Declarative Demand-Driven Reverse Engineering

Yihao Sun*, Jeffrey Ching†, and Kristopher Micinski‡
Department of Electical Engineering and Computer Science, Syracuse University
Email: *ysun67@syr.edu, †cching01@syr.edu, ‡kkmicins@syr.edu

***Abstract*—Binary reverse engineering is a challenging task because it often necessitates reasoning using both domain-specific knowledge (e.g., understanding entrypoint idioms common to an ABI) and logical inference (e.g., reconstructing interprocedural control flow). To help perform these tasks, reverse engineers often use toolkits (such as IDA Pro or Ghidra) that allow them to interactively explicate properties of binaries. We argue that deductive databases serve as a natural abstraction for interfacing between visualization-based binary analysis tools and high-performance logical inference engines that compute facts about binaries. In this paper, we present a vision for the future in which reverse engineers use a visualization-based tool to understand binaries while simultaneously querying a logical-inference engine to perform arbitrarily-complex deductive inference tasks. We call our vision declarative demand-driven reverse engineering (D^{3}RE for short), and sketch a formal semantics whose goal is to mediate interaction between a logical-inference engine (such Soufflé) and a reverse engineering tool. We describe a prototype tool, d3re, which are using to explore the D^{3}RE vision. While still a prototype, we have used d3re to reimplement several common querying tasks on binaries. Our evaluation demonstrates that d3re enables both better performance and more succinct implementation of these common RE tasks.**

I. Introduction

Binary reverse engineering (henceforth RE) is the process by which we start with some input binary (sequence of bytes) and employ various reasoning principles to explicate its behavior when executed as code. While RE tasks are often partially automated (e.g., via decompilation), full automation is often impossible: the extreme semantic expressivity afforded to binaries (including encrypted code, stripped symbol tables, etc..) often necessitates open-ended exploration and case-specific reasoning. Recent literature suggests that many practicioners follow an iterative approach involving several rounds of hypothesis formation and validation/falsification, often assisated via a combination of static and dynamic analysis [1]–[3].

To rapidly interact with a binary, RE practicioners often use reverse engineering tools such as Ghidra [4], IDA Pro [5], or Radare2 [6]. The goal of these tools is to allow an RE[1] to quickly explore the binary and visualize it (typically interactively, via a GUI or CLI) in a variety of ways. For example, a reverse engineer looking for a time bomb may first search for calls to the system's time function, and then walk backwards to understand whether each call is associated with legitimate or malicious behavior. In doing so, the RE may need to reason about, e.g., indirect control flow, or even identify the time function (in a stripped binary). Because REs are expert users, and often skilled programmers, RE tools provide programmatic interfaces that enable REs to systematize reasoning tasks via extensions. A broad range of popular extensions exist for several tools which perform such tasks as loading the results of static analyses [7], [8], interacting with debuggers [9], and identifying common cryptographic-relevant code [10].

In this paper, we argue that *deductive databases* (e.g., Datalog) serve as a natural abstraction boundary between RE tools and logical inference tasks over binaries. We envision a future in which a reverse engineer interactively explores a binary using an RE tool while simultaneously querying arbitrarily-complex logical properties written in a terse declarative style. We call this Declarative Demand-Driven Reverse Engineering (henceforth D^{3}RE). In D^{3}RE, an RE interacts with a deductive database by giving inputs (e.g., the currently highlighted address) to a rule-based deductive inference system written in a declarative language such as Datalog. Rules inductively compute *relations* over facts about the binary. As an example, consider a relation direct_call $\in$ *Addr* $\times$ *Addr* which relates callsite addresses (offsets within the binary) to procedure invocation target addresses. In our vision, D^{3}RE allows REs to interactively compute with and visualize the results of queries over these deductive rules.

We see D^{3}RE as a natural extension of several observations about the state of the start. First, many existing RE tools assemble databases to index various properties (e.g., addresses, symbols, etc...) of binaries for quick exploration. Deductive databases further allow REs to write arbitrary logical queries which are computed maximally efficiently via, e.g., compilation to relational algebra kernels as done in Soufflé. Deductive databases have also enabled several recent advances in binary analysis demonstrating both efficiency and robustness over conventional techniques. For example, the Datalog-based disassembler ddisasm achieves both faster and more-precise disassembly than other state-of-the-art disassemblers, and OOAnalyzer uses Prolog to enable declarative recovery of classes from compiled C++ code.

In this short paper we describe our progress in implementing a prototype tool, d3re, which we are building to realize the

[1]When unambiguous, we will use the term RE both to mean the process of reverse engineering and a reverse engineering practicioner

Network and Distributed Systems Security (NDSS) Symposium 2021
21 February 2021, Virtual
ISBN 1-891562-69-X
https://dx.doi.org/10.14722/bar.2021.23011
www.ndss-symposium.org

D^{3}RE vision. d3re allows REs to interactively define and calculate queries of arbitrary complexity over large production binaries and then visualize their results using Ghidra. To implement d3re we have designed an interface, which we call the *mediator*, that sits between a traditional Datalog solver and an RE tool. We briefly formalize this interaction between the RE tool and logic solver in Section III, and go on to describe our prototype Ghidra extension that enables visualizing the results of binary analyses in our tool. Using this formalism, we describe how d3re readily enables a broad range of binary analyses and sketch a vision for how we believe D^{3}RE will prove to be a natural ergonomic for reverse engineering.

We have measured the robustness of d3re in several ways. First, we wanted to know whether d3re could truly live up to our vision of being a natural replacement for the kinds of scripts REs already use in their day-to-day work. To evaluate this, we reimplemented a set of currently-existing Ghidra scripts. We happily observed that d3re was not only an ergonomic advantage (allowing us to write succinct but obviously-correct queries) but also a performance advantage. For example, many Ghidra scripts play tricks to avoid unnecessarily complexity that would arise in a straightforward implementation, e.g., iterating over a set of functions in a loop to check a property resulting in super-linear complexity. In d3re, the Datalog solver was naturally able to compile and organize work in an optimal way. We discuss this and other results in Section IV. We conclude with a brief overview of related work and our outlook on future directions in Section V

Specifically, we claim the following three contributions:

- A formalization of our metadatabase as a database of databases used to optimize subsequent invocations of the Datalog solver (Section III).
- A prototype tool, d3re, consisting of a server which wraps ddisasm with logic to enable chaining multiple subsequent calls via the metadatabase. Also included in d3re is an extension to the Ghidra RE toolkit to enable visualizing results computed using d3re.
- An evaluation of d3re on a set of benchmarks demonstrating positive initial results indicating that d3re could replace present-day binary analysis infrastructure (e.g., Ghidra scripts) and directly enable more efficient and succinct implementation.

II. Overview

In this section, we demonstrate the vision and application of D^{3}RE by illustrating how a reverse engineer might explicate a vulnerability due to an uninitialized global variable. We consider a particular binary, `CROMU_00038`, from DARPA's Cyber Grand Challenge which contains a function pointer which is uninitialized when an invalid flag is set in the metadata portion of an input file [11], [12]. We demonstrate how our prototype tool, d3re, can be used to build a declarative query to find uninitialized function entry points and visualize them within Ghidra. We do not claim that d3re can immediately or automatically discover vulnerabilities—in this section we try to focus on how its declarative reasoning instead enables rapidly exploring a binary to uncover some property.

```
1  // swap_short and swap_word only initialized
       within if
2  if (tiff_hdr->Byte_Order == 0x4949) {
3    printf("Intel formatted integers\n");
4    swap_word = intel_swap_word;
5  }
6  else if (tiff_hdr->Byte_Order == 0x4d4d) {
7    printf("Motorola formatted integers\n");
8    swap_word = motorola_swap_word;
9  }
10 #ifdef PATCHED
11 else {
12   printf("Invalid header values\n");
13   _terminate(-1);
14 }
15 #endif
16 // might cause an uninitialized variable bug here
17 offset = swap_word(tiff_hdr->Offset_to_IFD);
```

Fig. 1: Uninitialized variable vulnerability in CROMU0038 source code

```
>>> load dl/use_def_global.dl
>>> run dl/use_def_global.dl
>>> load dl/uninitialized.dl
>>> run dl/uninitialized.dl
>>> highlight
>>> comment
>>> query use_before_def_global
00004feb 0000a180 swap_short
00005017 0000a188 swap_word
...
0000515e 0000a180 swap_short
```

Fig. 2: d3re REPL session used in this overview.

The vulnerable segment of code is a use-before-definition bug shown in Figure 1. The swap_word function is initialized inside of the main function based on a value parsed in a TIFF header—if the flag does not match 0x4949 or 0x4d4d the function is left uninitialized and the call on line 17 crashes.

Loading the binary: To begin an analysis of a binary, an RE will load the binary into a reverse engineering tool. In our current implementation of d3re, a user opens two processes silmultaneously: a GUI-based instance of Ghidra, and a terminal running d3re's REPL. The user can explore the binary using all of the normal features of Ghidra and use all of its conventional analyses (e.g., to recover entrypoints). However, d3re's REPL communicates with Ghidra so that when d3re's analysis finishes Ghidra's views update as appropriate.

Initial processing: It is conventional that reverse engineering tools will apply a set of analyses to a binary to disassemble it and index various items such as entrypoints and callsites In d3re, the user builds queries in Datalog starting from a large initial set of Datalog rules that build on top of ddisasm, a Datalog-based disassembly engine [13]. Analogously to the indexing and analysis operations provided by Ghidra (and other RE tools), d3re invokes ddisasm once

```
def_global(EA,dest) :-
  code(EA), instruction_get_dest_op(EA,Index,_),
  pc_relative_operand(EA,Index,dest),
  defined_symbol(dest,_,"OBJECT","GLOBAL",_,_).

used_global(EA,dest,Index) :-
  code(EA), instruction_get_src_op(EA,Index,_),
  pc_relative_operand(EA,Index,dest),
  defined_symbol(dest,_,"OBJECT","GLOBAL",_,_).

def_used_global(EA_def,GA,EA_used,Index) :-
  used_global(EA_used,GA,Index),
  block_last_def_global(EA_used,EA_def,GA).

def_used_global(EA_def,GA, EA_used, Index) :-
  last_def_global(Block,EA_def,GA),
  code_in_block(EA_used, Block),
  used_global(EA_used, GA, Index),
  !block_last_def_global(EA_used,_,GA),.
```

Fig. 3: Global Var Use-Def analysis

to build an initial database.

Building on top of ddisasm was initially a strategic choice—ddisasm already includes facilities to parse object files and transform them into input databases in the style required by Soufflé. Initially, we extended ddisasm's set of rules with additional user-specific queries—a slow process, as ddisasm can take several minutes to run on large binaries. This was at odds with our goal of enabling rapid real-time feedback to users of d3re.

D^{3}RE builds upon a key observation that we have found crucial to enable efficient interactive binary analyses in practice: because Datalog is monotonic, we can evaluate an extended program (i.e., a program extended with a set of additional rules or queries) by using the database resulting from the calculation of the previous program. Thus, running ddisasm *once* allows pre-populating a large set of inferred relations for a wide range of interesting facts about binaries, including intraprocedural reachability and calling conventions.

When a binary is loaded, d3re invokes ddisasm with one slight modification: every Datalog relation in ddisasm's rule database (used by ddisasm to build a disassembly) is modified to be an output relation. In ddisasm, only dissassembly-relevant relations are output, rather than internal relations (e.g., those that relate to intraprocedural reachability). By marking all ddisasm's relations as output relations, d3re provides them to the user as primitives with which to build queries over binaries[2]. After the binary is loaded, all rules declared in ddisasm will be available for querying. Additionally, ddisasm will be run only once, even if the user uploads the same binary several times. All facts generated in this step will be stored in a temporary folder on disk managed by the metadatabase (described in Section III).

Designing a query to explicate use-before-define: In the D^{3}RE approach, REs interactively build queries to highlight various portions of the program that match certain properties.

[2]A relevant analogy might be that ddisasm is the standard library of d3re

```
use_before_def_global(EA_used,GA,Name) :-
  used_global(EA_used,GA,Index),
  !def_used_global(_,GA,EA_used,_),
  defined_symbol(GA,_,"OBJECT","GLOBAL",_,Name).

use_before_def_global(EA_used,GA,Name) :-
  used_global(EA_used,GA,Index),
  def_used_global(EA_def,GA,EA_used,_),
  !def_null_global(EA_def,GA),
  defined_symbol(GA,_,"OBJECT","GLOBAL",_,Name).
```

Fig. 4: uninitialized variables

They then manually inspect the results of their queries and use their intuition to build subsequent queries. Along the way, the RE may choose to add comments to various instructions, functions, or other forms and browse those instructions in Ghidra. In d3re, the communication between the logical rules and the state of the RE tool is reconciled by input and output tables—RE users can write queries that consume the state of the RE tool (such as currentAddress, the currently-selected address) as input relations, perform logical inference, and leave their output in relations such as comment(addr,"vuln").

Like ddisasm, d3re uses the Soufflé Datalog engine to perform logical inference over binaries. Users of d3re can incrementally build up more rules in the interactive REPL (shown in Figure 2). Currently, our REPL allows loading rules by loading new files—we plan on adding direct support for new rules, along with error-reporting feedback soon.

Knowing there was an uninitialized global function pointer being used, a user of d3re might first define a set of relations to build up def-use-chains of global variables. Datalog code to implement these queries is illustrated in Figure 3. The last two rules build up a relation def_used_global(EA_def,GA,EA_used,Index), which infers that at address EA_def, the global variable (at address GA) is defined and used at address EA_used at operand index Index. While this is a relatively coarse query, we envision the user could run the query on the binary to visualize a large answer set. In our setting, this can be done using the highlight or comment commands, which display the data marked to be highlighted by the most recent result computation.

Based on our definitions in Figure 3, we can define a relation for variables which are possibly used before they are defined. We demonstrate this in Figure 4. In the first rule, we say that if there is some usage of a global variable at some address, but in that address, we can't find any definition related to it, then we will consider variable there as an uninitialized variable; The second clause says that for some usage of a global variable even if it has some definition associated with it, if that definition is nullptr, we will still consider that there is a use-before-def vulnerability here.

Refining the query: In d3re, users can easily access the result of the rule and all facts generated by ddisasm through the GUI by writing into output tables using d3re rules. Unfortunately, our above query produces over 50

possible results—checking each occurrence would still be a timely endeavor. Next, we narrow down the query space to the range of just the main function. We use an auxiliary predicate, code_in_range, which we seed with constants for the beginning and end of the main function we gain from inspecting the binary in Ghidra.

```
code_in_range(19490,21704).
use_before_def_global(EA_used, GA, Name) :-
  code_in_range(from, to), EA_used >= from,
      EA_used < to,
  used_global(EA_used,GA,Index),
  !def_used_global(_,GA,EA_used,_),
  defined_symbol(GA,_,"OBJECT","GLOBAL",_,Name).
```

After new rules are applied, the output of the program becomes empty: however, this does not specify the program is free from the vulnerability. First, because of our constraint, only the main function is searched, bugs may still hide in other functions. Secondly, if all usage of a variable is before it's definition, null pointer error can still appear: programmers may initialize a variable to NULL and use several non-total branches to initialize the pointer, leaving the pointer uninitialized at the join point when no switch fires. We modify our rules to account for this:

```
def_null_global(EA,GA) :-
  def_global(EA,GA), instruction_get_src_op(EA,_,
      Op),
  op_immediate(Op,offset), offset=0.

use_before_def_global(EA_used, GA, Name) :-
  code_in_range(from,to),
  EA_used >= from, EA_used < to,
  used_global(EA_used,GA,Index),
  def_used_global(EA_def,GA,EA_used,_),
  !def_null_global(EA_def,GA),
  defined_symbol(GA,_,"OBJECT","GLOBAL",_,Name).
```

This change results in 19 addresses to search, and combining these results with use-def information in the previous step and intra-procedural control-flow graph in Ghidra, we can fairly easily infer that the global variable `swap_word` is initialized to 0 at address `0x4c2a`, that both conditional jumps `0x4f80` and `0x4fb8` fail, and observe a subsequent usage of `swap_word` at `0x5017` which will trigger a crash. At any stage in our process, we can sync Ghidra's UI with the current database using several REPL commands (an example is shown in Figure 5). In a fully-fledged implementation of d3re, we hope to have UI gadgets (or templates) to help users interactively build queries. For example, we may allow the user to select a region of the binary and build a rule that applies only to that region, or right-click on a function and build a rule specific to callers of that function. We believe this will need to be informed by a combination of interviews with expert users, participatory design, and (perhaps) user studies. This is work we plan to undertake now that we have proven initial success to ourselves with d3re.

We conclude this section by remarking upon the nature of our analyses. Our analyses would be considered naïve by the standards of industrial static analyses. Indeed, our reasoning is not even sound—we can restrict ourselves to looking at results for only one function or ignore complex behavior. Still, we believe that this iterative ad-hoc reasoning is a technique many reverse engineers already employ—the vision of D^{3}RE is to harmoniously leverage state-of-the-art deductive reasoning engines while performing human-guided RE tasks.

III. Design and Implementation

In this section, we present both a formal semantics for D^{3}RE and describe our implementation of d3re. The high-level architecture of d3re is outlined in Figure 6. Conceptually, the key idea of our semantics is to maintain a *metadatabase* to allow efficient incremental reuse of previously-computed databases. In d3re, this metadatabase takes the form of a server which accepts Datalog programs to run to a fixed-point.

The metadatabase (server) interacts with both the REPL process and Ghidra to render output databases into view annotations (e.g., highlights or comments) in the Ghidra UI based on REPL commands. Our visualization is currently limited to printing to Ghidra's console, highlighting a set of lines (typically some output relation), or annotating a line with a comment (whose contents may be dynamically determined via a Datalog query). We plan to investigate adding comments to other Ghidra UI elements (such as inferred classes) and other visual integration as future work.

A. *Formal semantics of* D^{3}RE

Due to space restrictions, we present only a sketch of a formal semantics for D^{3}RE. Semantics of Datalog programs are typically phrased in terms of an extensional database (EDB), an extensionally-enumerated set of ground facts, and intensional database (IDB), the set of rules defining the program [14]. Datalog's semantics is given by a least-fixed-point of an "immediate consequence" operator over the rules for the program. Because Datalog programs have a finite Herbrand

```
0000506e bf ff ff ff    MOV     EDI,0xffffffff
00005073 e8 db c0 ff ff CALL    _terminate

                        [d3re] : use_before_def_global >>>
                          00005078      0000a188      swap_word

                        LAB_00005078
00005078 48 8b 15 09 51 00 00  MOV     RDX,qword ptr [swap_word]
0000507f 48 8b 45 e8    MOV     RAX,qword ptr [RBP + local_20]
00005083 8b 40 04       MOV     EAX,dword ptr [RAX + 0x4]
00005086 89 c7          MOV     EDI,EAX
00005088 ff d2          CALL    RDX=>intel_swap_word
0000508a 89 c2          MOV     EDX,EAX
0000508c 48 8b 45 e8    MOV     RAX,qword ptr [RBP + local_20]
00005090 48 01 d0       ADD     RAX,RDX
00005093 48 89 45 d0    MOV     qword ptr [RBP + local_38],RAX
                        [d3re] : use_before_def_global >>>
                          00005097      0000a180      swap_short

00005097 48 8b 15 e2 50 00 00  MOV     RDX,qword ptr [swap_short]
0000509e 48 8b 45 d0    MOV     RAX,qword ptr [RBP + local_38]
000050a2 0f b7 00       MOVZX   EAX,word ptr [RAX]
000050a5 0f b7 c0       MOVZX   EAX,AX
000050a8 89 c7          MOV     EDI,EAX
```

Fig. 5: Ghidra with highlights and comments declaratively specified to output results inferred via d3re for our example.

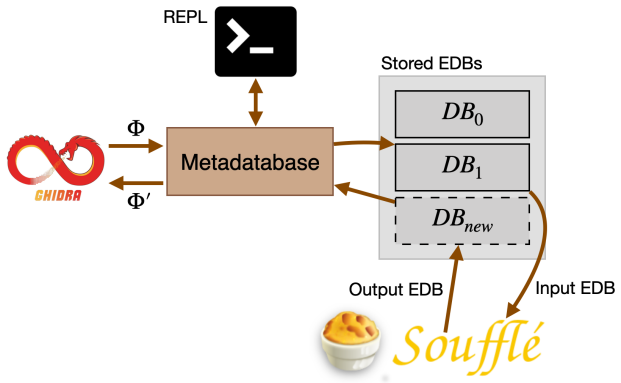

Fig. 6: High-level components and their interactions in d3re.

base (sets of atoms), this fixed-point necessarily exists (though in practice Datalog engines allow extra-logical behavior such as arithmetic). Datalog's conventional semantics is *monotonic*, in the sense that strictly more facts are accumulated as the fixed-point computation evolves—negation is allowed only when it may be stratified.

We define an *EDB metadatabase* as a graph of EDBs with labeled edges, $(\Delta, \xrightarrow{P})$, where Δ is a set of EDBs, each EDB enumerating tuples for a given set of relations, and $\xrightarrow{P}$ is a relation in $\Delta \times \mathit{Rules} \times \Delta$. When we process a program, P, using an input EDB, we traverse the graph $(\Delta, \xrightarrow{P})$ to find the most optimal, *compatible* EDB to start execution of P. Aided by Datalog's monotonicity, we define an EDB as compatible if it was produced by a subset of rules (or facts) from the input program / EDB. We conclude our formalism sketch by remarking that $(\Delta, \xrightarrow{P})$, given our usage, also forms a lattice.

B. Implementation of d3re

d3re is implemented in two parts: a REPL that communicates with Ghidra's GUI and a background service to manage the metadatabase and run the Datalog engine. The REPL currently communicates with Ghidra via a third-party extension named ghidra_bridge [15], which we plan to replace imminently with an extension using protocol buffers.

To execute a Datalog program P, d3re analyzes the file using the logic sketched in the above section to determine an optimal compatible EDB to use. In the common case, a user will gradually accumulate a stream of programs P, P', P''' consisting of a mix of rules and assumptions. In the future, we envision that certain assumptions (e.g., about calling conventions) may be implemented as GUI extensions rather than, e.g., manually-enumerated facts. After each run, the metadatabase will index the output facts and associate them with the program P, establishing an edge in the aforementioned graph. In our experiments, we refer to this as "caching."

IV. Evaluation

We evaluated d3re both qualitatively, by implementing several queries, and quantitatively by measuring its performance in benchmarks. While d3re is still a work in progress, we had several hypotheses we aimed to test as we designed and conducted these experiments. First, we wanted to understand whether d3re provided the necessary building blocks to enable replacing currently-existing Ghidra scripts. Second, we wanted to understand whether d3re could offer performance competitive with the kinds of Ghidra scripts that reverse engineers typically use. Last, we wanted to understand the performance of Ghidra for performing several repeated queries that might mirror a realistic end-to-end workload using d3re.

TABLE I: Script size (lines of code) of Ghidra script (Python) vs. d3re Datalog

	Ghidra Python	d3re Datalog
non-xor	33	8
basicblk	37	4
overflow	60	18
findcrypto	166	45

TABLE II: Running time of Ghidra scripts vs. equivalent implementation in d3re (all numbers in seconds).

	bison	souffle	gzip	re2c	redis	rsync
non-xor Ghidra	3.569	107.5	2.205	3.903	10.52	3.050
non-xor d3re	0.518	6.515	0.097	0.756	1.306	0.486
overflow Ghidra	0.370	0.247	0.600	0.240	0.760	0.180
overflow d3re	0.617	0.319	0.051	0.094	0.095	0.044
basicblk Ghidra	340.6	–	4.664	472.1	1806	107.4
basicblk d3re	0.539	7.13	0.094	0.812	1.433	0.571
findcrypt Ghidra	0.207	1.033	0.224	0.214	0.475	0.289
findcrypt d3re	1.287	14.53	0.224	1.701	2.938	1.186

Ghidra Script Replication Study: we wanted to determine whether d3re could realistically be used to accomplish the kinds of tasks that reverse engineers face on a day-to-day basis. This is an admittedly challenging question, which we plan to eventually evaluate in several ways including user studies. However, as initial work in this direction we arbitrarily selected four Ghidra scripts listed in the `awesome-ghidra` GitHub repository [16]. The scripts we chose are listed in Table I, along with their corresponding lines of code in Python / Datalog. While Ghidra scripts may consist of a mix of Python and Java, our experience is that most scripts use a small subset of the Python API. The first three are relatively small and find instructions that match a specific template, e.g., non-xor finds xor instructions that aren't zeroing registers, and overflow heuristically searches for potential overflows in calls to common functions such as strcpy. Our largest was findcrypto, which looks for common cryptographic constants.

Qualitative Results of our Replication Study: Our experience using d3re to replace Ghidra scripts must be understood in the context that we are expert users and the developers of d3re. However, we are pleasantly surprised that d3re enabled us to succinctly write equivalent implementations of each Ghidra script: we rewrote each script in substantially less Datalog code. This is because the declarative nature of Datalog

TABLE III: Runtime of successive invocations to d3re with (C) and without (S) rule caching.

	ddisasm	stack_var	heap_var	static_var	unl_static
souffle C	170	11.88	58.35	5.008	0.039
souffle S	170	11.79	66.02	67.00	66.52
bison C	7	0.932	1.409	0.545	0.022
bison S	7	0.934	1.916	2.122	2.075
re2c C	9	1.457	4.417	0.704	0.025
re2c S	9	1.494	5.257	5.449	5.458
redis C	11	1.918	2.544	1.302	0.025
redis S	11	1.919	3.525	3.712	3.726
rsync C	8	0.766	0.908	0.481	0.028
rsync S	8	0.783	1.325	1.423	1.384

eliminates the need for much of the conventional ceremony around, e.g., looping over instructions and checking against a type that we found in our evaluation scripts. Key to D^{3}RE's success, we believe, is its ability to directly use relations from ddisasm: we found that much of the necessary work of, e.g., filtering instructions by their type or operand was very useful at achieving succinct Datalog in practice. We are in the initial planning stages of developing a reverse engineering tutorial (or mini-course) around d3re, and are hoping to use this to recruit developers to get more realistic assessment of d3re's usability by professional REs.

Quantitative Results of our Replication Study: We hoped that d3re, being based on a high-performance Datalog solver, would offer performance competitive with Ghidra's current scripts. Each of our evaluation scripts processed the entire binary and would highlight or label certain instructions. To test the Ghidra scripts, we used Python's standard time function before and after the script's work finished. We evaluated the corresponding Datalog program by using Soufflé's internal performance timers. We then benchmark Ghidra vs. d3re on a corpus of six binaries (all sized less than 10MB), five from ddisasm's test suite and Soufflé, shown at the top of Table II. We used the latest versions of each pre-built in the latest Arch Linux, but we used a pre-built version of Soufflé. For each script, we waited for all of Ghidra's typical analyses to finish, and similarly we ran ddisasm to build up the initial input database for d3re.

The body of Table II compares the runtime of each Ghidra script versus its corresponding implementation in d3re. The single occurrence of – indicates that Ghidra did not finish within an hour. Broadly, we found that d3re outperformed Ghidra for each of the scripts in our replication study. As we had hoped, d3re's design allowed us to leverage useful relations from ddisasm. We found that many scripts do things like naive loops over sets of functions or symbols to locate some property. By contrast, the declarative style of d3re allowed us to write these not only more succinctly (e.g., Datalog naturally aggregates results) but also more efficiently—Soufflé optimally compiles input programs to efficient relational algebra kernels that loop only when necessary. We did observe various ways in which d3re's limitations could cause performance issues. For example, the findcrypto script scans the binary for 256-segments of code. d3re is built on Soufflé, which supports 64-bit primitive ints, but not 256-byte sequences. Thus, we had to build up sequences via a set of Datalog rules, causing inefficient memory representation due to the necessary duplication due to representing subsequences as Datalog facts.

Evaluating End-to-End Behavior in Subsequent Invocations: To understand the effect of caching via repeated calls to d3re, we ran four subsequent analysis queries in a row using both our caching-based approach and without caching (wherein we started only with the results of ddisasm). Our results are shown in Table III: the time of the cached run (C) is shown above the time for the correspond sequential run (S). As each query builds on the previous, we expect caching to reduce the amount of work and commensurately reduce the runtime. stack_var finds stack-allocated variables, while heap_var calculates stack variables holding pointers to heap values based on stack_var. static_var and unl_static attempt to find uninitialized global variables. Overall, we found rule caching was especially important on larger binaries versus sequential runs, justifying our choice to structure the metadatabase as a graph.

V. Related Work and Conclusion

We conclude with a brief discussion of proximately-related work that lies at the synthesis of reverse engineering and static / dynamic analysis, and contextualize this work in terms of our aspirations for the future of D^{3}RE. There has been extensive work using logic programming, and in particular Datalog, for static analysis of higher-level langauges such as Java [17]–[19]. The success of the Soufflé Datalog engine has inspired recent adoption of logic programming within the binary analysis community. For example, Datalog Disassembly uses Soufflé to achieve both faster and more-precise disassembly than the state-of-the-art disassembler Ramblr [13]. Similarly, OOAnalyzer uses XSB-Prolog, a version of Prolog implemented as a library [20]. We are currently reimplementing OOAnalyzer in d3re targeting C++ Linux binaries. We feel particularly excited about this direction because we believe Soufflé will be immediately more scalable than XSB-Prolog.

While there are a broad range of plugins for Ghidra and IDA Pro to load the *results* of static analyses, we believe d3re is the first to focus on the combination of open-ended deductive logical inference and rapid interactivity (enabled by our metadatabase). We believe the most closely related work is Ponce [21], which enables GUI-based symbolic execution. We plan to integrate symbolic execution into d3re as a long-term goal, inspired by the recent work of Formulog [22].

Our goal in this work was to introduce a new vision for reverse engineering, D^{3}RE, wherein expert users rapidly query high-performance logical inference engines to help them accomplish their day-to-day work in RE, vulnerability construction, and penetration testing. Visualization-based tools such as Ghidra are of immense value in understanding a binary, but have fundamentally different design considerations than high-performance logical inference enginges (such as Soufflé).

Recent work in compiling Datalog to parallel relational algebra (e.g., Gilray et al. [23]) has enabled a new frontier in scale of Datalog-based analyses. We hope that developments such as these will someday enable realizing fully the vision of D^3RE to help reverse engineers perform powerful static binary analyses at unprecedented scale.

REFERENCES

[1] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *USENIX Security 2020*, pp. 1875–1892, 2020.

[2] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. Richter Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," pp. 248–259, 08 2015.

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," pp. 672–681, 05 2013.

[4] "Ghidra released by national security agency." https://ghidra-sre.org/.

[5] Hexray, *Hex-rays:The IDA Pro disassembler and debugger.*

[6] "Radare2." https://github.com/radareorg/radare2.

[7] E. Schulte, J. Dorn, A. Flores-Montoya, A. Ballman, and T. Johnson, "Gtirb: Intermediate representation for binaries," 07 2019.

[8] Grammatech, "Gtirb." https://github.com/GrammaTech/gtirb-ghidra-plugin.

[9] "ret-sync." https://github.com/bootleg/ret-sync.

[10] "py-findcrypt-ghidra." https://github.com/AllsafeCyberSecurity/py-findcrypt-ghidra.

[11] "Grammatech's cyber grand challenge program repository." https://github.com/GrammaTech/cgc-cbs. Accessed: 2020-01-10.

[12] "Qualifier challenge - cromu_00038." https://github.com/GrammaTech/cgc-cbs/tree/master/cqe-challenges/CROMU_00038. Accessed: 2020-01-10.

[13] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[14] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[15] "Ghidra bridge." https://github.com/justfoxing/ghidra_bridge. Accessed: 2020-01-10.

[16] "Awesome ghidra." https://github.com/AllsafeCyberSecurity/awesome-ghidra.

[17] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 485–495, 2014.

[18] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*, pp. 422–430, Springer, 2016.

[19] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, (New York, NY, USA), pp. 196–206, Association for Computing Machinery, 2016.

[20] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using logic programming to recover c++ classes and methods from compiled executables," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), pp. 426–441, Association for Computing Machinery, 2018.

[21] "Ponce (ida pro plugin)." https://github.com/illera88/Ponce. Accessed: 2020-01-10.

[22] A. Bembenek, M. Greenberg, and S. Chong, "Formulog: Datalog for smt-based static analysis," *Proc. ACM Program. Lang.*, vol. 4, Nov. 2020.

[23] T. Gilray and S. Kumar, "Distributed relational algebra at scale," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 12–22, 2019.