller function and its callee functions. We propose a model extension that improves intra-procedural data-flow analysis by incorporating information about the calling convention implemented by the target function. Consider Listing 6, in which register `rdi` is used to pass a memory address from the caller function `f_target` (line 3) to callee function `f_callee` (line 8) as a function argument. By identifying such function arguments, a policy can be used to determine whether to preserve or kill definitions at these addresses. We implement our model extensions $\mathcal{C}_1$ with a policy that naively assumes callee functions have no impact on intra-procedural data flows and therefore data flows should be preserved.

b) $\mathcal{C}_2$: Stack frame preservation: Conventionally, callee functions are implemented to preserve and restore the stack frame of their caller function. As the callee function is out of scope for an intra-procedural analysis, our model extension

preserves the stack frame artificially for all callee functions. This is challenging as it requires suppressing the effect that the `call` instruction itself has on the stack frame. In architectures such as `x86_64`, the `call` instruction pushes the subsequent instruction address to the stack automatically, modifying the stack pointer. The matching return instruction, popping this instruction address and restoring the stack pointer, is out of scope of analysis. Therefore, artificial restoration is necessary.

2) Field sensitivity:

a) $\mathcal{F}$: Constant-based Field Disunion: Programming languages often allow the programmer to group related data together into a structure (aka *structs*). Separate values in such a struct are referred to as *fields*. In machine code, this is usually implemented by storing a *base address* of the struct, and accessing the fields by computing an offset from the base address. The sizes of the different fields must be defined at compile time, meaning the offsets to the different fields are constant and can be observed in the assembly code. Our extension assumes there is no data flow between two memory access instructions using distinct constant offsets, as these represent different fields.

In Section VI-A, we mentioned that angr handles undefined memory addresses by concretizing them to a single arbitrary address. We implement $\mathcal{F}$ such that instead of concretizing the entire undefined address, we only concretize the undefined registers used in the address expression. This keeps address concretization sensitive to offsets, as required by $\mathcal{F}$.

C. Evaluating our model extensions

We refer to angr extended with $\mathcal{C}_1$ and $\mathcal{C}_2$ as angr$_{\mathcal{C}}$, with $\mathcal{F}$ as angr$_{\mathcal{F}}$ and with all three extensions as angr$_{\mathcal{CF}}$.

We show the difference between angr and angr$_{\mathcal{C}}$ using our microbenchmarks in Table VI and similarly for angr$_{\mathcal{F}}$ in Table VII. Table VI shows that we have successfully extended angr to preserve data flows that cross a callee function. Table VII shows that we have reduced the cases where angr reports impossible data flows when two distinct offsets are employed. An exception here is with the `(Heap, Heap)` alias class in which 475 (10.54%) impossible data flows with distinct offsets are still reported. After manually investigating a number of the remaining cases, we concluded that the reason for this is because of multi-byte memory accesses. A multi-byte memory access instruction writes or reads to memory at a small range of addresses. Even though two distinct offsets are used for the memory write and read instruction, the ranges overlap. We also observe an unexpected improvement gained by angr$_{\mathcal{F}}$ reducing the reported impossible data flows with *equal* offset transformations in the `(Heap, Heap)` and `(Stack, Stack)` alias classes. We established that the reason for this is a secondary offset introduced by the compiler due to different data types accessed by the write pointer and read pointer. Since angr$_{\mathcal{F}}$ does not distinguish between offsets added explicitly in the source code or by the compiler, it correctly does not report the data flow.

We prove the real-world improvement of angr$_{\mathcal{CF}}$ by re-performing our evaluation on real-world test cases and show the results in Table VIII. We see that angr$_{\mathcal{CF}}$ has a higher F_1 score estimation than angr, Ghidra and Miasm. We gain a significant increase in true positives and reduction in assumed false positives. Indeed, angr$_{\mathcal{CF}}$ achieves nearly perfect recall, meaning any real data flow is likely to be reported by angr$_{\mathcal{CF}}$ with near guaranteed certainty. This is achieved, while simultaneously improving precision from 13% to 32%, i.e., while reducing false positives.

TABLE VIII
THE CONCRETE IMPROVEMENT GAINED BY EXTENDING ANGR WITH OUR MODEL EXTENSIONS $\mathcal{C}_1$, $\mathcal{C}_2$ AND $\mathcal{F}$.

	angr	angr$_{\mathcal{CF}}$
True positives (lower bound)	$1,014$	$2,569$
False positives (upper bound)	$7,087$	$5,351$
False negatives (lower bound)	$1,570$	15
Precision (lower bound)	0.1252	0.3244
Recall (estimation)	0.3924	0.9942
F_1 score (estimation)	0.1898	0.4891

D. Security impact of our model extensions

Data flow is a critical component of vulnerability discovery. Many vulnerability discovery tools, such as BVdetector [17] and BinHunter [18], rely on data flow to capture binary instructions relevant to the vulnerability. Improving the accuracy of flow analysis can help identify the relevant instructions more accurately and, in turn, help improve vulnerability discovery. In this section, we present three case studies. In each case study, we perform manual analysis on the source and assembly code of a vulnerable program in order to identify which instructions are relevant to the vulnerability in question, and to identify the (ground truth) data flows between these instructions. We select these examples to show that angr$_{\mathcal{CF}}$ is capable of identifying the data flows between vulnerability-relevant instructions more accurately than the original angr.

1) CVE-2018-5785: This is an integer-overflow vulnerability that occurred in `openjpeg2` in the module responsible for processing bitmap image files. The bitmap image file format includes a device-independent bitmap (DIB) header, which contains metadata about the image, for example its width, height and color format. The first 4 bytes of this header specify the size of the header. When this size is larger than 56 bytes, the header includes a section for bitmasks, which are used to define which bits in a 32-bit pixel value correspond to the red, green, blue and alpha (transparency) channels. Listing 14 shows the part of the source code of function `bmp_read_info_header` in the vulnerable module, responsible for parsing the bitmasks. In this code snippet, if the header size is at least 56 bytes, the program reads four bytes from the input file and combines them to reconstruct the red-color bitmask value. This is done by shifting and merging each byte into a 32-bit integer. This code is the root cause of an integer overflow, because it allows the mask values to be zero. This zero value is later used to calculate a negative shift value of a left-shift operator (`1 << -1`). This is undefined behavior and may cause crashes or security issues. As shown in Listing 14, the value of `biRedMask` is used

as the left-hand operand in a subsequent operation, creating a data flow across lines 3, 4, 5, and 6. Listing 15 presents the corresponding assembly instructions for this code snippet. The value of `biRedMask`, stored in memory at `[RAX+0x28]`, is used in instruction `fb0d` (line 13), creating a data flow between instructions `faf5` (line 6) and `fb0d` (line 13). Similarly, data flows exist between instruction pairs `(fb16, fb2e)` and `(fb37, fb4f)`. While the original angr fails to detect any of these data flows, angr$_{CF}$ successfully captures all of them. This behavior is consistent across other mask values as well, including `biGreenMask`, `biBlueMask`, and `biAlphaMask`. In total angr$_{CF}$ was able to recover 12 additional true positive data flows between vulnerability-related instructions.

```
1 if (header->biSize >= 56U) {
2
3    header->biRedMask = (OPJ_UINT32)getc(IN);
4    header->biRedMask |= (OPJ_UINT32)getc(IN) << 8;
5    header->biRedMask |= (OPJ_UINT32)getc(IN) << 16;
6    header->biRedMask |= (OPJ_UINT32)getc(IN) << 24;
7 }
```

Listing 14. Part of source code function bmp_read_info_header with integer overflow vulnerability

```
1  fae3: MOV  RAX,qword ptr [RBP + local_10]
2  fae7: MOV  RDI,RAX
3  faea: CALL <EXTERNAL>::_IO_getc
4  faef: MOV  EDX,EAX
5  faf1: MOV  RAX,qword ptr [RBP + local_18]
6  faf5: MOV  dword ptr [RAX + 0x28],EDX     ; Write-1
7  faf8: MOV  RAX,qword ptr [RBP + local_10]
8  fafc: MOV  RDI,RAX
9  faff: CALL <EXTERNAL>::_IO_getc
10 fb04: SHL  EAX,0x8
11 fb07: MOV  EDX,EAX
12 fb09: MOV  RAX,qword ptr [RBP + local_18]
13 fb0d: MOV  EAX,dword ptr [RAX + 0x28]     ; read-1
14 fb10: OR   EDX,EAX
15 fb12: MOV  RAX,qword ptr [RBP + local_18]
16 fb16: MOV  dword ptr [RAX + 0x28],EDX      ; Write-2
17 fb19: MOV  RAX,qword ptr [RBP + local_10]
18 fb1d: MOV  RDI,RAX
19 fb20: CALL <EXTERNAL>::_IO_getc
20 fb25: SHL  EAX,0x10
21 fb28: MOV  EDX,EAX
22 fb2a: MOV  RAX,qword ptr [RBP + local_18]
23 fb2e: MOV  EAX,dword ptr [RAX + 0x28]     ; read-2
24 fb31: OR   EDX,EAX
25 fb33: MOV  RAX,qword ptr [RBP + local_18]
26 fb37: MOV  dword ptr [RAX + 0x28],EDX     ; Write-3
27 fb3a: MOV  RAX,qword ptr [RBP + local_10]
28 fb3e: MOV  RDI,RAX
29 fb41: CALL <EXTERNAL>::_IO_getc
30 fb46: SHL  EAX,0x18
31 fb49: MOV  EDX,EAX
32 fb4b: MOV  RAX,qword ptr [RBP + local_18]
33 fb4f: MOV  EAX,dword ptr [RAX + 0x28]     ; read-3
34 fb52: OR   EDX,EAX
```

Listing 15. Corresponding assembly instructions for Listing 14, which contains an integer overflow

2) CVE-2022-4904: This is a buffer overflow vulnerability that occurred in the `c-ares` package, a C library designed for asynchronous DNS (Domain Name System) resolution. This vulnerability exists in the function `config_sortlist`, responsible for parsing and storing the sortlist configuration used in DNS resolution. We show a snippet of this function to illustrate the vulnerability in Listing 16. In this listing, the while loops on lines 3-4 and 12-13 are responsible for extracting an IP address or range from the configuration file. In line 5, the `memcpy` function copies `q - s` bytes from this IP address in the `str` buffer to the `ipbuf` buffer. Note that no check is placed on the number of bytes copied into the `ipbuf` buffer. Therefore, if the number of bytes copied exceeds the size of `ipbuf` (16 bytes), a buffer overflow occurs. A similar issue arises on line 14 with the `ipbufpfx` buffer. Tracking the values of `q` and `str` is crucial for detecting these overflows, as they determine whether the buffer boundaries are respected in lines 5 and 14.

Listing 17 shows part of the corresponding assembly instructions of the function `config_sortlist`. The values of `str` and `q` used in the `size` function argument of `memcpy` are first defined at addresses `bb96` (line 3) and `bba9` (line 6). The value of `str` has been read in `bc00` (line 13) and `bc0a` (line 15) as arguments for the first `memcpy` function and then again read at addresses `bc93` (line 25) and `bc9d` (line 27) for the second `memcpy` function, and remains constant in between. This creates a data flow between (`bb96`, `bc00`), (`bb96`, `bc0a`), (`bb96`, `bc93`), and (`bb96`, `bc9d`). The original angr fails to identify the last two data flows blocked by the `memcpy` function, while angr$_{CF}$ identifies all of them correctly. Therefore, relying solely on angr's data-flow analysis may lead to missing critical instructions related to the vulnerability. Our evaluation demonstrated that angr$_{CF}$ was able to recover 36 true positive data flows involving vulnerable instructions that were missed by the original angr.

```
1
2  q = str;
3  while (*q && *q != '/' && *q != ';' && !ISSPACE(*q))
4   q++;
5  memcpy(ipbuf, str, q-str);
6  ipbuf[q-str] = '\0';
7  /* Find the prefix */
8
9  if (*q == '/')
10 {
11 const char *str2 = q+1;
12 while (*q && *q != ';' && !ISSPACE(*q))
13  q++;
14 memcpy(ipbufpfx, str, q-str);
15 ipbufpfx[q-str] = '\0';
16 str = str2;
17 }
```

Listing 16. A small part of the source code of the function `config_sortlist`

```
1  bb8e: MOV  qword ptr [RBP + local_80],RDI
2  bb92: MOV  qword ptr [RBP + local_88],RSI
3  bb96: MOV  qword ptr [RBP + local_90],RDX  ; Write-1
4  bb9d: JMP  LAB_0010bf6c
5  bba2: MOV  RAX,qword ptr [RBP + local_90]
6  bba9: MOV  qword ptr [RBP + local_10],RAX
7  bbad: JMP  LAB_0010bbb4
8  bbaf: ADD  qword ptr [RBP + local_10],0x1
9  ...
10 bbf8: TEST  EAX,EAX
11 bbfa: JZ    LAB_0010bbaf
12 bbfc: MOV   RAX,qword ptr [RBP + local_10]
13 bc00: SUB   RAX,qword ptr [RBP + local_90]
14 bc07: MOV   RDX,RAX
15 bc0a: MOV   RCX,qword ptr [RBP + local_90]
16 bc11: LEA   RAX=>local_58,[RBP + -0x50]
17 bc15: MOV   RSI,RCX
18 bc18: MOV   RDI,RAX
19 bc1b: CALL  <EXTERNAL>::memcpy
20 ...
21 bc4d: ADD   qword ptr [RBP + local_10],0x1
22 bc52: RAX,  qword ptr [RBP + local_10]
23 ...
24 bc8f: MOV   RAX,qword ptr [RBP + local_10]
25 bc93: SUB   RAX,qword ptr [RBP + local_90] ; read-1
26 bc9a: MOV   RDX,RAX
27 bc9d: MOV   RCX,qword ptr [RBP + local_90]
28 bca4: LEA   RAX=>local_78,[RBP + -0x70]
29 bca8: MOV   RSI,RCX
30 bcab: MOV   RDI,RAX
31 bcae: CALL  <EXTERNAL>::memcpy
```

Listing 17. Corresponding assembly instructions for function `config_sortlist`

3) CVE-2023-31130: This is a buffer underflow vulnerability in the `c-ares` library's `ares_inet_net_pton()` function, which can be triggered by certain malformed IPv6 addresses. Due to the scope of the vulnerability, we only focus on a small segment, shown in Listing 18. In this listing, the numerical value of an IPv6 hexadecimal segment should be stored in `val`. However, the function fails to constrain `val` within the correct numerical bounds ($0 \leq$ `val` $< 2^{16}$). In order to discover the possible inappropriate value for `val`, it is essential to recover all instructions that contribute to computing this value. Therefore, it is necessary to identify the data flow in Listing 18 from line 1 to 2. Listing 19 shows the corresponding assembly instructions for the source code. Observe that there should be a data flow between `(16608, 1661c)`. While the original angr does not report this data flow, angr$_{\mathcal{CF}}$ does. In total, angr$_{\mathcal{CF}}$ discovers 27 more correct data flows related to this vulnerability than the original angr.

```
1  val <<= 4;
2  val |= aresx_sztoui(pch - xdigits);
3  if (++digits > 4)
4    goto enoent;
```

Listing 18. Part of source code function inet_net_pton_ipv6 that contains buffer underflow vulnerability

```
1 16608: SHL  dword ptr [RBP + local_30],0x4    ; Write
2 1660c: MOV  RAX,qword ptr [RBP + local_48]
3 16610: SUB  RAX,qword ptr [RBP + local_20]
4 16614: MOV  RDI,RAX
5 16617: CALL aresx_sztoui
6 1661c: OR   dword ptr [RBP + local_30],EAX   ; read
```

Listing 19. Corresponding assmbly instructions for function inet_net_pton_ipv6 that contains buffer underflow vulnerability

VII. Future Work

In Section III-A2b we introduce the offset pointer transformation that operates by adding a value to the write or read pointer of a target data flow. Additional pointer transformations can be implemented, such as killing (redefining) the definition of the write instruction. Any additional transformation will yield more insight into how effectively a static data-flow analysis approach can identify data flows in case of such a transformation.

In Section III-A1 we define a pointer origin for pointers passed as function arguments, the *foreign pointers* and for pointers returned from a memory allocation function, the *heap pointers*. It is possible to bridge these two pointer origins with pointers returned from functions other than memory allocation functions. Such a pointer is essentially also a type of foreign pointer, as no information is available regarding its definition site.

VIII. Related Work

To the best of our knowledge, we are the first to evaluate static data-flow analysis approaches on an extensive data set of binary executables. There have been a number of other benchmarks with related, but orthogonal goals. Andriesse et al. [19] and similarly Pang et al. [20], [21] evaluate disassembler implementations on a data set consisting of binaries extracted from the SPEC CPU 2006 benchmark, as well as real-world binaries. Such an evaluation has a number of overlapping goals with ours, such as establishing ground truth information for real-world binaries, but disassembly is a problem orthogonal to data-flow analysis. Di Federico et al. evaluate CFG recovery by creating a data set of binaries with ground truth function boundaries [22]. They compare their novel approach REV.NG with other approaches toward function boundary detection. Data-flow analysis involves a number of challenges independent of control-flow analysis, as discussed in Section II-B.

Hind [23] has surveyed a number of approximating alias analysis solutions on source code. This work is complementary to ours. It approaches the challenge from a theoretical perspective, while we focus on measuring the concrete strengths and weaknesses of implementations of binary data-flow analysis.

Machiry et al. introduced AutoFacts, an approach to inject synthetic facts into real-world programs [24]. These facts allow for ground truth knowledge, that is both sound and complete, regarding aliasing pointers. The injected facts, however, are entirely separate from the logic of the program into which they are injected. Our approach focuses on the other two ends of this spectrum: testing microbenchmarks that are disjoint from real-world program logic and testing data flows fully intertwined in real-world program logic. Additionally, while the AutoFacts data set is introduced, authors did not use it to analyze the implementations of static program analysis. In both cases [23], [24] the alias approximations are divided into a number of dimensions, called sensitivities. Since our selected approaches (angr, Miasm, Ghidra) do not allow for enabling or disabling these sensitivities, we do not use AutoFacts in our evaluation.

IX. Conclusion

In this paper, we introduced a novel approach to classify data flows, namely alias classes. Using these alias classes as a guide, we implemented an open source framework to create

a data set of both microbenchmarks and real-world binaries to evaluate data-flow analysis implementations. We also implement an open source framework to perform this evaluation. Finally, we evaluate angr, Ghidra and Miasm using our data set and framework and provide insights into limitations that each engine has with regards to data flow analysis. By leveraging this evaluation, we propose three model extensions to angr that greatly improve accuracy of its data-flow analysis and can be used to improve vulnerability discovery.

REFERENCES

[1] G. Balakrishnan and T. W. Reps, "WYSINWYX: what you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, 2010. [Online]. Available: https://doi.org/10.1145/1749608.1749612

[2] angr, "The Angr binary analysis platform," http://angr.io, 2016.

[3] Ghidra, "Ghidra," https://ghidra-sre.org/, 2022.

[4] Miasm, "Miasm," https://miasm.re, 2019.

[5] T. W. Reps, "Undecidability of context-sensitive data-independence analysis," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 162–186, 2000. [Online]. Available: https://doi.org/10.1145/345099.345137

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. [Online]. Available: https://www.worldcat.org/oclc/12285707

[7] Á. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy, "Interprocedural static slicing of binary executables," in *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), 26-27 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 2003, p. 118. [Online]. Available: https://doi.org/10.1109/SCAM.2003.1238038

[8] G. Balakrishnan, T. W. Reps, D. Melski, and T. Teitelbaum, "WYSINWYX: what you see is not what you execute," in *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, ser. Lecture Notes in Computer Science, B. Meyer and J. Woodcock, Eds., vol. 4171. Springer, 2005, pp. 202–213. [Online]. Available: https://doi.org/10.1007/978-3-540-69149-5_22

[9] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical society*, vol. 74, no. 2, pp. 358–366, 1953.

[10] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1467–1471, 1994. [Online]. Available: https://doi.org/10.1145/186025.186041

[11] pyelftools, "Pyelftools," https://github.com/eliben/pyelftools, 2023.

[12] coreutils, "Coreutils - GNU core utilities," https://www.gnu.org/software/coreutils/, 2023.

[13] apache, "Apache - HTTP Server Project," https://httpd.apache.org/, 2023.

[14] Artifex, "MuJS," https://mujs.com/, 2023.

[15] cjson, "cJSON - Ultralightweight JSON parser in ANSI C ," https://github.com/DaveGamble/cJSON, 2023.

[16] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[17] J. Tian, W. Xing, and Z. Li, "Bvdetector: A program slice-based binary code vulnerability intelligent detection system," *Inf. Softw. Technol.*, vol. 123, p. 106289, 2020. [Online]. Available: https://doi.org/10.1016/j.infsof.2020.106289

[18] S. Arasteh, J. Mirkovic, M. Raghothaman, and C. Hauser, "Binhunter: A fine-grained graph representation for localizing vulnerabilities in binary executables*," in *Annual Computer Security Applications Conference, ACSAC 2024, Honolulu, HI, USA, December 9-13, 2024*. IEEE, 2024, pp. 1062–1074. [Online]. Available: https://doi.org/10.1109/ACSAC63791.2024.00087

[19] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 583–600. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

[20] C. Pang, T. Zhang, R. Yu, B. Mao, and J. Xu, "Ground truth for binary disassembly is not easy," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 2479–2495. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/pang-chengbin

[21] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 833–851. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00012

[22] A. D. Federico, M. Payer, and G. Agosta, "rev.ng: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*, P. Wu and S. Hack, Eds. ACM, 2017, pp. 131–141. [Online]. Available: http://dl.acm.org/citation.cfm?id=3033028

[23] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, J. Field and G. Snelting, Eds. ACM, 2001, pp. 54–61. [Online]. Available: https://doi.org/10.1145/379605.379665

[24] A. Machiry, N. Redini, E. Gustafson, H. Aghakhani, C. Kruegel, and G. Vigna, "Towards automatically generating a sound and complete dataset for evaluating static analysis tools," *Workshop on Binary Analysis Research (BAR)*, 2019. [Online]. Available: https://par.nsf.gov/biblio/10155111