Datalog Disassembly

Antonio Flores-Montoya
GrammaTech, Inc.
afloresmontoya@grammatech.com

Eric Schulte
GrammaTech, Inc.
eschulte@grammatech.com

Abstract

Disassembly is fundamental to binary analysis and rewriting. We present a novel disassembly technique that takes a stripped binary and produces reassembleable assembly code. The resulting assembly code has accurate symbolic information, providing cross-references for analysis and to enable adjustment of code and data pointers to accommodate rewriting. Our technique features multiple static analyses and heuristics in a combined Datalog implementation. We argue that Datalog's inference process is particularly well suited for disassembly and the required analyses. Our implementation and experiments support this claim. We have implemented our approach into an open-source tool called `Ddisasm`. In extensive experiments in which we rewrite thousands of x64 binaries we find `Ddisasm` is both faster and more accurate than the current state-of-the-art binary reassembling tool, `Ramblr`.

1 Introduction

Software is increasingly ubiquitous and the identification and mitigation of software vulnerabilities is increasingly essential to the functioning of modern society. In many cases—e.g., COTS or legacy binaries, libraries, and drivers—source code is not available so identification and mitigation requires binary analysis and rewriting. Many disassemblers [9, 10, 23, 31, 36, 56, 57, 59], analysis frameworks [4, 7, 12, 19, 25–27, 29, 39, 48], rewriting frameworks [9, 16, 17, 32, 33, 52, 55, 58, 63], and reassembling tools [36, 56, 57] have been developed to support this need. Many applications depend on these tools including binary hardening with control flow protection [20, 37, 40, 54, 62, 64], memory protections [15, 41, 49], memory diversity [14, 30], binary refactoring [53], binary instrumentation [44], and binary optimization [44, 47, 55].

Modifying a binary is not easy. Machine code is not designed to be modified and the compilation and assembly process discards essential information. In general reversing assembly is not decidable. The information required to produce reassembleable disassembly includes:

Instruction boundaries Recovering where instructions start and end can be challenging especially in architectures such as x64 that have variable length instructions, dense instruction sets[1], and sometimes interleave code and data. This problem is also referred as *content classification*.

Symbolization information In binaries, there is no distinction between a number that represents a literal and a reference that points to a location in the code or data. If we modify a binary—e.g., by moving a block of code—all references pointing to that block, and to all of the subsequently shifted blocks, have to be updated. On the other hand, literals, even if they coincide with the address of a block, have to remain unchanged. This problem is also referred to as *Literal Reference Disambiguation*.

We have developed a disassembler that infers precise information for both questions and thus generates reassembleable assembly for a large variety of programs. These problems are not solvable in general so our approach leverages a combination of static program analysis and heuristics derived from empirical analysis of common compiler and assembler idioms. The static analysis, heuristics, and their combination are implemented in Datalog. Datalog is a declarative language that can be used to express dataflow analyses very concisely [50] and it has recently gained attention with the appearance of engines such as Souffle [28] that generate highly efficient parallel C++ code from a Datalog program. We argue that Datalog is so well suited to the implementation of a disassembler that it represents a qualitative change in what is possible in terms of accuracy and efficiency.

We can conceptualize disassembly as taking a series of decisions. *Instruction boundary identification* (IBI) amounts to deciding, for each address x in an executable section, whether x represent the beginning of an instruction or not. *Symbolization* amounts to deciding for each number that appears inside an instruction operand or data section whether it corresponds to a literal or to a symbolic expression and what kind of symbolic expression it is.[2]

[1] Almost any combination of bytes corresponds to a valid instruction.
[2] E.g., `symbol`, `symbol+constant`, or `symbol−symbol`.

The high level approach for each of these decisions is the same. A variety of static analyses are performed that gather evidence for possible interpretations. Then, Datalog rules assign weights to the evidence and aggregate the results for each interpretation. Finally, a decision is taken according to the aggregate weight of each possible interpretation. Our implementation infers instruction boundaries first (described in Sec. 4). Then it performs several static analyses to support the symbolization procedure: the computation of def-use chains, a novel register value analysis, and a data access pattern analysis described in Sec. 5.1, 5.2, and 5.3 respectively. Finally, it combines the results of the static analyses with other heuristics to inform symbolization. All these steps are implemented in a single Datalog program. It is worth noting that—Datalog being a purely declarative language—the sequence in which each of the disassembly steps is computed stems solely from the logical dependencies among the different Datalog rules. Combining multiple analyses and heuristics is essential to achieve high accuracy for IBI and symbolization. No individual analysis or heuristic provides perfect information but by combining several, `Ddisasm` maximizes its chances to reach the right conclusion. The declarative nature of Datalog makes this combination easy.

We have tested `Ddisasm` and compared it to `Ramblr` [56] (the current best published disassembler that produces reassembleable assembly) on 200 benchmark programs including 106 Coreutils, 25 real world applications, and 69 binaries from DARPA's Cyber Grand Challenge (CGC) [1]. We compile each benchmark using 7 compilers and 5 or 6 optimization flags (depending on the benchmark) yielding a total of 7658 unique binaries (888 MB of binary data). We compare the precision of the disassemblers by making semantics-preserving modifications to the assembly code—we "stretch" the program's code address space by adding NOPs at regular intervals—reassembling the modified assembly code, and then running the test suites distributed with the binaries to check that they retain functionality. Additionally, we evaluate the symbolization step by comparing the results of the disassembler to the ground truth extracted from binaries generated with all relocation information. Finally, we compare the disassemblers in terms of the time taken by the disassembly process. `Ddisasm` is faster and more accurate than `Ramblr`.

Our contributions are:

1. We present a new disassembly framework based on combining static analysis and heuristics expressed in Datalog. This framework enables much faster development and empirical evaluation of new heuristics and analyses.
2. We present multiple static analyses implemented in this framework to support building reassembleable assembly.
3. We present multiple empirically motivated heuristics that are effective in inferring the necessary information to produce reassembleable assembly.
4. Our implementation is called `Ddisasm` and it is open source and publicly available[3]. `Ddisasm` produces assembly text as well as an intermediate representation (IR) tailored for binary analysis and rewriting[4].
5. We demonstrate the effectiveness of our approach through an extensive experimental evaluation of over 7658 binaries in which we compare `Ddisasm` to the state-of-the-art tool in reassembleable disassembly `Ramblr`.

2 Related Work

2.1 Disassemblers

Bin-CFI [64] is an early work in reassembleable disassembly. This work requires relocation information (avoiding the need for symbolization). With this information, disassembly is reduced to the problem of IBI. Bin-CFI combines linear disasssembly with the backward propagation of invalid opcodes and invalid jumps. Our IBI also propagates invalid opcodes and jumps backwards, but it couples it with a more sophisticated forward traversal.

Many other works focus solely on IBI [10,31,36,59]. None of these address symbolization. In general they try to obtain a superset of all possible instructions or basic blocks in the binary and then determine which ones are real using heuristics. This idea is also present in our approach. Both Miller et al. [36] and Wartell et al. [59] use probabilistic methods to determine which addresses contain instructions. In the former, probabilistic techniques with weighted heuristics are used to estimate the probability that each offset in the code section is the start of an instruction. In the latter, a probabilistic finite state machine is trained on a large corpus of disassembled programs to learn common opcode operand pairs. These pairs are used to select among possible assembly codes.

Despite all the work on disassembly, there are disagreements on how often challenging features for IBI—e.g., overlapping instructions, data in code sections, and multi-entry functions—are present in real code [6,35,36]. Our experience matches [6] for GCC and Clang, in that we did not find data in executable sections nor overlapping instructions in ELF binaries. However, this is not true for the Intel compiler (ICC) which often allocates jump tables in executable sections.

There are only a few systems that address the symbolization problem directly. Uroboros [57] uses *linear disassembly* as introduced by Bin-CFI [64] and adds heuristics for symbolization. The authors distinguish four classes of symbolization depending on if the source and target of the reference are present in code or data. The difficulty of each class is assessed and partial solutions are proposed for each class.

`Ramblr` [56] is the closest related work. It improves upon Uroboros with increasingly sophisticated static analyses. `Ramblr` is part of the Angr framework for binary analysis [48]. Our system also uses static analyses in combination with

[3] https://github.com/GrammaTech/ddisasm
[4] https://github.com/GrammaTech/gtirb

heuristics. Our static analyses (Sec. 5) are specially tailored to enable symbolization while remaining efficient. Moreover, our Datalog implementation allow us to easily combine analysis results and heuristics.

RetroWrite [18] also performs symbolization, but only for position independent code (PIC) as it relies on relocations. In Sec. 7.1, we argue why we believe that relocations are not enough to perform symbolization even for PIC.

2.2 Rewriting Systems

REINS [58] rewrites binaries in such a way as to avoid making difficult decisions about symbolization. REINS partitions the memory of rewritten programs into untrusted *low-memory* which includes rewritten code and trusted *high-memory* (divided at a power of two for efficient guarding). They implement a lightweight binary lookup table to rewrite each old jump targets with a tagged pointer to its new location in the rewritten code. REINS targets Windows binaries and its main goal is to rewrite untrusted code to execute it safely. REINS uses IDA Pro [3] to perform IBI and to resolve indirect jumps.

SecondWrite [52] also avoids making symbolization decisions by translating jump targets at their point of usage. They do a conservative identification of code and data by performing speculative disassembly and keeping the original code section intact. Any data in the code section can still be accessed, but jumps and call targets are translated to a rewritten code section. SecondWrite disassembles to LLVM IR.

MULTIVERSE [9] goes a step further than SecondWrite and also avoids making code location determinations by treating every possible instruction offset as a valid instruction. Similarly to SecondWrite, it avoids making symbolization determinations by generating rewritten executables in which every indirect control flow is mediated by additional machinery to determine where the control flow would have gone in the original program and redirecting it to the appropriate portion of the rewritten program.

The approaches of REINS, SecondWrite and MULTIVERSE increasingly avoid making decisions about code location and symbolization and thus offer more guarantees to work for arbitrary binaries. However, these approaches also have disadvantages. They introduce overhead in the rewritten binaries both in terms of speed and size. Moreover, the additional translation process for indirect jumps or calls is likely to hinder later analyses on the disassembled code. On the other hand, our approach, although not guaranteed to work, generates assembly code with symbolic references. This enables performing advanced static analyses on the assembly code that can be used to support more sophisticated rewriting techniques. A binary can be rewritten multiple times without introducing a new layer of indirection in every rewrite.

2.3 Static Analysis Using Datalog

Datalog has a long history of being used to specify and implement static analyses. In 1995 Reps [43] presented an approach to obtain demand driven dataflow analyses from the exhaustive counterparts specified in Datalog using the magic sets transformation. Much of the subsequent effort has been in scaling Datalog implementations. In that vein, Whaley et al. [60,61] achieved significant pointer analysis scalability improvements using an implementation based on binary decision diagrams. More recently, Datalog-based program analysis has received new impetus with the development of Souffle [28], a highly efficient Datalog engine. The most prominent application of Datalog to program analysis to date has been Doop [11,50,51], a context sensitive pointer analysis for Java bytecode that scales to large applications. Doop is currently one of the most comprehensive and efficient pointer analysis for Java.

In the context of binary analysis, we are only aware of the work of Brumley et al. [13] which uses Datalog to specify an alias analysis for assembly code. Schwartz et al. [46] present a binary analysis to recover C++ classes from executables written in Prolog. Prolog, being more expressive than Datalog, is typically evaluated starting from a goal—in contrast to Datalog which can be evaluated bottom-up—and using backtracking. Thus, in Prolog programs the order of the inference rules is important and its evaluation is harder to parallelize.

Very recently, Grech et al. [24] have implemented a decompiler, named Gigahorse, for Etherium virtual machine (EVM) byte code using Datalog. Gigahorse shares some high level ideas with our approach, i.e. the inference of high level information from low-level code using Datalog. However, both the target and the inferred information differ considerably. In EVM byte code, the main challenge is to obtain a register based IR (EVM byte code is stack based), resolve jump targets and identify function boundaries. On the other hand, `Ddisasm` focuses on obtaining instruction boundaries and symbolization information for x64 binaries. Additionally, although Gigahorse also implements heuristics using Datalog rules, it does not use our approach of assigning weights to heuristics and aggregating them to make final decisions.

3 Preliminaries

3.1 Introduction to Datalog

A Datalog program is a collection of Datalog rules. A Datalog rule is a restricted kind of horn clause with the following format: $h :- t_1, t_2, \ldots, t_n$ where $h, t_1, t_2, \ldots, t_n$ are predicates. Rules represent a logical entailment: $t_1 \wedge t_2 \wedge \ldots \wedge t_n \rightarrow h$. Predicates in Datalog are limited to flat terms of the form $t(s_1, s_2, \ldots, s_n)$ where $s_1, s_2 \ldots, s_n$ are variables, integers or strings. Given a Datalog rule $h :- t_1, t_2, \ldots, t_n$, we say h is the head of the rule and $t_1, t_2, \ldots, t_n$ is its body.

```
instruction(A:𝔸,Size:ℤ64,Prefix:𝕊,Opcode:𝕊,Op1:𝕆,Op2:𝕆,Op3:𝕆,Op4:𝕆)
invalid(A:𝔸)
op_regdirect(Op:𝕆,Reg:ℝ)
op_immediate(Op:𝕆,Immediate:ℤ64)
op_indirect(Op:𝕆,Reg1:ℝ,Reg2:ℝ,Reg3:ℝ,Mult:ℤ64,Disp:ℤ64,Size:ℤ64)
```

```
data_byte(A:𝔸,Val:ℤ64)
address_in_data(A:𝔸,Val:ℤ64)
```

Figure 1: Initial facts. Facts generated for executable sections on the left and facts generated for all sections on the right.

Datalog rules are often recursive, and they can contain negated predicates, represented as $!t$. However, negated predicates need to be *stratified*—there cannot be circular dependencies that involve negated predicates e.g. $p(X) \text{:}- !q(X)$ and $q(A) \text{:}- !p(A)$. This restriction guarantees that its semantics are well defined. Additionally, all variables in a Datalog rule need to be *grounded*, i.e. they need to appear in at least one non-negated predicate on the rule's body. Datalog also admits disjunctive rules denoted with a semicolon e.g. $h \text{:}- t_1 ; t_2$ that are equivalent to several regular rules $h \text{:}- t_1$ and $h \text{:}- t_2$.

The Datalog dialect that we adopt (Souffle's dialect) supports additional constructs such as arithmetic operations, string operations and aggregates. Aggregates compute operations over a complete set of predicates such as summation, maximums or minimums, and we use them to integrate the results of our heuristics.

A Datalog engine takes as input a set of facts, which are predicates known to be true, and a Datalog program (a set of rules). The engine generates new predicates by repeatedly applying the inference rules until a fixpoint is reached. One of the appeals of Datalog is that it is fully declarative. The result of a computation does not depend on the order in which rules are considered or the order in which predicates within a rule's body are evaluated. This makes it easy to define multiple analyses that depend and collaborate with each other.

In our case, the initial set of facts encodes all the information present in the binary, the disassembly procedure (with all its auxiliary analyses) is specified as a set of Datalog rules. The results of the disassembly are the new set of predicates. These predicates are then used to build an IR for binaries that can be reassembled.

3.2 Encoding Binaries in Datalog

The first step in our analysis is to encode all the information present in the binary into Datalog facts. We consider two basic domains: strings, denoted as $\mathbb{S}$, and 64 bit machine numbers, denoted as $\mathbb{Z}_{64}$. We consider also the following sub-domains: addresses $\mathbb{A} \subseteq \mathbb{Z}_{64}$, register names $\mathbb{R} \subseteq \mathbb{S}$ and operand identifiers $\mathbb{O} \subseteq \mathbb{Z}_{64}$. We adopt the convention of having Datalog variables start with a capital letter and predicates with lower case. We represent addresses in hexadecimal and all other numbers in decimal. We only use the prefix `0x` for hexadecimal numbers if there is ambiguity.

Fig. 1 declares the predicates used to represent the initial

```
416C35:   mov RBX, -624
416C3C:   nop
416C40:   mov RDI, QWORD PTR [RIP+0x25D239]
416C47:   mov RSI, QWORD PTR [RBX+0x45D328]
416C4E:   mov EDX, OFFSET 0x45CB23
416C53:   call 0x413050
416C58:   add RBX, 24
416C5C:   jne 0x416C40
```

Figure 2: Assembly (before symbolization) extracted from wget-1.19.1 compiled with Clang 3.8 and optimization -O2. This code reads 8 byte data elements at address `416C47` within the address range $[45D0B8, 45D328]$ and spaced every 24 bytes.

set of raw instruction facts. Predicate fields are annotated with their type. To generate these initial facts we apply a decoder (Capstone [42]) to attempt to decode *every* address x in the executable sections of a binary[5]. If the decoder succeeds, we generate an `instruction` fact with `A`$= x$. If the decoder fails, the fact `invalid`(x) is generated instead. In each `instruction` predicate, the field `Size` represents the size of the instruction, `Prefix` is the instruction's prefix, and `Opcode` is the instruction code. Instruction operands are stored as independent facts `op_regdirect`, `op_immediate` and `op_indirect`, whose first field `Op` contains a unique identifier. This identifier is used to match operands to their instructions. The fields `Op1` to `Op4` in predicate `instruction` contain the operands' unique identifiers or 0 if the instruction does not have as many operands. We place source operands first and the destination operand last. The predicate `op_regdirect` contains a register name `Reg`, `op_immediate` contains an immediate `Immediate` and `op_indirect` represents an indirect operand of the form `Reg1:[Reg2+Reg3×Mult+Disp]`. That is, `Reg1` is the segment register, `Reg2` is the base register, `Reg3` is the index register, `Mult` represents the multiplier, and `Disp` represents the displacement. Finally, the field `Size` represents the size of the data element being accessed in bytes.

Example 1. Consider the code in Fig. 2. The encoding of the instructions at addresses `416C47` and `416C58` together with their respective operands can be found below:

[5] This is different from linear disassembly which would try to decode an instruction at address $x+s$ after decoding an instruction of size s at address x (skipping the addresses in between).

```
instruction(416C47,7,'','mov',14806,538,0,0)
op_indirect(14806,'NONE','RBX','NONE',1,45D328,8)
op_regdirect(538,'RSI')

instruction(416C58,4,'','add',188,519,0,0)
op_immediate(188,24)
op_regdirect(519,'RBX')
```

Note that the operand identifiers have no particular meaning. They are assigned to operands sequentially as these are encountered during the decoding.

In addition to decoding every possible instruction, we encode every section (both data and executable sections) as follows. For each address `A` in a section, a fact `data_byte(A, Val)` is generated where `Val` is the value of the byte at address `A`. We also generate the facts `address_in_data(A,Addr)` for each address `A` in a section such that the values of the bytes from `A` to `A`+7 (8 bytes)[6] correspond to an address `Addr` that falls in the address range of a section in the binary. These facts will be our initial candidates for symbolization. Executable sections are also encoded this way to support binaries that interleave data with code.

Finally, additional facts are generated from the section, relocation, and symbol tables of the executable as well as a special fact `entry_point(A:`$\mathbb{A}$`)` with the entry point of the executable. Note that for libraries, function symbol predicates are generated for all exported functions and they will be considered as entry points.

4 Instruction Boundary Identification

The predicate `instruction` contains all the possible instructions that might be in the executable. IBI amounts to deciding which of these are real instructions.

Our IBI is based on three steps:

1. A backward traversal starting from `invalid` addresses.
2. A forward traversal that combines elements of *linear-sweep* and *recursive-traversal*.
3. A conflict resolution phase to discard spurious blocks.

Both the backward and forward traversals use the auxiliary predicates `may_fallthrough(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)` and `must_fallthrough(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)` to represent instructions at address `From` that may fall through or must fall through to an address `To`. Fig. 3 contains the rules that define both predicates[7]. An instruction at address `From` may fall through to the next one at address `From+Size` as long as it is not a return, a halt, or an unconditional jump instruction. Rule 1 depends in turn on other auxiliary predicates that abstract away specific aspects of concrete assembler instructions e.g. `return_operation` is simply defined as `return_operation ('ret')` for x64. The predicate `must_fallthrough` restricts

[6] Our analysis considers x64 architecture.

[7] Some of the rules have been slightly adapted for presentation purposes.

```
may_fallthrough(From,To):-
    instruction(From,Size,_,OpCode,_,_,_,_),
    To=From+Size,
    !return_operation(OpCode),
    !unconditional_jump_operation(OpCode),
    !halt_operation(OpCode).                          (1)
must_fallthrough(From,To):-
    may_fallthrough(From,To),
    instruction(From,_,_,OpCode,_,_,_,_),
    !call_operation(OpCode),
    !interrupt_operation(OpCode),
    !jump_operation(OpCode),
    !instruction_has_loop_prefix(From).               (2)
```

Figure 3: Auxiliary Datalog predicates used for traversal.

`may_fallthrough` further by discarding instructions that might not continue to the next instruction i.e. calls, jumps, or interrupt operations (we consider instructions with a loop prefix as having a jump to themselves).

The traversals also depend on other predicates whose definitions we omit: `direct_jump(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)`, `direct_call (From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)`, `pc_relative_jump(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)`, and `pc_relative_call(From:`$\mathbb{A}$`,To:`$\mathbb{A}$`)` represent instructions at address `From` that have a direct or RIP-relative jump or call to an address `To`.

Example 2. Consider the code in Fig. 2. The `mov` instruction at address `416C4E` generates the predicates `must_fallthrough(416C4E,416C53)` and `may_fallthrough (416C4E,416C53)` whereas the call instruction only generates `may_fallthrough(416C53,416C58)`. This is because the function at address `413050` (the target of the call) might not return. The call instruction also generates the predicate `direct_call (416C53,413050)`.

4.1 Backward Traversal

Our backward traversal simply expands the amount of `invalid` predicates through the implication that any instruction unconditionally leading to an invalid instruction must itself be invalid.

```
invalid(From):-
    (must_fallthrough(From,To) ;
        direct_jump(From,To) ;
        direct_call(From,To) ;
        pc_relative_jump(From,To) ;
        pc_relative_call(From,To)),
    (invalid(To) ;
         !instruction(To,_,_,_,_,_,_,_)).             (3)
possible_effective_address(A):-
    instruction(A,_,_,_,_,_,_,_), !invalid(A).        (4)
```

Rule 3 specifies that an instruction at address `From` that jumps, calls or must fall through to an address `To` that does

```
code_in_block_candidate(A,A):-
    possible_target(A),
    possible_effective_address(A).                    (5)
code_in_block_candidate(A,Block):-
    code_in_block_candidate(Aprev,Block),
    must_fallthrough(Aprev,A),
    !block_limit(A).                                  (6)
code_in_block_candidate(A,A):-
    code_in_block_candidate(Aprev,Block),
    may_fallthrough(Aprev,A),
    (!must_fallthrough(Aprev,A) ;
        block_limit(A)),
    possible_effective_address(A).                    (7)
possible_target(A):-
    initial_target(A).                                (8)
possible_target(Dest):-
    code_in_block_candidate(Src,_),
    (may_have_symbolic_immediate(Src,Dest) ;          (9)
        pc_relative_jump(Src,Dest) ;
        pc_relative_call(Src,Dest)).
possible_target(A):-
    after_block_end(_,A).                             (10)
```

Figure 4: Block forward traversal rules.

not contain an potential instruction or to an address `To` that contains an invalid instruction is also invalid. The predicate `possible_effective_address(A:𝔸)` contains the addresses of the remaining instructions not discarded by `invalid` (Rule 4).

4.2 Forward Traversal

The forward traversal follows an approach that falls between the two classical approaches *linear-sweep* and *recursive-traversal*. It traverses the code recursively but is much more aggressive than typical traversals in terms of the targets that it considers. Instead of starting the traversal only on the targets of direct jumps or calls, every address that appears in one of the operands of the already traversed code is considered a possible target. For example, in Fig. 2, as soon as the analysis traverses instruction **`mov EDX, OFFSET`** `0x45CB23`, it will consider the address `45CB23` as a potential target that it needs to explore. Additionally, potential addresses appearing in the data (instances of predicate `address_in_data`) are also considered potential targets.

The traversal is defined with two mutually recursive predicates: `possible_target(A:𝔸)` specifies addresses where we start traversing the code and `code_in_block_candidate(A:𝔸,Block:𝔸)` takes care of the traversing and assigning instructions to basic blocks. A predicate `code_in_block_candidate(A:𝔸,Block:𝔸)` denotes that the instruction address `A` belongs to the candidate code block that starts at address `Block`.

The definition of these predicates can be found in Fig. 4.

The traversal starts with the `initial_target` (Rule 8) that contains the addresses of: entry points, any existing function symbols, landing pad addresses (defined in the exception information sections), the start addresses of executable sections, and *all* addresses in `address_in_data`. This last component implies that all the targets of jump tables or function pointers present in the data sections will be traversed.

However, not all jump tables are lists of absolute addresses (captured by `address_in_data`). Sometimes jump tables are stored as differences between two symbols i.e. `Symbol1−Symbol2`. In these tables, the jump target `Symbol1` is computed by loading `Symbol2` first and then adding the content of the jump table entry. We found this pattern in PIC code and in position dependent code compiled with ICC (see App. A). An approximation of these jump tables is detected with ad-hoc rules and their targets are included in `initial_target`.

A possible target, marks the beginning of a new basic block candidate (Rule 5). The candidate block is then extended as long as the instructions are guaranteed to fall through and we do not reach a `block_limit` (Rule 6). The predicate `block_limit` over-approximates `possible_target` (it is computed the same way but without requiring the predicate `code_in_block_candidate` in Rule 9). Rule 7 starts a new block if the instruction is not guaranteed to fall through or if there is a block limit. That is where the previous block ends. Any addresses or jump/call targets that appear in a block candidate are considered new possible targets (Rule 9). `may_have_symbolic_immediate` includes direct jumps and calls but also other immediates. E.g. instruction **`mov EDX, OFFSET`** `45CB23` generates `may_have_symbolic_immediate(416C4E,45CB23)`. Note that this is much more aggressive than a typical recursive traversal that would only consider the targets of jumps or calls. Finally, Rule 10 adds a linear-sweep component to the traversal. `after_block_end(End:𝔸,A:𝔸)` contains addresses `A` after blocks that end with an instruction that cannot fall through at `End` (e.g. an unconditional jump or a return). This predicate skips any padding (e.g., contiguous NOPs) that might be found after the end of the previous block.

It is worth noting that in our Datalog specification we do not have to worry about many issues that would be important in lower level implementations of equivalent binary traversals. For instance, we do not need to keep track of which instructions and blocks have already been traversed nor do we specify the order in which different paths are explored.

4.3 Solving Block Conflicts

Once the second traversal is over, we have a set of candidate blocks, each one with a set of instructions (encoded in the predicate `code_in_block_candidate`). These blocks represent our best effort to obtain an over-approximation of the basic blocks in the original program. In principle, it is possible to miss code blocks. However, such code block would have to be reachable only through a computed jump/call *and*

be preceded by data that derails the linear-sweep component of the traversal (Rule 10). We have not found any instance of this situation. We remark that if the address of a block appears anywhere in the code or in the data, it will be considered. For instance, ICC puts some jump tables in executable sections. By detecting these jump tables, we consider their jump targets (which are typically the blocks after the jump table) as possible targets in our traversal.

The next step in our IBI is to decide which candidate blocks are real. For that, we detect the blocks that overlap with each other or with a potential data segment (e.g. a jump table in the executable section). Overlapping blocks are extremely uncommon in compiled code. The situations in which they appear tend to respond to very specific patterns such as a block starting with or without a `lock` prefix [35]. We recognize those patterns with ad-hoc rules and consider that the remaining blocks should not overlap. Thus, if two blocks overlap, we assume one of them is spurious and needs to be discarded. This assumption could be relaxed if we wanted to disassemble malware but it is generally useful for compiled binaries.

We decide which blocks to discard using heuristics. Each heuristic is implemented as a Datalog rule that produces a predicate of the form `block_points(Block:`$\mathbb{A}$`,Src:`$\mathbb{A}$`,Points:`$\mathbb{Z}_{64}$`,Why:`$\mathbb{S}$`)`. Such a predicate assigns `Points` points to the block starting at address `Block`. The field `Src` is an optional reference to another block that is the cause of the points or zero for heuristics that are not based on other blocks. The field `Why` is a string that describes the heuristic for debugging purposes and to distinguish the predicate from others generated from different heuristics.

We compute the total number of points for each block using Souffle's aggregates [28]. Then, given two overlapping blocks, we discard the one with the least points. In case of a tie, we keep the first block and emit a warning. We also discard blocks if their total points is below a threshold. This is useful for blocks whose heuristics indicate overlap with data elements.

Our heuristics are mainly based on how blocks are interconnected, how they fit together spatially, and whether they are referenced by potential pointers or overlap with jump tables. Some of the heuristics used are described below (+ for positive points and − for negative points):

+ The block is called, jumped to, or there is a fallthrough from a non-overlapping block.
+ The block's initial address appears somewhere in the code or data sections. If the appearance is at an aligned address, it receives more points.
+ The block calls/jumps other non-overlapping blocks.
− A potential jump table overlaps with the block.

All memory not covered by a block is considered data.

5 Auxiliary Analyses

The next step in our disassembly procedure is symbolization. However, we first perform several static analyses to infer how data is accessed and used, and thus deduce its layout.

5.1 Register Def-Use Analysis

First, we compute register definition-uses chains. The analysis produces predicates of the form:

`def_used(Adef:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`,Aused:`$\mathbb{A}$`,Index:`$\mathbb{Z}_{64}$`)`

The register `Reg` is defined at address `Adef` and used at address `Aused` in the operand with index `Index`.

The analysis first infers definitions `def(Adef:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`)` and uses `use(Aused:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`,Index:`$\mathbb{Z}_{64}$`)`. Then, it propagates definitions through the code and matches them to uses. The analysis is intra-procedural in that it does not traverse calls but only direct jumps. This makes the analysis incomplete but improves scalability. During the propagation of definitions, the analysis assumes that certain registers keep their values through calls following Linux x64 calling convention [34].

Example 3. Consider the code fragment in Fig. 2. The Def-Use analysis produces the following predicates:

```
def_used(416C35,'RBX',416C47,1)
def_used(416C35,'RBX',416C58,2)
def_used(416C58,'RBX',416C58,2)
def_used(416C58,'RBX',416C47,1)
```

One important detail is that the analysis considers the 32 bits and 64 bits registers as one given that the x64 architecture zeroes the upper part of 64 bits registers whenever the corresponding 32 bits register is written. That means that for instruction **`mov EDX, OFFSET`** `0x45CB23` at address `416C4E`, the analysis generates a definition `def(416C4E,RDX)`.

Once we have def-use chains, we want to know which register definitions are potentially used to compute addresses to access memory. For that purpose, the disassembler computes a new predicate:

`def_used_for_address(Adef:`$\mathbb{A}$`,Reg:`$\mathbb{R}$`)`

that denotes that the register `Reg` defined at address `Adef` might be used to compute a memory access. This predicate is computed by traversing def-use chains backwards starting from instructions that access memory. This traversal is transitive, if a register R is used in an instruction that defines another register R' and that register is used to compute an address, then R is also used to compute an address. This is captured in the following Datalog rule:

```
def_used_for_address(Adef,Reg):-
    def_used_for_address(Aused,_),
    def_used(Adef,Reg,Aused,_).
```
(11)

5.2 Register Value Analysis

In contrast to instructions that refer to code, where direct references (direct jumps or calls) predominate, memory accesses

are usually computed. Rather than accessing a fixed address, instructions typically access addresses computed with a combination of register values and constants. This address computation is often done over several instructions. Such is the case in the example code in Fig. 2.

In order to approximate this behavior, we developed an analysis that computes the value held in a register at an address. There are many ways of approximating register values ranging from simple constant propagation to complex abstract domains that take memory locations into account e.g. [8]. Generally, the more complex the analysis domain, the more expensive it is. Therefore, we have chosen a minimal representation that captures the kind of register values that are typically used for accessing memory. Our value analysis representation is based on the idea that typical memory accesses follow a particular pattern where the memory address that is accessed is computed using a base address, plus an index multiplied by a multiplier. Consequently, the value analysis produces predicates of the form:

```
reg_val(A:𝔸,Reg:ℝ,A2:𝔸,Reg2:ℝ,Mult:ℤ64,Disp:ℤ64)
```

which represents that the value of a register `Reg` at address `A` is equal to the value of another register `Reg2` at address `A2` multiplied by `Mult` plus an displacement `Disp` (or offset).

The analysis proceeds in two phases. The first phase produces predicates of the form `reg_val_edge` which share the signature with `reg_val`. We generate one `reg_val_edge` per instruction and def-use predicate for the instructions whose behavior can be modeled in this domain and are used to compute an address (`def_used_for_address`). For example, Rule 12 below generates `reg_val_edge` predicates for `add` instructions that add a constant to a register:

```
reg_val_edge(A,Reg,Aprev,Reg,1,Imm):-
    def_used_for_address(Aprev,Reg),
    def_used(Aprev,Reg,A,_),
    instruction(A,_,_,'add',Op1,Op2,0,0),
    op_immediate(Op1,Imm),
    op_regdirect(Op2,Reg).
```
(12)

Example 4. Continuing with Example 3, the predicates `reg_val_edge` generated for the code in Fig. 2 are:

*P*1 `val_reg_edge(416C35,'RBX',416C35,'NONE',0,−624)`
*P*2 `val_reg_edge(416C58,'RBX',416C35,'RBX',1,24)`
*P*3 `val_reg_edge(416C58,'RBX',416C58,'RBX',1,24)`

Predicate *P*1 captures that `RBX` has a constant value after executing the instruction in address `416C35` (note that the multiplier is 0 and the register has a special value `'NONE'`). Predicate *P*2, generated from Rule 12, specifies that the value of `RBX` defined at address `416C58` corresponds to the value of `RBX` defined at `416C35` plus 24. Finally, *P*3 denotes that the value of `RBX` at `416C58` can be the result of incrementing the value of `RBX` defined at the same address by 24.

The set of predicates `reg_val_edge` can be seen as directed relational graph. The nodes in the graph are pairs of address and register `(A, Reg)` and the edges express relations between their values i.e. they are labeled with a multiplier and offset.

Once this graph is computed, we perform a propagation phase akin to a transitive closure. This propagation phase chains together `reg_val_edge` predicates. The chaining starts from the leafs of the graph (nodes with no incoming edges). Leafs in the `reg_val_edge` graph can be instructions that load a constant into a register such as **`mov RBX`**, `-624` in Fig. 2 or instructions where a register is assigned the result of an operation not supported by the domain. For example, loading a value from memory **`mov RDI`**, `[RIP+0x25D239]` in Fig. 2. In that case, the generated predicate would be the tautological predicate `reg_val(416C40,RBX,416C40,RBX,1,0)`.

In order to ensure termination and for efficiency reasons we limit the number of propagation steps by a constant `step_limit` with an additional field `S:`$\mathbb{Z}_{64}$ in the `reg_val` predicates. The main rule for combining `reg_val_edge` predicates is the following:

```
reg_val(A1,R1,A3,R3,M1*M2,(D2*M1)+D1,S+1):-
    reg_val(A2,R2,A3,R3,M2,D2,S),
    reg_val_edge(A1,R1,A2,R2,M1,D1), A1 != A2,
    step_limit(Limit), S+1 < Limit.
```
(13)

This rule chains edges linearly by combining their multipliers and displacements. It keeps track of operations that involve one source register and one destination register. However, we also want to detect situations where multiple edges converge into one instruction. Specifically, we want to detect loops and operations that involve multiple registers.

Detecting Simple Loops. The following rule (Rule 14) detects situations where a register `R` is initialized to a constant `D1`, then incremented/decremented in a loop by a constant `D2`.

```
reg_val(A,Reg,A2,'Unknown',D2,D1,S+1):-
    reg_val(A,R,A2,'NONE',0,D1,S),
    reg_val_edge(A,R,A,R,0,D2),
    step_limit(Limit), S+1 < Limit.
```
(14)

This pattern can be interpreted as `D1` being the base for a memory address and `D2` being the multiplier used to access different elements of a data structure. Our new multiplier `D2` does not actually multiply any real register, so we set the register field to a special value `'Unknown'`.

Example 5. Consider the propagation of the predicates in Example 4. The generated predicates are:

*P*4 `val_reg(416C35,'RBX',416C35,'NONE',0,−624)`
*P*5 `val_reg(416C58,'RBX',416C35,'NONE',0,−600)`
*P*6 `val_reg(416C58,'RBX',416C35,'Unknown',24,−600)`

First, predicate *P*4 is generated from *P*1 which is a leaf. Then, *P*4 is combined with *P*2 using Rule 13 into predicate *P*5. Finally, Rule 14 is applied to *P*5 and *P*3 to generate *P*6 which denotes that the register `RBX` takes values that start at −600 and are incremented in steps of 24 bytes.

Multiple Register Operations. In general, operations over two source registers cannot be expressed with `reg_val` predi-

cates. However, if one of the registers has a constant value or both registers can be expressed in terms of a third common register (a diamond pattern), we can propagate their value.

Example 6. The following assembly code contains a simple diamond pattern:

```
0: mov RBX, [RCX]
1: mov RAX, RBX
2: add RAX, RAX
3: add RAX, RBX
```

The last instruction adds the registers `RAX` and `RBX`. However, the value of `RAX` is two times the value of `RBX`. This is reflected in the predicates `reg_val(2,RAX,0,RBX,2,0)` and `reg_val(0,RBX,0,RBX,1,0)`. Therefore, we can generate a predicate `reg_val(3,RAX,0,RBX,3,0)`.

Note that the register value analysis intends to capture some of the relations between register values but it makes no attempt capture all of them. The goal of this analysis is not to obtain a sound over-approximation of the register values but to provide as much information as possible about how memory is accessed. The analysis is also not strictly an under-approximation as it is based on def-use chains which are over-approximating.

5.3 Data Access Pattern Analysis

The data access pattern (DAP) analysis takes the results of the register value analysis and the results of the def-use analysis to infer the register values at each of the data accesses and thus compute which addresses are accessed and which pattern is used to access them. The DAP analysis generates predicates of the form:

`data_access_pattern(A:`$\mathbb{A}$`,Size:`$\mathbb{Z}_{64}$`,Mult:`$\mathbb{Z}_{64}$`,From:`$\mathbb{A}$`)`

which specifies that address `A` is accessed from an instruction at address `From` and `Size` bytes are read or written. Moreover, the access uses a multiplier `Mult`.

Example 7. The code in Fig. 2 generates several DAPs:

*P*7 `data_access_pattern(673E80,8,0,416C40)`
*P*8 `data_access_pattern(45D0B8,8,0,416C47)`
*P*9 `data_access_pattern(45D0D0,8,24,416C47)`

The instruction at address `416C40` produces *P*7 which represents an access to a fixed address that reads 8 bytes. Conversely, the instruction at address `416C47` yields two predicates: *P*8 and *P*9. This is because register `RBX` can have multiple values at address `416C47`. If there are multiple DAPs to the same address, we choose the one with the highest multiplier.

These DAPs provide very sparse information, but if an address x is accessed with a multiplier m, it is likely that $x+m$, $x+2m$, etc., are also accessed the same way. Thus, we extend DAPs based on their multiplier. The analysis produces a predicate `propagated_data_access` with the same format as `data_access_pattern`. Our auxiliary analyses provide no information on what is the upper limit of an index in a data access. Thus, we simply propagate a DAP until it reaches the next DAP that coincides on the same address or that has a different multiplier. The idea behind this criterion is that the next data structure in the data section is probably accessed from somewhere in the code. So rather than trying to determine the size of the data structure being accessed, we assume that such data structure ends where the next one starts. These propagated DAPs will inform our symbolization heuristics.

Example 8. In our running example (Fig. 2) the DAP `data_access_pattern(45D0D0,8,24,416C40)` is propagated from address `45D0D0` up to address `45D310` in 24 byte intervals. The generated predicates are:

```
propagated_data_access(45D0D0,8,24,416C40)
propagated_data_access(45D0E8,8,24,416C40)
   ...                                 ...
propagated_data_access(45D310,8,24,416C40)
```

The DAP is not propagated to the next address `45D328` because that address contains another DAP generated at a different part of the code.

5.4 Discussion

There are two important aspects that set our register value analysis and DAP analysis apart from previous approaches like `Ramblr` [56].

First, the register value analysis is relational—it represents the value of one register at some location in terms of the value of another register at *another* location—in contrast to traditional value set analyses (VSA) [8]. This is also different from the affine-relations analysis [38] used in VSA analyses which computes relations between register values at the *same* location. A `reg_val` predicate between two registers also implies a data dependency i.e. a register is defined in terms of the other.

As a consequence, register value analysis can provide useful information (for our use-case) in many cases where obtaining a concrete value for a register would be challenging. Consider the code in Example 6. Our analysis concludes that at address 3 `RAX` is 3 times the value of `RBX` at address 0 regardless of what that value might be. In contrast, a traditional VSA analysis will only provide useful information for the value of `RAX` as long as it can precisely approximate the value of `RCX` and the values of all the possible memory locations pointed by `RCX`. If any of those locations has an imprecise abstract value e.g. $\top$, so will `RAX`.

Example 9. Let us consider a continuation of Example 6:

```
4: mov R8, QWORD PTR [RAX*8+0x1000]
5: mov R9, WORD PTR [RAX*8+0x1008]
6: mov R10, BYTE PTR [RAX*8+0x1010]
```

There will be DAPs for addresses `0x1000`, `0x1008` and `0x1010`

with sizes 8, 2, and 1 and a multiplier of 24 each. This information, though unsound in the general case (we are assuming `RAX` can take the value 0), is useful in practice.

These DAPs are the second distinguishing aspect of our analyses. `Ramblr` recognizes primitives and arrays of primitives. However, these DAPs indicate that address `0x1000` likely contains a `struct` with (at least) three fields of different sizes. Moreover, thanks to the multiplier and the `propagated_access_pattern` predicate we can conclude that address `0x1000` holds in fact an array of structs where the first field (at addresses `0x1000`, `0x1018`, `0x1030`...) has size 8 and might contain a pointer whereas the second and third fields (at addresses `0x1008`, `0x1020`, `0x1038`... and `0x1010`, `0x1028`, `0x1040`... respectively) have size 2 and 1 and thus are unlikely to hold a pointer.

6 Symbolization

The next step to obtain assembleable code is to perform symbolization. It consists of deciding for each constant in the code or in the data whether it is a literal or a symbol. A first approximation can be achieved by considering as symbols all numbers that fall within the range of the address space. However, as reported by Wang et al. [56], this leads to both false positives and false negatives. Next, we explain our approach to reduce the presence of false positives and negatives.

6.1 False Positives: Value Collisions

False positives are due to value collisions, literals that happen to coincide with range of possible addresses. In order to reduce the false positive rate, we require additional evidence in order to classify a number as a symbol.

6.1.1 Numbers in Data

For numbers in data, similarly to the approach used for blocks, we start by defining a set of "data object" candidates. Each candidate has an address, a type, and a size. We define data object candidates for the following types:

Symbol Whenever the number falls into the right range (`address_in_data`).

String A sequence of printable characters ended in 0.

Symbol-Symbol We detect jump tables using ad-hoc rules based on def-use chains, register values, and the DAPs computed in Sec. 5 (see App. A).

Other An address is accessed with a different size than the pointer size (8 bytes in x64 architecture) using the predicate `propagated_data_access` computed in Sec. 5.3.

We assign points to each of the candidates using heuristics based on the analyses results and detect if they are overlapping. If they are, we discard the candidate with fewer points. This process is analogous to how conflicts are resolved among basic blocks in Sec. 4.3. Note that detecting objects of type "String" and "Other" helps to discard false positives (i.e. symbol candidates) that overlap with them. As with blocks, we discard candidates if their total points fall below a threshold.

The main heuristics for data objects are (+ positive points and − for negative points):

+ **Pointer to instruction beginning:** A symbol candidate points to the beginning of an instruction. This heuristic relies on the results of the already computed IBI.
+ **Data access match:** The data object candidate is accessed from the code with the right size. This heuristic checks the existence of a `propagated_data_access` that matches the data object candidate's address and size.
+ **Symbol arrays:** There are several (at least 3) contiguous or evenly spaced symbol candidates. This indicates that they belong to the same data structure. Also, it is less likely to have several consecutive value collisions.
+ **Pointed by symbol array:** Multiple candidates of the same type pointed by a single symbol array.
+ **Aligned symbols:** A symbol candidate is located at an address with 8 bytes alignment.
+ **Strings:** A string candidate receives some points by default. If the string is longer that 5 bytes, it receives more points.
− **Access conflict:** There is some data access in the middle of a symbol candidate.
− **Pointer to special section:** A symbol candidate points to a location inside a special section such as `.eh_frame`.

6.1.2 Numbers in Code

We follow the same approach to disambiguate numbers in instruction operands. However, only the first and the last heuristics of the ones listed above, "Pointer to instruction beginning" and "Pointer to special section," are applicable to numbers in code. We distinguish two cases: numbers that represent immediate operands and numbers that represent a displacement in an indirect operand. After taking these two heuristics into account, we have not found false positives in displacements. For immediate operands we consider the following additional heuristics:

+ **Used for address:** The immediate is stored in a register used to compute an address (detected using predicate `def_used_for_address` from Sec. 5).
− **Uncommon pointer operation:** The immediate or the register where it is loaded is used in an operation uncommon for pointers such as `MUL` or `XOR`.
− **Compared to non-address:** The immediate is compared or moved to a register that in turn is compared to another immediate that cannot be an address.

These heuristics are tailored to the inference of how the immediate is used, and they rely on def-use chains and the results of the register value analysis.

6.2 False Negatives: Symbol+Constant

False negatives can occur in situations where the original code contains an expression of the form `symbol+constant`. In such cases, the binary under analysis contains the result of computing that expression.

There is no general procedure to recover the original expression in the code as that information is simply not present in the binary. Having a new symbol pointing to the result of the `symbol+constant` expression instead of the original expression is not a problem for rewrites which leave the data sections unmodified (even if the sections are moved) or rewrites that only add data to the beginning or the end of data sections. However, sometimes the resulting address of a `symbol+constant` expression falls outside the data section ranges or falls into the wrong data section. In such cases, a naive symbolization approach can result in false negatives.

We detect and correct these cases by detecting common patterns where compilers generate `symbol+constant` using the results of our def-use analysis and the register value analysis. We distinguish two cases: displacements in an indirect operands and immediate operands.

6.2.1 Displacements in Indirect Operands

For displacements in indirect operands, we know that the address that results from the indirect operand should be valid. Consider a generic data access `[R1+R2×M+D]` where `R1` and `R2` are registers, `M` is the multiplier and `D` the displacement. The displacement `D` might not fall onto a data section, but the expression `R1+R2×M+D` should.

Typically, in a data access as the one above, one of the addends represents a valid base address that points to the beginning of a data structure and the rest of the addends represent an offset into the data structure. In our generic access, `D` might be the base address, in which case it should be symbolic, or the base address might be in one of the registers, in which case `D` should not be symbolic.

We detect cases in which `D` should be symbolic even if it does not fall in the range of a data section. For example if the data access is of the form `[R2×M+D]` with $\texttt{M} > 1$, it is likely that `D` represents the base address and should be symbolic. We can detect less obvious cases with the help of the register value analysis (see Sec. 5.2). If we have a data access of the form `[R1+D]` but the value of `R1` can be expressed as the value of some other register `Ro` multiplied by a multiplier $\texttt{M} > 1$ (there is a predicate of the form `reg_val(_,R1,_,Ro,M,0))`, then `D` is also likely to be the base address and thus symbolic. On the other hand, if `R1` has a value that is a valid data address (there is a predicate `reg_val(_,R1,_,'NONE',0,A)` where `A` falls in a data section), then `D` is probably not a base address.

Knowing that a displacement should be symbolic is not enough, we need to infer the right data section to which the symbolic expression should refer. If the data access generates a DAP, we use the destination address of the DAP as a reference for creating the symbolic expression. Otherwise, we choose the closest boundary of a data section as a reference.

6.2.2 Immediate Operands

Having a symbolic immediate that falls outside the data sections is uncommon. The main pattern that we have identified is when the immediate is used as an initial value for a loop counter or as a loop bound to which the counter is compared.

Example 10. Consider the following code fragment taken from the program conflict-6.0 compiled with GCC 5.5 and optimization -O1. It presents an immediate of the form `symbol +constant` landing in a different section.

```
40109D:   mov EBX, 402D40
4010A2:   mov EBP, 402DE8
4010A7:   mov RCX,QWORD PTR [RBX]
 ...        ...
4010C5:   add RBX,8
4010C9:   cmp RBX,RBP
4010CC:   jne 4010A7
```

The number `402DE8` loaded at `4010A2` represents a loop bound and it is used in instruction `4010C9` to check if the end of the data structure has been reached. Address `402d40` is in section `.rodata` but address `402DE8` is in section `.eh_frame_hdr`.

We detect this and similar patterns by combining the information of the def-use analysis and the value analysis. We note that in these situations, the address that falls outside the section or on a different section and the address range of the correct section are within the distance of one multiplier. That is, let x be a candidate address that might represent the result of a `symbol+constant` expression, and let $[s_i, s_f)$ be the address range of the original symbol's section. Then $x \in [s_i - M, s_f + M]$ where M is the increment of the loop counter. Therefore, our detection mechanism generates an extended section range as above for every register that we identify as loop counter. Then, it checks if there is some immediate compared to the loop counter that falls within this extended range. If that happens, the immediate is rewritten using the base of the loop counter as a symbol.

Example 11. Example 10 continued. The register value analysis detects that `RBX` is a loop counter with a base address of `402D40` and a step size of `8`. Thus, we consider an extension of section `.rodata` to the range `[402718, 402DF0]` (the original address range is `[402720, 402DE8]`). Finally, using def-use chains we detect that the loop counter is compared to the immediate `402DE8` which falls within the extended section range. Consequently, we generate the following statement:

```
4010A2: mov EBP,OFFSET .L_402D40+168
```

where `.L_402D40` is a new symbol pointing to address `402D40`.

Program	Size	Program	Size	Program	Size	Program	Size	Program	Size
bar-1.11.0	91	bison-2.1	359	bool-0.2	48	conflict-6.0	28	doschk-1.1	18
ed-0.9	63	enscript-1.6.1	253	flex-2.5.4	196	gawk-3.1.5	485	gperf-3.0.3	409
grep-2.5.4	181	gzip-1.2.4	81	lighttpd-1.4.18	255	m4-1.4.4	154	make-3.80	202
marst-2.4	104	patch-2.6.1	155	re2c-0.13.5	2554	rsync-3.0.7	1685	sed-4.2	201
tar-1.29	547	tnef-1.4.7	74	units-1.85	65	wget-1.19.1	620	yasm-1.2.0	899

Table 1: Real world example benchmarks. Each program is annotated with its size in KB when compiled with GCC 7.1.0 and optimization flag -O0.

Benchmark	Binaries	Refs	Ddisasm				Ramblr			
			FP	FN	WS	Broken	FP	FN	Broken	Broken w/o ICC
Real world	1050	5957016	0	20	50	6	50258	62060	408	273
Coreutils	3710	4279339	3	0	0	3	8246	140774	752	323
CGC	2898	7220451	0	17	2	12	10892	43683	391	31

Table 2: Symbolization evaluation of `Ddisasm` and `Ramblr`. "Refs" represents the total number of references in these binaries; "FP" and "FN" list the number of false positives and false negatives respectively for each tool; "WS" lists the number of references pointing to the wrong section (only shown for `Ddisasm`); "Broken" lists the number of binaries that are broken (have at least one "FP," "FN" or "WS"). "Broken w/o ICC" lists broken binaries without counting the ones compiled with ICC.

7 Experimental Evaluation

We implemented our disassembly technique in a tool called `Ddisasm`. `Ddisasm` takes a binary and produces an IR called GrammaTech Intermediate Representation for Binaries (GTIRB) [45]. This representation can be printed to assembly code that can be directly reassembled. Currently `Ddisasm` only supports x64 Linux ELF binaries but we plan to extend it to support other architectures and binary formats. `Ddisasm` is predominantly implemented in Datalog (4336 non-empty LOC) which is compiled into highly efficient parallel C++ code using Souffle [28].

Benchmarks. We performed several experiments against a variety of benchmarks, compilers, and optimization flags. We selected 3 benchmarks. The first one is `Coreutils 8.25` which is composed of 106 binaries and has been used in the experimental evaluations of `Ramblr` [56] and Uroboros [57]. Programs in Coreutils are known to share a lot of code [5], so it is important to also consider other benchmarks. The second benchmark is a subset of the programs from the DARPA Cyber Grand Challenge (CGC). We adopt a modified version of these binaries that can be compiled for Linux systems in x64 [2]. We exclude programs that fail to compile or fail all their tests. That leaves 69 CGC programs. Finally, the third benchmark is a collection of 25 real world open source applications whose binary size ranges from 28 KB to 2.5 MB. Table 1 contains the names, version, and sizes (in KB) of the applications in the real world benchmark. Some of the original binaries in all benchmarks fail some tests. We take the results of the original binary as a baseline which rewritten binaries must match exactly—including failures.

Compilation Settings. For each of those programs we compile the binaries with 7 compilers: GCC 5.5.0, GCC 7.1.0, GCC 9.2.1, Clang 3.8.0, Clang 6.0, Clang 9.0.1, and ICC 19.0.5. For each compiler we use the following 6 compiler flags: -O0, -O1, -O2, -O3, -Os, and -Ofast. All programs are compiled as *position dependent code*[8].That means that for each original program we test 42 versions except for Coreutils where -Ofast generates original binaries that fail many of the tests and thus we skip it. In summary, we test 3710 different binaries for Coreutils, 2898 binaries for the CGC benchmark, and 1050 binaries from our real world selection. All benchmarks together represent a total of 888 MB of binaries. Note that the real world examples represent a significant portion of the binary data (324 MB).

7.1 Symbolization Experiments

We disassemble all the benchmarks and collect the number of false positives (FP) and false negatives (FN) in the symbolization procedure. We obtain ground truth by generating binaries with complete relocation information using the `-emit-relocs` ld linker option. We also detect an additional kind of error WS—i.e. when we create a symbolic expression, but the symbol points to the wrong section (see Sec. 6.2).

For comparison, we run the same experiments using `Ramblr`, the tool with the best published symbolization results. Table 2 contains the results of this experiment. Detailed tables with results broken down by compiler and optimization flag can be found in [22]. The complete set of binaries, detailed experiment logs, and the scripts to replicate the experiments can be found at [21].

[8]This is harder to disassemble than position *independent* code (PIC), which is though to be easier because it contains relocation information for absolute addresses [18]. Nonetheless, this does not make symbolization of PIC trivial as we argue in Sec. 7.1.

Benchmark	Binaries	Ddisasm			Ramblr			
		Disasm	Reassemble	Test	Disasm	Reassemble	Test	Test w/o ICC
Real world	1050	100.00%	100.00%	99.90%	99.62%	74.29%	39.90%	45.55%
Coreutils	3710	100.00%	100.00%	100.00%	99.27%	88.35%	71.26%	80.22%
CGC	2865	100.00%	99.93%	99.44%	100.00%	73.75%	51.24%	58.18%

Table 3: The functionality of binaries reassembled using Ddisasm and Ramblr as measured using the test suites distributed with the binaries. The "Disasm," "Reassemble," and "Test" (w/o ICC) columns list the percentage of binaries successfully disassembled, reassembled into a new binary, and that pass their original test suite (without counting binaries compiled with ICC) respectively.

Ddisasm presents a very low error rate. This shows the effectiveness of the approach. Ddisasm builds on many of the ideas implemented in Ramblr, but makes significant improvements (see Sec.5.4). App. B contains a discussion of Ddisasm's failures. Ramblr performs well on Coreutils and CGC compiled with GCC and Clang (in line with their experiments). 315 out of the 323 broken Coreutils binaries (without counting ICC) are broken due to a unique symbolization error in the binaries compiled with Clang 9.0.1. This illustrates the degree to which programs in Coreutils share code. Nonetheless, Ramblr's precision drops greatly against the real world examples (39% of broken examples) and binaries compiled with ICC (where all optimized binaries are broken). Additionally, we do not detect WS in Ramblr, as this information is not readily available. Thus, the numbers in the 'Broken' column are biased against Ddisasm as there might be binaries broken by Ramblr that are not counted.

It is worth pointing out that the ground truth extracted from relocations is incomplete for binaries compiled with ICC. This compiler generates jump tables with Symbol−Symbol entries. These jump tables do not need nor have relocations associated to them—even in PIC. We believe that this directly contradicts the claim made by Dinesh et al. [18] that x64 PIC code can be symbolized without heuristics—only using relocations.

The heuristics' weights for both IBI and symbolization have been manually set and work well generically across compilers and flags. Importantly, we fixed the weights before running the experiments on GCC 9.2.1 and Clang 9.0.1. Nonetheless, the results for these two compilers are on par with the results for the other compilers. Only 5 of a total of 21 broken binaries were compiled with GCC 9.2.1 or Clang 9.0.1. Thus, the heuristics's weights are robust across compiler versions. When ground truth can be obtained, these weights could be automatically learned and adjusted based on a program corpus, we leave that for future work.

Finally, we are interested in knowing the importance of different heuristics. Thus, we repeat the symbolization experiments for the real world benchmarks deactivating different kinds of heuristics. We deactivate heuristics that 1) detect strings, 2) heuristics that use DAPs ("Data access match" and "Access conflict"), and 3) both kinds at the same time. The results are in Table 4. Without both kinds of heuristics (row 3), we have a high number of FPs. Detecting strings (row 2) brings this number down, but we miss symbols that look like strings (FNs). DAPs give us additional evidence for those symbols through the "Data access match" heuristic. With DAPs but no strings (row 1), we also discard some FPs (by detecting objects of type "Other") but not all. The heuristics complement each other. Note that the 20 FNs produced by DAPs correspond to an array of structs that is correctly detected, but its pointer fields are accessed with size 4 instead of 8 which derails the analysis.

Heuristics	FP	FN	WS	Broken
No Strings	59	20	50	53
No DAP	45	43	50	49
No DAP & Strings	113	0	50	98

Table 4: Symbolization evaluation of Ddisasm on the real world benchmarks deactivating groups of heuristics.

7.2 Functionality Experiments

Using the same benchmarks we check how many of the disassembled binaries can be reassembled and how many of those pass their original test suites without errors.

For Ddisasm, we perform the experiment on the stripped versions of the binaries. Additionally, in order to increase our confidence that both IBI and symbolization are correct, we modify the locations (and relative locations) of all the instructions by adding NOPs at regular intervals before reassembling. We add 8 NOPs every 8 instructions to maintain the original instructions' alignment throughout the executable section[9]. We also add 64 zero bytes at the beginning of each data section. This demonstrates that our symbolization is robust to significant modification of code (by adding or removing code) and data (by adding content at the beginning of sections).

For Ramblr, we use unstripped binaries because Ramblr fails to produce reassembleable assembly for the stripped versions of most binaries. Many of the failures are because Ramblr generates assembly with undefined labels or with labels defined twice. This kind of inconsistency is easy to avoid in a Datalog implementation. Additionally, we do not

[9]We skip regions in between jump table entries of the form .byte Symbol−Symbol. Adding NOPs to these regions can easily make the result of Symbol−Symbol fall out of the range expressible with one byte.

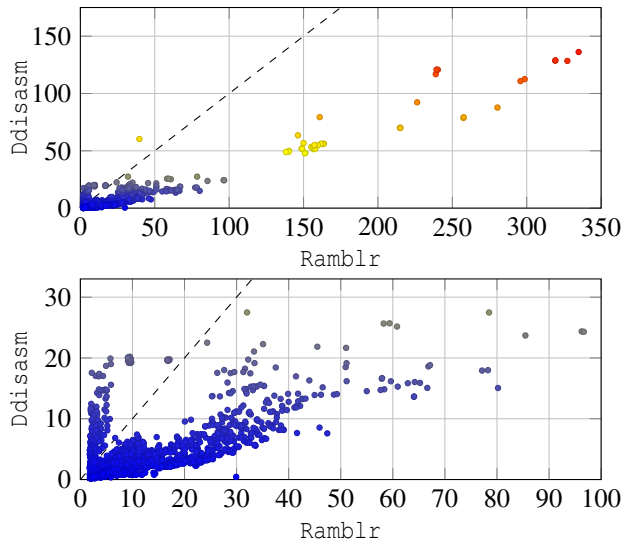

Figure 5: Disassembly time. The two graphs show the disassembly times (in seconds) for all the binaries at two different scales (the bottom graph displays smaller binaries in detail). `Ddisasm`'s disassembly time is plotted (vertically) against `Ramblr`'s (horizontally). In all graphs, points below the diagonal represent binaries for which `Ddisasm` is faster than `Ramblr`.

perform any modification of assembly generated by `Ramblr`—this ensures that we do not report an overly pessimistic result for `Ramblr` by accidentally breaking the code generated by `Ramblr`. So we compare `Ddisasm` at a significant handicap against `Ramblr`.

The results of this experiment are in Table 3. For CGC, we discarded 33 binaries that fail their tests non-deterministically leaving 2865 binaries. `Ddisasm` produces reassembleable assembly code for all the binaries but two. One binary in the real world benchmarks and 14 binaries in the CGC benchmark fail their tests. This is close to the results of our previous experiment (Table 2). The FNs in real world examples and 5 of the 17 FNs in CGC cause 1 and 5 test failures respectively. The remaining FPs, FNs, and WS symbols do not cause test failures. Additionally, there are 9 other test failures in CGC not caused by symbolization errors.

7.3 Performance Evaluation

Finally, we measure and compare the performance of both `Ramblr` and `Ddisasm`. We measure the time that it takes to disassemble each of the binaries in the three benchmarks. The results can be found in Fig. 5. `Ddisasm` is faster than `Ramblr` in all but 294 of 7658 total binaries. In particular, `Ddisasm` is on average 4.9 times faster than `Ramblr`.

8 Conclusion

We have developed a new reassembleable disassembler called `Ddisasm`. `Ddisasm` in implemented in Datalog and combines novel static analyses and heuristics to determine how data is accessed and used. We show that Datalog is well suited to this task as it enables the compositional and declarative specification of static analyses and heuristics, and it compiles them into a unified, parallel, and efficient executable.

`Ddisasm` is, to the best of our knowledge, the first disassembler for machine code implemented in Datalog. Our experiments show that `Ddisasm` is both more precise and faster than the state-of-the-art tools for reassembleable disassembly, and better handles large complex real-world programs. `Ddisasm` makes binary rewriting practical by enabling binary rewriting of real world programs compiled with a range of compilers and optimization levels with unprecedented speed and accuracy.

9 Acknowledgments

This material is based upon work supported by the Office of Naval Research under contract No. N68335-17-C-0700. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

References

[1] Cyber grand challenge (CGC). https://www.darpa.mil/program/cyber-grand-challenge.

[2] GrammaTech's CGC benchmarks. https://github.com/grammatech/cgc-cbs.

[3] Hex-rays: The IDA Pro disassembler and debugger. https://www.hex-rays.com/products/ida.

[4] National Security Agency. Ghidra, 2019. https://www.nsa.gov/resources/everyone/ghidra/.

[5] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, April 2017.

[6] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *The 25th USENIX Security Symposium*, pages 583–600, Austin, TX, 2016. USENIX Association.

[7] Cryptic Apps. Hopper. https://www.hopperapp.com/.

[8] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 01 2018.

[10] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In *The International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 16:1–16:10, New York, NY, USA, 2016. ACM.

[11] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'09, pages 243–262, NY, USA, 2009. ACM.

[12] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] David Brumley and James Newsome. Alias analysis for assembly. Technical report, Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.

[14] Xi Chen, Herbert Bos, and Cristiano Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *The 2017 IEEE European Symposium on Security and Privacy*, pages 514–529. IEEE, 2017.

[15] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *The 2015 Annual Network and Distributed System Security Symposium*, 2015.

[16] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: Binary component extraction and embedding for software security applications. In *European Symposium on Research in Computer Security*, pages 200–218. Springer, 2013.

[17] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to LLVM. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[18] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *The 41st Symposium on Security and Privacy*. IEEE, 2020. To Appear.

[19] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011.

[20] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict virtual call integrity checking for C++ binaries. In *2017 ACM on Asia Conference on Computer and Communications Security*, pages 140–154. ACM, 2017.

[21] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly: Artifact evaluation, February 2020. https://doi.org/10.5281/zenodo.3637587.

[22] Antonio Flores-Montoya and Eric M. Schulte. Datalog disassembly. *CoRR*, abs/1906.03969, 2019. http://arxiv.org/abs/1906.03969.

[23] Free Software Foundation. *GNU Binary Utilities*. Free Software Foundation, May 2002.

[24] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *ICSE*, 2019. To appear.

[25] Galois Inc. Open source binary analysis tools. https://github.com/GaloisInc/macaw.

[26] Vector 35 Inc. Binary ninja: a new kind of reversing platform. https://binary.ninja/.

[27] Software Engineering Institute. Automated static analysis tools for binary programs. https://github.com/cmu-sei/pharos.

[28] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[29] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. B2R2: Building an efficient front-endfor binary analysis. In *Binary Analysis Research (BAR), 2019*, 2019.

[30] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[31] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *The 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[32] James R Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *ACM Sigplan Notices*, volume 30, pages 291–300. ACM, 1995.

[33] Zephyr Software LLC. IRDB cookbook examples. https://git.zephyr-software.com/opensrc/irdb-cookbook-examples.

[34] Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, Milind Girkar, Hongjiu Lu, David Kreitzer, and Vyacheslav Zakharin. *System V Application Binary Interface: AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*, 2013.

[35] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *The 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[36] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *International Conference on Software Engineering (ICSE)*. ACM, 2019.

[37] Vishwath Mohan, Per Larsen, Stefan Brunthaler, K Hamlen, and Michael Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[38] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 330–341, New York, NY, USA, 2004. ACM.

[39] pancake. radare. https://www.radare.org/r/.

[40] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[41] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

[42] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.

[43] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196, Boston, MA, 1995. Springer US.

[44] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *USENIX Windows NT Workshop*, volume 1997, pages 1–8, 1997.

[45] Eric M. Schulte, Jonathan Dorn, Antonio Flores-Montoya, Aaron Ballman, and Tom Johnson. GTIRB: intermediate representation for binaries. *CoRR*, abs/1907.02859, 2019.

[46] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover C++ classes and methods from compiled executables. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 426–441, NY, USA, 2018. ACM.

[47] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.

[49] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *USENIX Annual Technical Conference*, pages 125–137, 2012.

[50] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 245–251, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[51] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, NY, USA, 2014. ACM.

[52] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 52–61. IEEE, 2013.

[53] Eli Tilevich and Yannis Smaragdakis. Binary refactoring: Improving code behind the scenes. In *The 27th international conference on Software engineering*, pages 264–273. ACM, 2005.

[54] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[55] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *The Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. IEEE, 2005.

[56] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[57] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, Washington, D.C., 2015. USENIX Association.

[58] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *The 28th Annual Computer Security Applications Conference*, pages 299–308. ACM, 2012.

[59] Richard Wartell, Yan Zhou, Kevin W Hamlen, and Murat Kantarcioglu. Shingled graph disassembly: Finding the undecideable path. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 273–285. Springer, 2014.

[60] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Programming Languages and Systems*, pages 97–118, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[61] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *The ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, NY, USA, 2004. ACM.

[62] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[63] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. In *The 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 129–140. ACM, 2014.

[64] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, pages 337–352, 2013.

A Symbol-Symbol Jump Tables

This appendix describes jump tables with relative offsets and how they are detected by our disassembler.

Most jump tables in programs compiled with GCC and Clang (position dependent code) are lists of absolute addresses that can be detected like any other symbolic value. This is not the case for jump tables generated by ICC and jump tables generated by PIC code. These jump tables are often expressed as lists of `Symbol−Symbol` expressions.

In this kind of jump tables, one of the symbols represents a reference point, and the other symbol represents the jump target. The reference point is the same for all the jump table entries and the actual value stored at each the jump table entry is the distance between the jump target and the reference point. The size of jump table entries can vary i.e. 1, 2, 4 or 8 bytes.

```
47DA7b:   lea RDX, QWORD PTR [RIP+.L_4A09F0]
47DA82:   movzx EDX, BYTE PTR [RDX+RCX*1]
47DA86:   lea RAX, QWORD PTR [RIP+.L_47DA93]
47DA8d:   add RAX,RDX
47DA90:   jmp RAX

4a09f0:   .byte .L_47DB3F-.L_47DA93
          .byte .L_47DB36-.L_47DA93
          .byte .L_47DB2B-.L_47DA93
          .byte .L_47DB20-.L_47DA93
```

Figure 6: Assembly (after symbolization) extracted from tar-1.29 compiled with ICC -O2. This code implements a jump table of `Symbol-Symbol` entries of size 1 byte.

Example 12. Consider the example code in Fig. 6. The first instruction loads the start address of the jump table onto `RDX`; the second instruction reads the jump table entry and stores it in `RDX`; the third instruction loads address `47DA93` that acts as a reference for the jump table onto `RAX`; the fourth instruction computes the jump target by adding `RAX` and `RDX` and the last instruction executes the jump.

In order to find a jump table, we need to determine: the jump table starting point, the jump table reference point, and the size of each jump table entry. Fortunately, the code patterns used to implement this kind of jump tables are relatively regular. We have specialized Datalog rules to detect them.

```
jump_table_start(AJump,Size,Start,Reference):-
    reg_jump(AJump,_),
    def_used(ASum,Reg,AJump,_),
    reg_reg_op(ASum,Reg,RegEntry,RegRef,1,0),

    def_used(AEntry,RegEntry,ASum,_),
    data_access_pattern(Start,Size,Size,AEntry),

    def_used(ARef,RegRef,ASum,_),
    reg_val(ARef,RegRef,_,'NONE',0,Reference).
```
(15)

Rule 15 is simplified version of the rule that detect the pattern in Fig. 6. The rule finds a jump that uses a register and "walks back" the code using def-use chains to the instruction where the jump target is computed (at address `Asum`). At that location, `reg_reg_op` represents an abstraction of an assembly instruction on two registers `Reg=RegEntry+RegRef`$\times$`1+0`. Then, the rule examines the definition of `RegEntry` to find where the jump table entry is read (at address `AEntry`) and thanks to its `data_access_pattern`, it determines the jump table starting address `Start` and the size of each entry `Size`. The other register `RegRef` should contain the jump table reference point. So its value is obtained using `reg_val` which should contain a constant value (not expressed in terms of another register).

By relying on the analyses presented in Sec. 5, i.e def-use chains, DAPs and the register value analysis; the Datalog rule is more robust than exact pattern matching. The instructions involved in the jump table do not necessarily appear all together or in a fixed order, and the rule does not rely on specific instructions being used. E.g. the jump target computation is sometimes done using `LEA` instead of `ADD`.

Once we have found the jump table beginning and its corresponding `data_access_pattern`, we can use the `propagated_data_access` (see Sec. 5.3) to create `symbol`$-$`symbol` candidates for each of the jump table entries. That means that we will consider that the jump table extends until there is another data access from a different part of the code.

The detection of these jump tables has been the main addition required to support the ICC compiler. Other analyses and heuristics have remained largely the same. We expect that supporting additional compilers will require similar additions as each compiler has its own particular code patterns. However, the analyses described in Sec. 5 remain useful building blocks that facilitate supporting these special constructs in a robust manner.

B Symbolization Failures

We manually examined and diagnosed `Ddisasm`'s symbolization failures to determine what are the causes that lead to the remaining FPs, FNs or WS.

Real world Benchmarks In the real world benchmarks, the 20 FPs corresponds to a single array of structs that contains pointers. Our analysis obtains the right DAP with the right multipliers but only 4 bytes of each of the pointers are read instead of 8. This leads `Ddisasm` to conclude that those locations contains data objects of type "Other" of size 4 instead of symbols. These FNs cause the corresponding tests to fail.

The 50 symbols pointing to the wrong section (WS) are displacements in indirect operands and happen in 5 variants of the same program (lighttpd-1.4.18) compiled with Clang 6.0 and Clang 9.0.1. These particular cases are not currently detected by our heuristics but they also do not cause test failures in our functionality experiments.

Coreutils Benchmarks In Coreutils, there are 3 FPs, all in binaries compiled with -O0. They correspond to immediate operands that are moved or compared to registers. Those registers are loaded from the stack immediately before the location of the immediate and they are stored in the stack again immediately after. Therefore, our analyses do not obtain any evidence on the type of those immediates. These FPs do not cause tests failures, probably because the Coreutils test suites are not exhaustive.

CGC Benchmarks In the CGC benchmarks, 5 of the 12 "Broken" binaries have FNs where the corresponding relocations refer to the symbols `__init_array_start` and `__init_array_end`. These binaries, compiled with ICC, do not have an `.init_array` section and in fact the symbols' addresses are the same and fall outside all data sections. Nonetheless, the code uses the difference between the two symbols (which is zero) and thus it has the same behavior even though these references have not been made symbolic. In fact, we do not observe test failures in these binaries.

There are 2 other binaries, variants of the same program compiled with ICC, that have displacements in an indirect operand pointing to the wrong section (WS). These particular cases are not currently detected by our heuristics. They also do not cause test failures.

Two variants of the same binary compiled with Clang 9.0.1 have a FN in an indirect operand. The symbol candidate points to the end of the `.rodata` section which coincides with the beginning of `.eh_frame_hdr`. This triggers the "Pointer to special section" heuristic which leads `Ddisasm` to incorrectly discard the symbol candidate. We plan to refine the "Pointer to special section" heuristic to avoid this corner case.

The 3 remaining failures are due to FN in variants of the same program compiled with GCC 7.1. They correspond to an immediate that should be a `Symbol+Constant`. The immediate is a loop bound but it corresponds to a triple nested loop that our heuristics do not detect well. The extended section considered is not large enough for the constant required by the immediate. These FPs cause the tests to fail.