Toward a *Best-of-Both-Worlds* Binary Disassembler

Stefan Nagy
January 2022

This past winter, I was fortunate to have the opportunity to work for Trail of Bits as a graduate student intern under the supervision of Peter Goodman and Artem Dinaburg. During my internship, I developed Dr. Disassembler, a Datalog-driven framework for transparent and mutable binary disassembly. Though this project is ongoing, this blog post introduces the high-level vision behind Dr. Disassembler's design and discusses the key implementation decisions central to our current prototype.

Introduction

Binary disassembly is surprisingly difficult. Many disassembly tasks (e.g., code/data disambiguation and function boundary detection) are *undecidable* and require meticulous heuristics and algorithms to cover the wide range of real-world binary semantics. An ideal disassembler has two key properties: (1) **transparency**, meaning that its underlying logic is accessible and interpretable, and (2) **mutability**, meaning that it permits ad hoc interaction and refinement. Unfortunately, despite the abundance of disassembly tools available today, none have both transparency *and* mutability. Most off-the-shelf disassemblers (e.g., objdump, Dyninst, McSema, and Angr) perform "run-and-done" disassembly, and while their underlying heuristics and algorithms are indeed open source, even the slightest of changes (e.g., toggling on a heuristic) requires a complete rebuild of the tool and regeneration of the disassembly. In contrast, popular commercial disassemblers like IDA Pro and Binary Ninja provide rich interfaces for user-written plugins, yet these tools are almost entirely proprietary, making it impossible to fully vet where their core heuristics and algorithms fall short. Thus, reverse engineers are left to choose between two classes of disassemblers: **those full of ambiguity or those with zero flexibility.**

In this blog post, I introduce our vision for a *best-of-both-worlds* (transparent *and* mutable) platform for binary disassembly. Our approach was inspired by recent disassembly tools like ddisasm and d3re, which use the Soufflé Datalog engine. Dr. Disassembler uses Trail of Bits' in-house incremental and differential Datalog engine, Dr. Lojekyll, to specify the disassembly process. Below, I describe how Dr. Disassembler's relational view of disassembly is a step toward transparent, mutable disassembly—streamlining the integration of new heuristics, algorithms, and retroactive updates—without the need to perform *de novo* disassembly per every incremental update.

Background: Disassembly, Datalog, and Dr. Lojekyll

Disassembly is the process of translating a binary executable from machine code into a human-interpretable, assembly language representation of the program. In software security, disassembly forms the backbone of many critical tasks such as binary analysis, static rewriting, and reverse engineering. At Trail of Bits, disassembly is the crucial first step in our executable-to-LLVM lifting efforts, such as Remill and McSema.

At a high level, a disassembler begins by first parsing a binary's logical sections to pinpoint those that contain executable code. From there, instruction decoding translates machine code into higher-level instruction semantics. This procedure uses one of two strategies: *linear sweep* or *recursive descent*.

Linear sweep disassemblers (e.g., objdump) perform instruction decoding on every possible byte, beginning at the very first byte index. However, on variable-length instruction set architectures like x86, a linear sweep disassembler that naively treats all bytes as instructions could perform instruction decoding on non-instruction bytes (e.g., inlined jump tables). To overcome this issue, many modern disassemblers improve their analyses by recovering metadata (e.g., debugging information) or applying data-driven heuristics (e.g., function entry patterns).

On the other hand, recursive descent disassemblers (e.g., IDA Pro) follow the observed control flow to selectively re-initiate linear sweep only on recovered branch target addresses. While recovering the target addresses of jump tables is generally sound, recovering the targets for indirect calls is a far more challenging problem, in which common-case soundness has yet to emerge.

Datalog is one of the more popular members in a class of programming languages known as *logical* programming. Compared to imperative programming languages (e.g., Python, Java, C, and C++), which are structured around a program's control flow and state, logical programming (e.g., Prolog and Datalog) is structured solely around logical *statements*. In our use case of binary disassembly, a logical statement can be useful for capturing the addresses in a binary that correspond to plausible function entry points: (1) targets of direct call instructions, (2) common function prologues, or (3) any function address contained in the symbol table. This use case is shown below in Dr. Lojekyll syntax:

```
#query plausible_function(free u64 FuncEA):
    ; Clause 1: targets of direct calls.
    transfer(FromAddr=_, ToAddr=FuncEA, TransferType=CALL_DIRECT).

    ; Clause 2: a common function prologue (push rbp, mov rbp rsp).
    instruction(Addr=FuncEA, Type=PUSH, Op1=RBP, Op2=None),
    instruction(Addr=FuncEA+1, Type=MOV, Op1=RBP, Op2=RSP).

    ; Clause 3: any recovered function symbol.
    symbol(Addr=FuncEA, SymbolType=Function).
```

Listing 1: This query retrieves the set of all plausible function entry points. Here, "`free`" denotes that the query must find all candidates that match the subsequent clauses. In a bounded clause (e.g., given some fixed address), the tag "`bound`" is used instead (see listing 5).

From a logical programming perspective, the code snippet above is interpreted as follows: there is a plausible function at address `FuncEA` if a direct call to `FuncEA`, a known function entry instruction sequence starting at `FuncEA`, or a function symbol at `FuncEA` exists.

At a higher level, logical and functional programming are part of a broader paradigm known as declarative programming. Unlike imperative languages (e.g., Python, Java, C, and C++), declarative languages dictate only what the output result *should* look like. For instance, in the previous example of retrieving function entry points, our main focus is the end result—the set of function entry points—and not the step-by-step computation needed to get there. While there is certainly more to logical and declarative programming than the condensed explanation offered here, **the key advantage of logical programming is its succinct representation of data as statements**.

Here's where Datalog shines. Suppose that after populating our database of "facts"—sections, functions, and instructions—we want to make some adjustments. For example, imagine we're analyzing a position-independent "hello world" binary with the following disassembly obtained for function `<main>`:

```
 523: 89 04 24                 mov   %eax,(%esp)
 526: e8 fc ff ff ff           call  527 <main+0x17>
 52b: 31 c9                    xor   %ecx,%ecx
```

Listing 2: An example of a relocated call target

We also know that the following relocation entries exist:

```
$ readelf -r helloworld.elf.x86_32.pie

Relocation section '.rel.dyn' at offset 0x308 contains 10 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000527  00000202 R_386_PC32        00000000   printf@GLIBC_2.0

Relocation section '.rel.plt' at offset 0x358 contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000200c  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
```

Listing 3: Relocation entry information for the example in listing 2

At runtime, the dynamic linker will update the operand of the call at `0x526` to point to `printf@PLT`. When the call is taken, `printf@PLT` then transfers to `printf`'s Global Offset Table (GOT) entry, and the execution proceeds to the external `printf`.

If you're familiar with IDA Pro or Binary Ninja, you'll recognize that both tools adjust the relocated calls to point to the external symbols themselves. In the context of binary analysis, this is useful because it “fixes up” the otherwise opaque calls whose targets are revealed only through dynamic linking. In Datalog, we can simply accommodate this with a few lines:

```
#export indirect_call_to_external(u64 CallEA, u64 ExternEA)
    : instruction(Address=CallEA, Type=CALL_INDIRECT, Op1=AddrOfTargetEA, _)
    , relocation(Address=AddrOfTargetEA, ExternalAddress=ExternEA)
    , instruction(Address=ExternEA, _, _, _).
```

Listing 4: This exported message rewrites the call to skip its intermediary Procedure Linkage Table (PLT) entry. Here, “`#export`” denotes that the message will alter some fact(s) in the Datalog database.

Voila! Our representation of the indirect call no longer requires the intermediary redirection through the PLT. As a bonus, we can maintain a relationship table to map every call of this type to its targets. With this example in mind, we envision many possibilities in which complex binary semantics are modelable through relationship tables (e.g., points-to analysis, branch target analysis, etc.) to make binary analysis more streamlined and human-interpretable.

Dr. Lojekyll is Trail of Bits’ new Datalog compiler and execution engine and the foundation on which Dr. Disassembler is built. It adopts a publish/subscribe model, in which Dr. Lojekyll-compiled programs “subscribe” to messages (e.g., *there exists an instruction at address X*). When messages are received, the program may then introduce new messages (e.g., *there exists a fall-through branch between instructions A and B*) or remove previous ones. Compiled programs may also publish messages to external parties (e.g., an independent server), which may then “query” data relationships from the Datalog side.

Dr. Lojekyll’s publish/subscribe model is well suited for tasks in which “undo”-like features are required. In binary disassembly, this opens up many possibilities in human-in-the-loop binary analysis and alterations (think *Compiler Explorer* but for binaries). At the time of writing, Dr. Lojekyll supports the compilation of Datalog into Python programs and has emerging support for C++.

Introducing Dr. Disassembler

Conventional “run-and-done” disassemblers perform their analyses on the fly, confining them to whatever results—even erroneous ones—are obtained from the outset. Instead, Datalog enables us to move all analysis to post-disassembly, thus streamlining the integration of plug-and-play refinements and retroactive updates. And with its painless syntax, Datalog easily represents one of the most powerful and expressive platforms for user-written disassembly plugins and extensions. We implement our vision of transparent and mutable disassembly as a prototype tool, Dr. Disassembler. While Dr. Disassembler can theoretically use any Datalog engine (e.g., DDLog), we currently use Trail of Bits’ own Dr. Lojekyll. The implementation of Dr. Disassembler discussed in this blog post uses Dr. Lojekyll’s Python API. However, at the time of writing, we have since begun developing a C++-based implementation due to Python’s many performance limitations. Here, I introduce the high-level design behind our initial (and forthcoming) implementations of Dr. Disassembler.

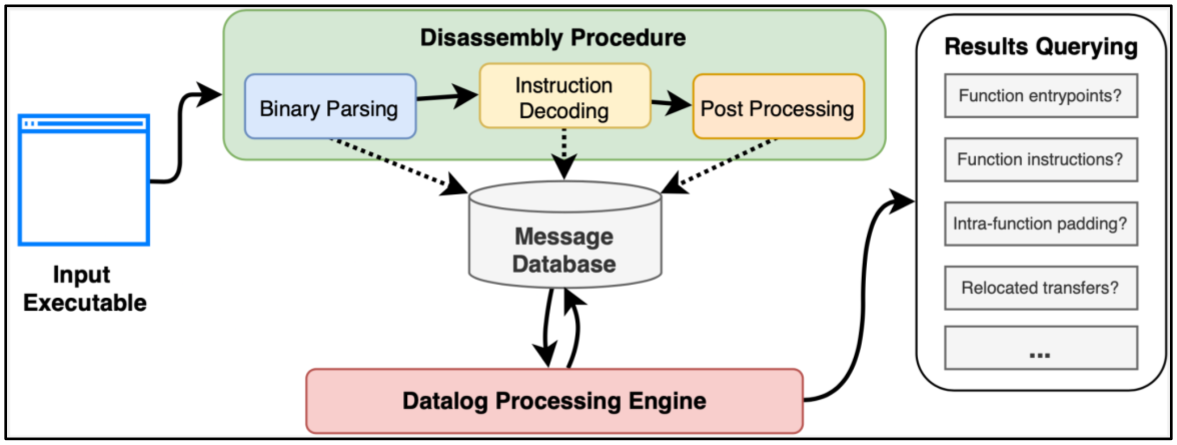

Figure 1: Dr. Disassembler's high-level architecture

Disassembly Procedure

Dr. Disassembler's disassembly workflow consists of three components: (1) parsing, (2) decoding, and (3) post-processing. In parsing, we scan the binary's sections to pinpoint those that contain instructions, along with any recoverable metadata (e.g., entry points, symbols, and imported/exported/local functions). For every identified code section, we begin decoding its bytes as instructions. Our instruction decoding process maps each instruction to two key fields: its type (e.g., call, jump, return, and everything else) and its outgoing edges.

Recovering Control Flow

An advantage of using Datalog is the ability to express complex program semantics as a series of simple, recursive relationships. Yet, when handling control flow, a purely recursive approach often breaks certain analyses like function boundary detection: recursive analysis will follow the control flow to each instruction's targets and resume the analysis from there. But, unlike calls, jumps are not "returning" instructions; so for inter-procedural jumps, the function will not be *re-entered*, thus causing the disassembler to *miss* the remaining instructions in the function containing the jump instruction.

To unify recursive and linear descent disassembly approaches, we developed the concept of non-control-flow successor instructions: for any unconditionally transferring jump or return instruction, we record an artificial fall-through edge from the instruction to the next sequential instruction. Though this edge has no bearing on the actual program, it effectively encodes the logical "next" instruction, thus unifying our linear and recursive analyses. These non-control-flow successor edges are the linchpin of our recursive analyses, like instruction-grouping and function boundary detection.

Post-Processing

At each step of parsing and decoding, we publish any interesting objects that we've found to our Dr. Lojekyll database. These core objects—symbols, sections, functions, instructions, and transfers—form the building blocks of our heuristic and recursive analyses. Our fundamental approach behind Dr. Disassembler is to "engulf" as much disassembly information as possible, regardless of correctness, and to refine everything afterward on the Datalog side. Because we consider every piece of information to be plausibly correct, we can retroactively update our disassembly when any new information is observed; and unlike conventional run-and-done tools, this does not require a de novo re-disassembly.

Example Exports and Queries

Dr. Disassembler streamlines binary analysis by focusing on disassembly artifacts themselves rather than the myriad steps needed to obtain them. To showcase some of Dr. Disassembler's many capabilities, this section highlights several implementation examples of rigorous binary analysis tasks facilitated by two of Dr. Disassembler's fundamental constructs: "exports" (messages that change/remove facts) and "queries" (which retrieve information about facts).

Query: Grouping Instructions into Functions

Given an arbitrary function address `FuncEA`, this query returns all the addresses of the instructions contained in that function. Two messages form this query: (1) `function(u64 StartEA)` and (2) `instruction(u64 InsnEA, type Type, bytes Bytes).`

```
#query function_instruction(bound u64 FuncEA, free u64 InsnEA).

; Clause 1: The first instruction of a function is always a member of
; that function.
function_instruction(FuncEA, FuncEA)
    : function(FuncEA)
    , instruction(FuncEA, _, _).

; Clause 2: If there is flow between two instructions in a function,
; then that flow's destination instruction is also in the function.
function_instruction(FuncEA, NextEA)
    : function_instruction(FuncEA, InsnEA)
    , intraprocdural_transfer(InsnEA, NextEA).
```

Listing 5: An example Dr. Disassembler query that returns the addresses of the instructions contained in a function

Export: Instructions Dominating Invalid Instructions

This export returns all the instructions whose control flow leads to *invalid* instructions (i.e., where instruction decoding fails). This heuristic is critical for Dr. Disassembler to filter-out the many “junk” instruction sequences that inevitably occur when decoding every possible byte sequence.

As in the previous example, we structure this relationship around two core messages: (1) `instruction` and (2) `raw_transfer(u64 StartEA, u64 DestEA)`, the latter of which contains the unaltered control flow recovered from the binary (i.e., no alterations like the one in listing 4 are made yet).

```
#export dominates_invalid_instruction(u64 EA).

; Clause 1: Fall-throughs to non-decodable instructions.
; Any transfer to a non-instruction at `EA` marks `EA`
; as dominating an invalid instruction.
dominates_invalid_instruction(EA)
    : raw_transfer(_, EA, _)
    , !instruction(EA, _, _).

; Clause 2: Recursive case for fall-throughs.
dominates_invalid_instruction(EA)
    : instruction(EA, INSN_NORMAL, _)
    , raw_transfer(EA, FallThroughEA, EDGE_FALL_THROUGH)
    , dominates_invalid_instruction(FallThroughEA).

; Clause 3: Recursive case for direct jumps.
dominates_invalid_instruction(EA)
    : instruction(EA, INSN_DIRECT_JUMP, _)
    , raw_transfer(EA, JmpTargetEA, EDGE_JUMP_TAKEN)
    , dominates_invalid_instruction(JmpTargetEA).

; Clause 4: Recursive case for conditional direct jumps.
dominates_invalid_instruction(EA)
    : instruction(EA, INSN_COND_DIRECT_JUMP, _)
    , raw_transfer(EA, TakenEA, EDGE_JUMP_TAKEN)
    , raw_transfer(EA, NotTakenEA, EDGE_JUMP_NOT_TAKEN)
    , dominates_invalid_instruction(TakenEA)
    , dominates_invalid_instruction(NotTakenEA).

; Clause 5: Recursive case for direct calls.
dominates_invalid_instruction(EA)
    : instruction(EA, INSN_DIRECT_CALL, _)
    , raw_transfer(EA, CalleeEA, EDGE_FUNCTION_CALL)
    , dominates_invalid_instruction(CalleeEA).
```

Listing 6: An example Dr. Disassembler export that updates the database with all the instructions whose control flow leads to invalid instructions

Export: Inter-Function Padding

This export returns all the instruction addresses that serve as "padding" *between* functions (e.g., NOPs that do not belong to any function). Here, we use the following messages: (1) `function`, (2) `section`, (3) `raw_transfer`, and (4) `basic_block(u64 BlockEA, u64 InsnEA).` Identifying inter-function padding is a crucial step to refining our function-instruction grouping.

```
#export inter_function_padding(EA).

; Base case for inter-function padding: An instruction that falls
; through to the beginning of a function, the beginning of a section,
; or the ending of a section, and that doesn't itself belong to any
; known basic block is treated as padding.
inter_function_padding(EA)
    : raw_transfer(EA, FuncEA, EDGE_FALL_THROUGH)
    , function(FuncEA)
    , !basic_block(_, EA).

inter_function_padding(EA)
    : raw_transfer(EA, SecStartEA, EDGE_FALL_THROUGH)
    , section(SecStartEA, _)
    , !basic_block(_, EA).

inter_function_padding(EA)
    : raw_transfer(EA, SecEndEA, EDGE_FALL_THROUGH)
    , section(_, SecEndEA)
    , !basic_block(_, EA).

; Inductive case for inter-function padding: An instruction that
; falls through to function  padding, where that instruction isn't
; part of any basic blocks, is also considered to be padding.
inter_function_padding(EA)
    : raw_transfer(EA, PaddingEA, EDGE_FALL_THROUGH)
    , inter_function_padding(PaddingEA)
    , !basic_block(_, EA).
```

Listing 7: An example Dr. Disassembler export that updates the database with all the instruction addresses that serve as "padding" between functions

Future Work and Extensions

Our immediate plan is to extend Dr. Disassembler to a fully C++ implementation. Along with improving the performance of the tool, we expect that this transition will open many new doors for research on binary analysis:

- **Streamlined binary analysis platforms**: Contemporary binary analysis platforms have rich interfaces for developing custom analysis plugins, but the sheer complexity of their APIs frequently leaves users bottlenecked by steep learning curves. As a next step, we

want to develop Dr. Disassembler into a full-fledged binary analysis platform, complete with all the features needed to facilitate the easy creation and customization of user plugins.
- **GUI interfaces for binary analysis and transformation**: By using Dr. Disassembler's mutable representation of disassembly, we can develop new interfaces that enable *real-time* analysis and editing of binary executables (e.g., to help developers visualize how toggling on different heuristics affects analysis results). Our end goal here is something akin to *Compiler Explorer*… for binaries!
- **Exposing analysis blind spots**: Our prototype of Dr. Disassembler is designed to use the outputs of multiple binary parsers and instruction decoders. Going forward, we would like to use Dr. Disassembler as a platform for developing automated techniques to identify where these competing tools agree and disagree with one another (e.g., on code-data disambiguation, branch target analysis, etc.) and pinpoint their weaknesses.

If any of these ideas interest you, feel free to get in touch with either me (Stefan Nagy) or Peter Goodman at Trail of Bits.

We'll release our prototype Python implementation of Dr. Disassembler and provide a PDF version of this post here. Happy disassembling!