Declarative, Demand-Driven RE

Yihao Sun, Jeffrey Ching, and Kris Micinski
Syracuse University

- Motivation and Background

- Motivation and Background
- D3RE Design and Methodology

- Motivation and Background
- D3RE Design and Methodology
- Demo using D3RE to perform an RE task

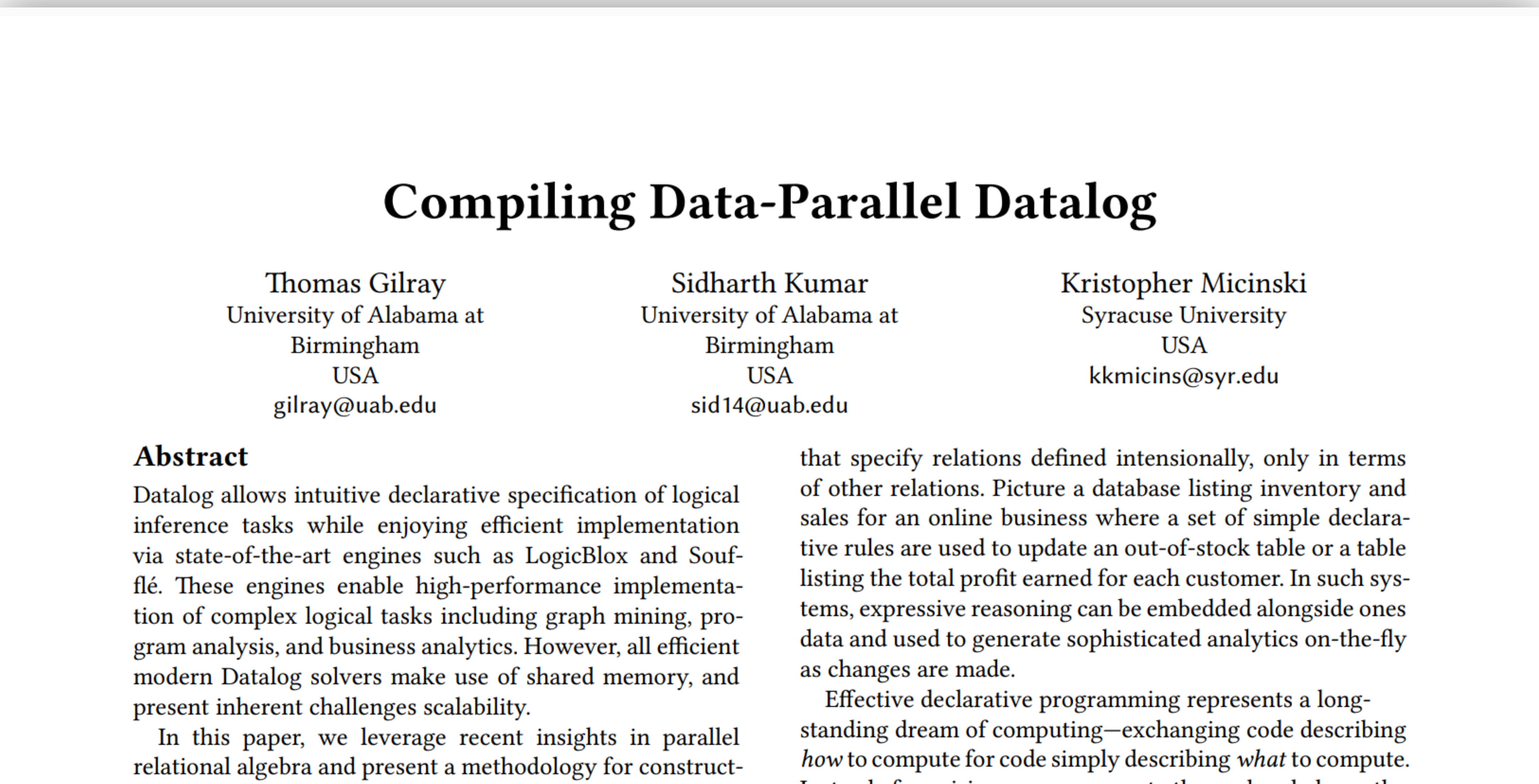
- Motivation and Background
- D3RE Design and Methodology
- Demo using D3RE to perform an RE task
- Implementation

- Motivation and Background
- D3RE Design and Methodology
- Demo using D3RE to perform an RE task

- Motivation and Background
- D3RE Design and Methodology
- Demo using D3RE to perform an RE task
- Implementation
- Evaluation
- Reflection and Future Work

- D3RE only a prototype
- Clear limitations: data transfer, etc…
- Future: parallel relational-algebra

- D3RE only a prototype
- Clear limitations: data transfer, etc…
- Future: parallel relational-algebra



Compiling Data-Parallel Datalog

Thomas Gilray
University of Alabama at Birmingham
USA
gilray@uab.edu

Sidharth Kumar
University of Alabama at Birmingham
USA
sid14@uab.edu

Kristopher Micinski
Syracuse University
USA
kkmicins@syr.edu

Abstract

Datalog allows intuitive declarative specification of logical inference tasks while enjoying efficient implementation via state-of-the-art engines such as LogicBlox and Soufflé. These engines enable high-performance implementation of complex logical tasks including graph mining, program analysis, and business analytics. However, all efficient modern Datalog solvers make use of shared memory, and present inherent challenges scalability.

In this paper, we leverage recent insights in parallel relational algebra and present a methodology for construct-

that specify relations defined intensionally, only in terms of other relations. Picture a database listing inventory and sales for an online business where a set of simple declarative rules are used to update an out-of-stock table or a table listing the total profit earned for each customer. In such systems, expressive reasoning can be embedded alongside ones data and used to generate sophisticated analytics on-the-fly as changes are made.

Effective declarative programming represents a long-standing dream of computing—exchanging code describing *how* to compute for code simply describing *what* to compute.

- D3RE only a prototype
- Clear limitations: data transfer, etc…
- Future: parallel relational-algebra
- Also: user studies for GUI
  - Relevant related work: Ponce

- Motivation and Background
- D3RE Design and Methodology
- Demo using D3RE to perform an RE task
- Implementation
- Evaluation
- Reflection and Future Work
- Wrap-up / conclusion

- Prior work shows REs take an iterative approach:
  - Overview
  - Subcomponent Scanning
  - Targeted Exploration

Unfortunately, **no tool supports every phase**, so REs frequently swap between tools

An Observational Investigation of Reverse Engineers' Processes

Daniel Votipka, Seth M. Rabin, Kristopher Micinski*,
Jeffrey S. Foster[†], and Michelle M. Mazurek
*University of Maryland; *Syracuse University; [†]Tufts University*
{dvotipka,srabin,mmazurek}@cs.umd.edu; kkmicins@syr.edu; jfoster@cs.tufts.edu

Abstract

Reverse engineering is a complex process essential to software-security tasks such as vulnerability discovery and malware analysis. Significant research and engineering effort has gone into developing tools to support reverse engineers. However, little work has been done to understand the way reverse engineers think when analyzing programs, leaving tool developers to make interface design decisions based only on intuition.

This paper takes a first step toward a better understanding of reverse engineers' processes, with the goal of producing insights for improving interaction design for reverse engineering tools. We present the results of a semi-structured, observational interview study of reverse engineers (N=16).

Researchers, companies, and practitioners have developed an extensive array of tools to support RE [5–24]. However, there is limited theoretical understanding of the RE process itself. While existing tools are quite useful, design decisions are currently ad-hoc and based on each designer's personal experience. With a more rigorous and structured theory of REs' processes, habits, and mental models, we believe existing tools could be refined, and even better tools could be developed. This follows from recommended design principles for tools supporting complex, exploratory tasks, in which the designer should "pursue the goal of having the computer vanish" [25, pg. 19-22].

In contrast to RE, there is significant theoretical understanding of more traditional program comprehension—how devel-

A typical RE may use:

- IDA to disassemble / explore code
- Case-specific plugins (e.g., finding crypto)
- Decompiler when possible (sometimes impossible)
- angr for symbolic execution
- BAP to query properties

```
push    edi
push    edi
push    edi
push    offset sub_10001074
jmp     short loc_1000D0BD
```

```
push    edi                     ; lpThreadId
push    edi                     ; dwCreationFla
push    edi                     ; lpParameter
push    offset sub_10001365 ; lpStartAd
```

```
loc_1000D0BD:                   ; dwStackSize
push    edi
push    edi                     ; lpThreadAttributes
call    ebx ; CreateThread
mov     ds:hThread, eax
```

Functions window

Function name	Segment	Start
sub_100110FF	.text	100110FF
GetModuleFileNameExA	.text	100111A6
EnumProcessModules	.text	100111AC
GetAdaptersInfo	.text	100111B2
Process32Next	.text	100111B8
Process32First	.text	100111BE
CreateToolhelp32Snapshot	.text	100111C4
Module32Next	.text	100111CA
Module32First	.text	100111D0

Line 1 of 346

Name	Address
PSLIST	10007025
nullsub_1	1000706F
nullsub_2	1000707C
StartEXS	10007ECB
HandlerProc	1000C9DF

Line 4 of 764

Strings window

Address	Length	T...	String
.rdata:1...	0000000E	C	GetDeviceCaps
.rdata:1...	0000000A	C	CreateDCA
.rdata:1...	00000011	C	CreateDIBSection
.rdata:1...	0000000A	C	SetBkMode

Imports

Address	Ordinal	Name	Library
10016000		LookupPrivilegeValueA	ADVAPI32
10016004		OpenProcessToken	ADVAPI32

Exports

The D^{3}RE vision

D^{3}RE: Declarative, Demand-Driven Reverse Engineering

Our goal: enable arbitrarily-complex binary analysis that can be selectively applied to segments of the binary

- Make it **fast** by implementing via Datalog (Soufflé)
- Make it **useful** by basing on ddisasm (Datalog Dissassembly)
- Make it **interactive** by hooking into Ghidra

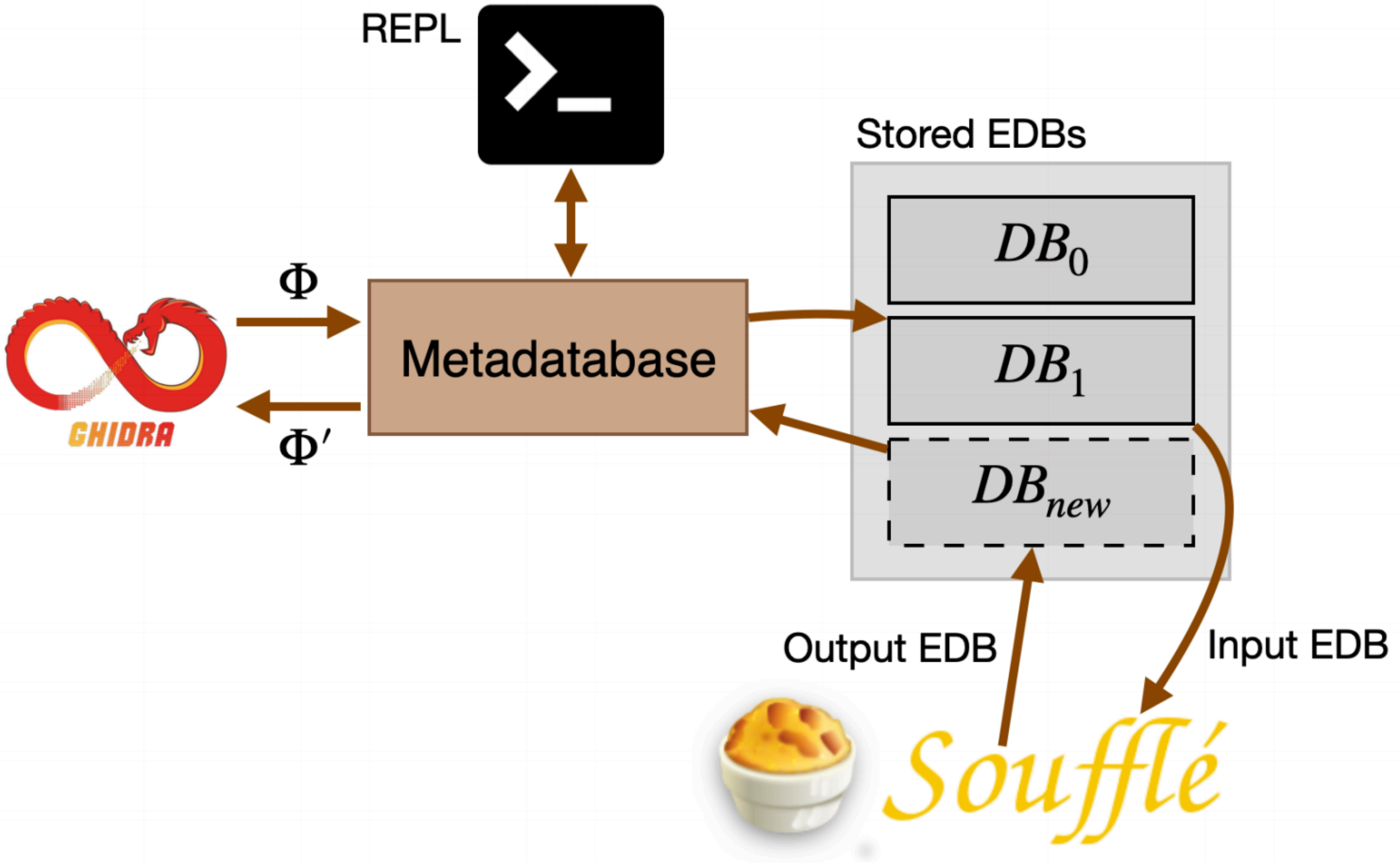
- Currently: REs swap between IDA/Ghidra/… and case-specific analysis tools
- Plugins: more interactive, but harder to scale
- IDA (/Ghidra/r2/…) AST not designed to enable high-performance binary analysis
  - ➡ Also, lacking facilities for analysis parallelism etc…
- In D^{3}RE, user writes declarative **rules** in Datalog

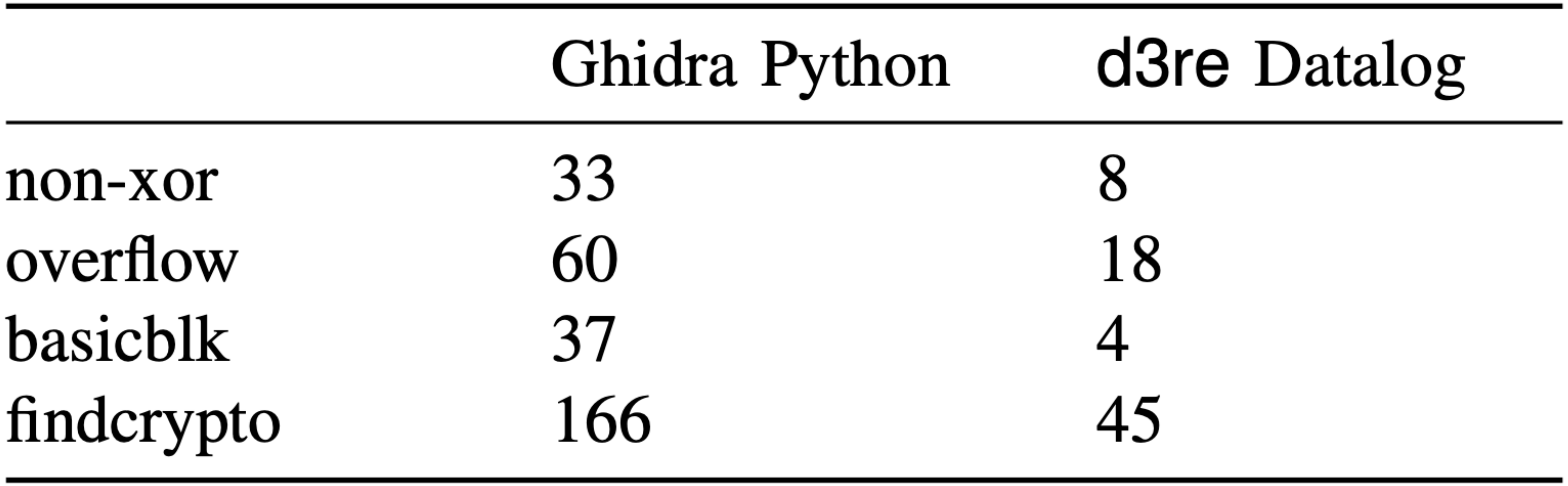
"An instruction address EA is a move when it is code and its direct operand contains RAX"

```
mov_rax(EA) :-
    code(EA),
    instruction_get_src_op(EA,_,Op),
    op_regdirect_contains_reg(Op,"RAX").
```

To use d3re (our tool), a user loads a binary into Ghidra and also processes the binary using **ddisasm**

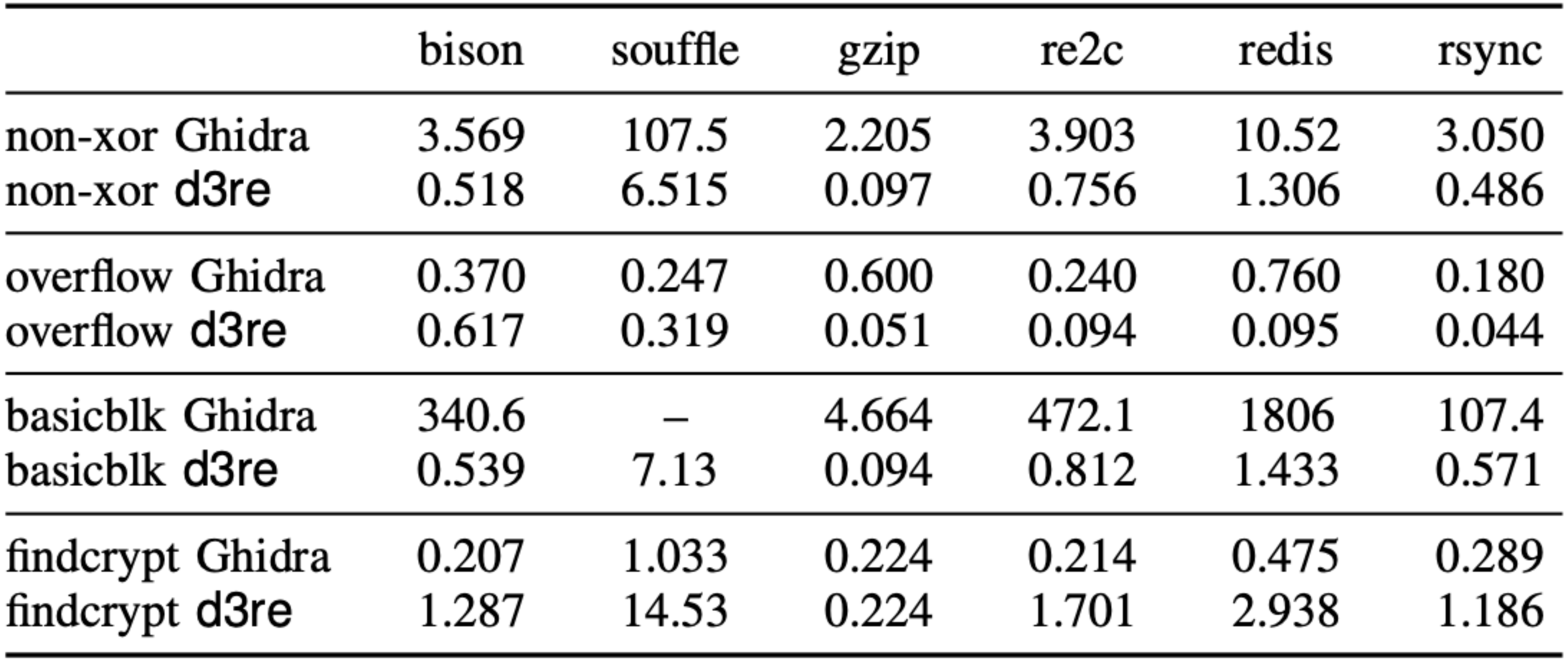
User can then interactively add additional rules (using the REPL, long-term will replace with GUI) and visualize them via Ghidra





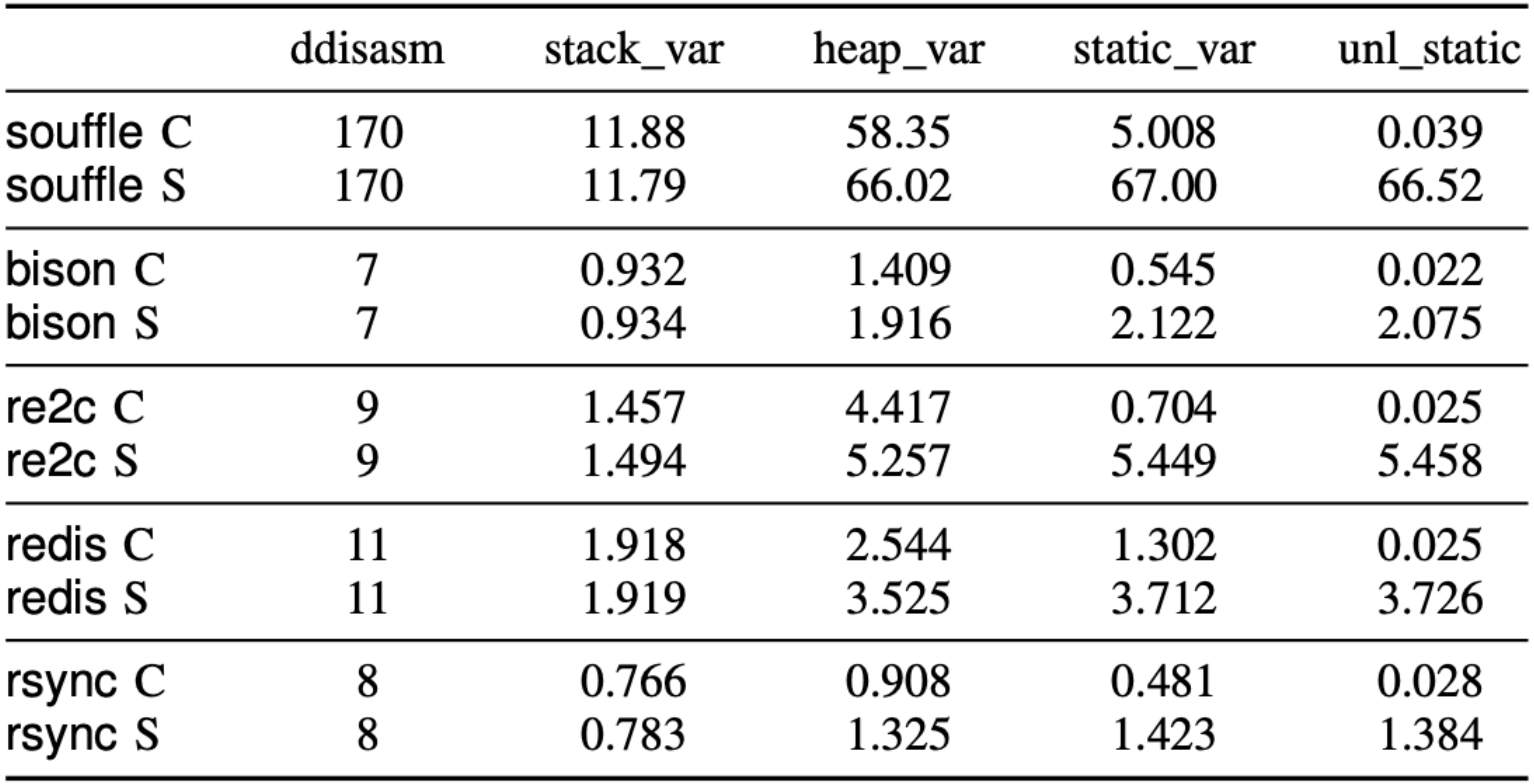
	Ghidra Python	d3re Datalog
non-xor	33	8
overflow	60	18
basicblk	37	4
findcrypto	166	45

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



	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

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

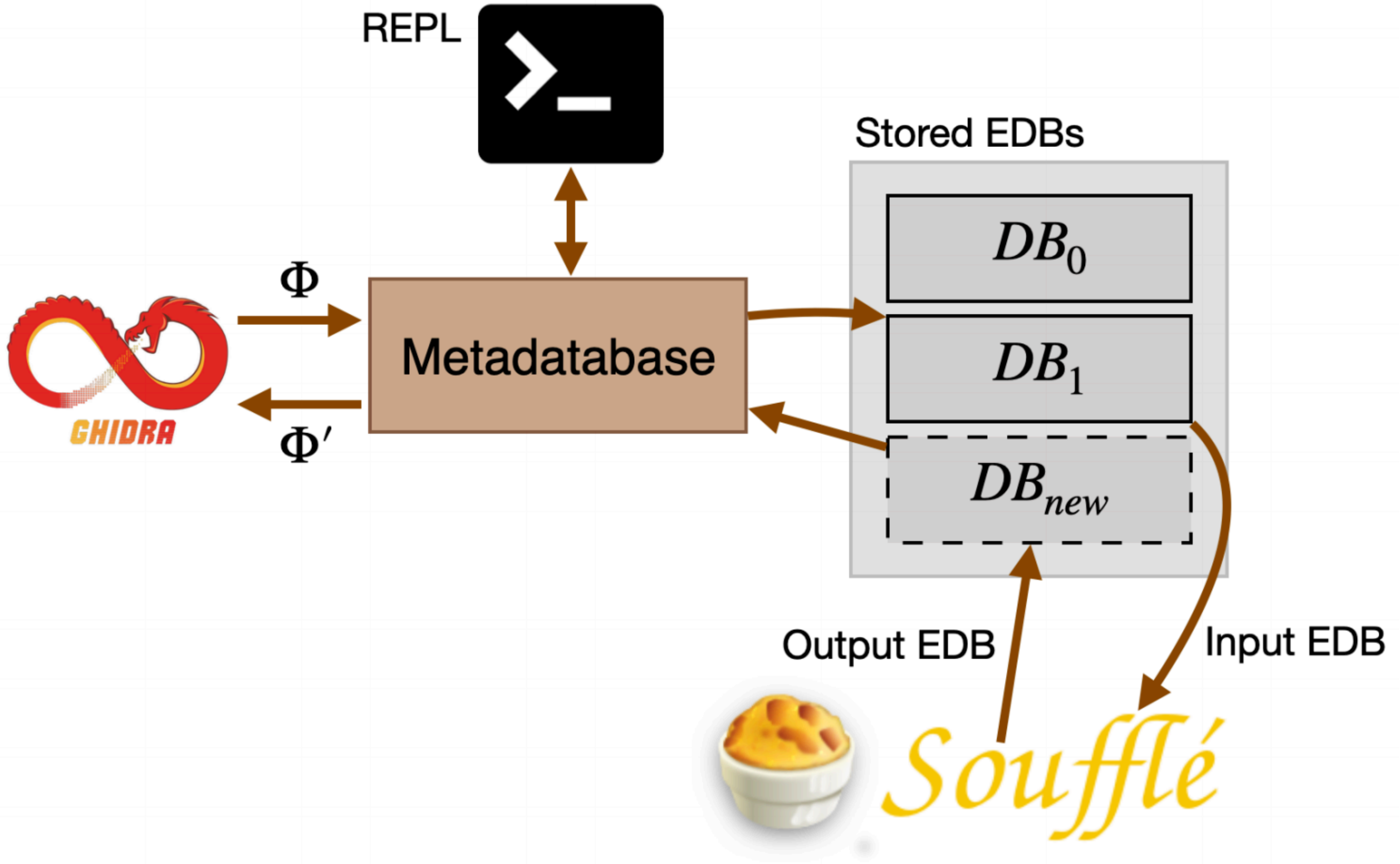


	ddisasm	stack_var	heap_var	static_var	unl_static
souffle C	170	11.88	58.35	5.008	0.039
souffle S	170	11.79	66.02	67.00	66.52
bison C	7	0.932	1.409	0.545	0.022
bison S	7	0.934	1.916	2.122	2.075
re2c C	9	1.457	4.417	0.704	0.025
re2c S	9	1.494	5.257	5.449	5.458
redis C	11	1.918	2.544	1.302	0.025
redis S	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

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

- Motivation and Background
- D3RE Design and Methodology
- Demo using D3RE to perform an RE task
- Implementation

- We implemented d3re in Python
- Our metadatabase (daemon) calls out to Soufflé
- "First pass" calls out to ddisasm to cache initial EDB (extensional DB)
- User types in REPL to execute tasks / communicate w/ Ghidra
- Currently using ghidra_bridge to communicate with Ghidra



D^3RE theory

- Datalog is monotonic
  - Handling non-monotonicity is possible in practice w/ restrictions
- d3re exploits monotonicity to cache DBs
  - Runs / queries can make use of previously-calculated EDB
- Metadatabase tracks **runs** (programs + input DBs) to select “best” starting DB