

An Integrated Program Analysis Framework for Graduate Courses in Programming Languages and Software Engineering

Prantik Chatterjee^{*}, Pankaj Kumar Kalita^{*}, Sumit Lahiri^{*}, Sujit Kumar Muduli^{*},

Vishal Singh^{*}, Gourav Takhar^{*} and Subhajit Roy^{*}

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

Kanpur, India

Email: {prantik, pkalita, sumitl, smuduli, vshlsng, tgourav, subhajit}@cse.iitk.ac.in

Abstract—Program analysis, verification and testing are important topics in programming languages and software engineering. They aim to produce engineers who are not only capable of empirically evaluating but, also formally reasoning on the correctness of software systems. We propose a specialized framework, CHIRON, designed to teach graduate-level courses on these topics. CHIRON has a small code base for easy understanding, uses a unified intermediate representation across all its analysis modules, maintains a modular architecture for plugging in new algorithms and uses a “fun” programming language to provide a gamified experience. Currently, it packages a dataflow analysis engine for driving compiler optimizations, an abstract interpretation engine for verification, a symbolic execution engine, a fuzzer and an evolutionary test generator for program testing, and a spectrum based statistical bug localization module.

Within CHIRON, program analysis tasks are posed in an unconventional setting (as adventures of a turtle) to provide a gamified experience; the accompanying animations (showing the movements of the turtle) allow the student to understand the underlying concepts better, and the detailed logs allow the teaching assistants in their grading activities.

CHIRON has been used in two offerings of a graduate level course on program analysis, verification and testing. In response to our survey questionnaire, all the students unanimously held the opinion that CHIRON was extremely helpful in aiding their learning, and recommended its use in similar courses.

I. INTRODUCTION

Program testing and verification courses (referred to as *program analysis*) impart essential skills to software engineers—enabling them to mathematically and empirically argue on the correctness, robustness and security of a software system. There exist many graduate level courses [1], [2], [3], [4], [5], [6], [7], [8] that cover relevant topics, such as fuzzing, compiler optimizations, symbolic execution, abstract interpretation, dataflow analysis, evolutionary test generation, and statistical fault localization. These courses require that the student not only assimilates the “theory” (mathematical theorems, algorithms etc.), but also masters the realization of the theory in practice. Hence, programming assignments are

integral components in all these courses. In fact, feedback from the students—in such a program analysis course offered by us for over a decade—has revealed that the students “get” the theoretical concepts, not in the lecture hall, but while toiling on the programming assignments.

Program analysis tools for different topics—compilers, testing tools, and verification engines—are quite complex, spanning a few tens of thousands of lines of code. Further, these tools use different intermediate representations, provide distinct API interfaces and are implemented in different languages. Hence, given the high cognitive load of mastering such tools, it is unreasonable to expose students to more than a couple such infrastructures in a semester long course.

In this work, we attempt to lower the cognitive load of understanding program analysis implementations. Towards that end, we present CHIRON—an integrated infrastructure for teaching diverse program analysis algorithms under the same umbrella. CHIRON was designed with the following objectives:

- **Small code base.** We keep the codebase small so that the students are not overwhelmed by the codebase while exploring the algorithms. We implement the core algorithms, close to the theoretical principles taught in the class, without complex optimizations that are essential for industrial-strength implementations. The whole codebase of CHIRON consists of approximately **1.72K** lines of Python code (base framework contains 0.77K lines of code and the implementation of 6 program analysis algorithms and interfaces consists of another 0.95K lines of code). We got indicators of the simplicity of the implementation when the students in the course started suggesting feature extensions or bug fixes while citing the exact lines in the codebase that needed changes.
- **Simple Intermediate Representation (IR).** Program analysis tools compile the source program into a data-structure, referred as the intermediate representation (IR), for further analysis. We design a very simple IR with a set of helper functions that students can digest quickly.
- **Fun Source Language.** Instead of mainstream program-

^{*} Authors contributed equally to the paper

ming languages, we picked a small and fun (but Turing complete) language, Turtle [9], as the source language. As Turtle is used to teach introductory programming, we attempt to excite the child within our graduate students. The Turtle language allowed us to design some “fun” assignments that posed program analysis problems in an altogether different light—allowing the students to realize the generality of the concepts, absorb the algorithms better, and hopefully, have more fun solving them.

- **Plug-and-play architecture.** Our framework allows addition new analysis algorithms as plugins that can collaborate with relevant parts of the core infrastructure. Once the base framework was built and the relevant interfaces were exposed, the effort to add a new analysis was about 2 to 3 person-weeks per algorithm. We also plan to add new algorithms in the future.

We have used CHIRON in two offerings of the program analysis course at our university. In our experience of over a decade of teaching this course, with and without CHIRON, we observed a better appreciation of the topics discussed in class when CHIRON was used. In response to post-course surveys collected from the students, 91% students found the CHIRON IR easy to understand and use—a large majority of the class (80%) felt that they needed no more than 3 days to get familiar with the IR and the codebase of CHIRON. About 91% students felt that the choice of a language like Turtle, that allows fun visualizations, helps learning. Finally, all of the students unanimously felt that CHIRON was extremely helpful or indispensable for graduate level courses on program analysis, and they recommend its in similar courses.

The idea of building specialized frameworks for education is not new: compiler design courses benefit from educational compilers like Bril [10] and Cool [11]; courses on operating systems regularly employ frameworks like PintOS [12] and NachOS [13]. However, courses in program analysis lack a similar infrastructure, and CHIRON attempts to fill that void. CHIRON is analogous to a *flight simulator* that is equipped to simulate a variety of aircrafts and environments—while it is important to fly a real plane eventually, hundreds of flight hours that one can afford to expend on a flight simulator is also crucial to the pilot’s training [14].

The contributions of this paper are as follows:

- We propose a new approach to teaching program analysis that reduces the overhead of programming assignments through specially designed frameworks;
- Towards that end, we build CHIRON, an integrated and extensible infrastructure with a small code footprint, supported by a simple intermediate language that enables multiple (pluggable) analysis on a fun language, Turtle, to teach graduate level program analysis topics;
- The course surveys indicated that our new approach to teaching program analysis was appreciated by students.

CHIRON is available open-source at: <https://github.com/PRAISE-group/Chiron-Framework>

Table I: **Summary of Frameworks and Programming Languages used in some course offerings**

Course	Tools & Frameworks
CS 232 [1]	BDDBBDB [15]
CS 6.820[2]	COQ [16], SPIN [17],
CS 15-819[3]	SOOT[18], DAFNY [19]
CS 17-819[4]	SOOT[18], CODEQL [20],
CS 501 [8]	POLYGOT [21]
CS 700[6]	RANDOO [22], AFL [23], LLVM [24], KLEE [25], KORAT [26], DELTA [27], CBI [28]
CS 6120[5]	LLVM [24], BRIL (IR) [10]
CS 673 [7]	CBMC [29], DAFNY [19], UPPAAL [30], NUSMV [31], SPIN [17], PRISM [32]
Our Course	CHIRON (OUR TOOL)

II. MOTIVATION

Program analysis courses balance have to balance theoretical and practical components—programming assignments are crucial to assimilation of the theoretical concepts discussed in lectures. In our courses, queries posed on assignment forums indicated that the understanding of the students deepen as they progress in their programming assignments. Hence, almost all program analysis courses offered worldwide have a strong component on programming assignments (see Tab. I).

Tab. II is a survey of tools used in some excellent program analysis courses across multiple universities. The size of the codebases (KLOC) is some indication of the difficulty of working with these tools; further, as the tools are implemented in different languages, it also requires the student to be well-versed in many languages. Hence, it is not surprising that students find it challenging to install these tools along with their many dependencies, navigate the large codebases, understand the different intermediate representations, discover the functionality of the (often sparsely documented) APIs, and complete non-trivial tasks on them within four months in a semester long course. Hence, the course designers and instructors face the challenging task of making the right compromises between the theoretical and practical aspects that maximizes the learning outcome of the course.

Currently, the programming assignments are designed as per one of the following two approaches:

- **Master a few.** Some courses are designed to focus on a few selected techniques. In such courses, the programming assignments are designed such that the students attempt *challenging problems* on a *few tools*.
- **Expose to many.** On the other hand, certain courses are designed to equip students on a broad set of topics. In such courses, the students are exposed to *simpler assignments* but cover a *broad array of tools*.

Tab. IV classifies courses as per how *challenging* the assignments are, versus number of different tools employed in the course: existing courses either fall in the second (*expose to many*) or the third quadrant (*master a few*).

In this work, we propose a third approach (the fourth quadrant in Tab. IV) to designing assignments where the students face relatively *challenging assignments* on a *broad set of topics*. To make it possible, instead of using industry-

Table II: Details about the tools used in various course offerings

Tools	KLOC [†]	Prog. Lang.	Purpose
BDDDBDBB[15]	—*	MINIJAVA/JAVA	Deductive Analysis
SPIN[17]	47	C/C++	Model Checker
AFL[23]	150	C/C++	Fuzzer
KLEE[25]	90	C/C++	Symbolic Executor
SOOT[18]	467	JAVA	Compiler Framework
LLVM[24]	2800	C/C++	Compiler Framework
CBMC[29]	857	C/C++, SWIG	Bounded Model Checker
POLYGOT[21]	133	JAVA, JULIA	Compiler Framework
CODEQL[20]	127 [#]	TYPESCRIPT	Code Analyzer
DAFNY[19]	141	C#	Deductive Verifier
RANDOOP[22]	197	JAVA	Test Suite Generator
DELTA[27]	1	PERL, C	Bug Localizer
CBI[28]	—*	—*	Bug Localizer
KORAT[26]	—*	JAVA	Test Suite Generator
UPPAAL[30]	—*	UPPAAL LANG	Model Checker
PRISM[32]	361	JAVA, C/C++	Prob. Model Checker
CHIRON	1.72	PYTHON	Compiler Framework
			Fuzzer, Bug Localizer
			Abstract Interpretation
			Symbolic Executor
			Test Suite Generator

*Source code not available. [#]Plugin stripped down size. [†]1KLOC=1000 Lines of Code.

strength tools in the course, we build CHIRON—a *specialized program analysis framework* focused for education.

CHIRON mimics the design principles of industry-strength frameworks that implement these algorithms, sans the large array of robust features and optimizations. Further, as CHIRON is an integrated framework with a small codebase, it allows us to capture detailed and expressive logs of the execution of the algorithms to facilitate grading by the teaching assistants. Overall, CHIRON provides the advantages of a small code base, a simple and unified intermediate representation across all analysis algorithms, a fun base language and a plug-and-play architecture for extending the framework (see Sect. V).

Tab. III shows the details on the implementation of CHIRON. We kept the codesize of each module small (*Module Size*) to ease understand of the implementation. Further, development effort for existing modules (*Module Effort*) is small, indicating that new modules can be added to CHIRON easily. Our assignments were designed to test the algorithmic thinking of the students, while keeping the implementation effort low; the lines of code (LOC) submitted by students remains small (see *Student Codesize*)—allowing us to float many assignments covering the breadth of the course.

All the above approaches are incomparable—an instructor should judiciously select the tools based on the learning objectives. *Master a few* can be used to focus in *depth* on a few selected topics. On the other hand, *expose to many* and our proposal target the *breadth* of techniques. The differentiator between *expose to many* and our approach is the instructor’s focus on algorithms versus tools. If the instructor values

Table III: Details of Modules used in CHIRON. Module size and student codesize are measured in KLOC, whereas module effort is measured in person weeks.

Module	Module Size [#]	Module Effort	Student Codesize*
CORE FRAMEWORK	0.77	8	—
DATA FLOW ANALYSIS	0.14	2	0.2–0.3
FUZZING	0.11	2	0.06–0.1
SYMBOLIC EXECUTION	0.31	3	0.15–0.17
ABSTRACT INTERPRETATION	0.15	2	0.1–0.3
SBFL FRAMEWORK	0.24	2	0.03–0.12

*Code written by students for the assignments. [#]Code written by CHIRON developers for framing the assignments.

Table IV: Tradeoffs b/w assignment difficulty vs. tools used

		# Tools explored →	
		Few	Many
Assignment difficulty ↓	Low	—	CS 700[6], CS 673 [7]
	High	CS 232 [1] CS 6.820[2] CS 15-819[3] CS 6120[5] CS 17-819[4] CS 501 [8]	Our course using CHIRON

exposition to industry-strength tools, the former should be used; as these tools are challenging and time-consuming to install and get familiar with, such courses can only attempt to “expose” students to these tools with “light” assignments. Our approach, however, focuses on the algorithms rather than the tools, and hence, allows the students to master the algorithmic ideas from a diverse set of topics on a unified framework that is specifically designed for this purpose.

III. OVERVIEW

A. Program analysis modules in CHIRON

Program analysis commences with parsing the source program to be analyzed, and compiling it to an efficient data-structure, referred to as the *intermediate representation (IR)*. Program analysis frameworks use diverse IRs—tailored for the algorithms implemented by the respective tools, and the users must get acquainted with them to use the framework.

Another important data-structure is the *control-flow graph (CFG)*, (s, N, E) , where the start node (n) is the entry-point of the procedure and the set of nodes are program statements (or a sequence of single-entry-single-exit statements referred to as *basic-blocks*). An edge ($e \in E$) exists between two nodes, s_1, s_2 , if control can transfer from statement s_1 to s_2 .

CHIRON covers a variety of algorithms, spanning those for control/dataflow analysis, verification and testing—we refer to all these algorithms simply as *program analysis* algorithms in this article. CHIRON, at present, provides the following analysis algorithms:

a) *Dataflow Analysis and Compiler Optimizations* [33], [34]: Dataflow analysis provides a formal framework to compute if certain properties hold at certain program points. This requires one to summarize the effect of every possible program construct on the property as a *transfer function*. A fixpoint computation eventually answers the state of the property at the different program points.

b) *Abstract Interpretation* [35]: Instead of reasoning on *concrete* values of variables, abstract interpretation advocates modelling the program values in an *abstract domain*. One has to define *abstract transfer functions* for each program operator for abstract interpretation of program executions.

c) *Program Fuzzing* [36]: Modern fuzzers employ genetic algorithms to learn from past program runs to identify *interesting inputs* and applies well-crafted *mutation operators* to generate new inputs that are likely to cause failures.

d) *Symbolic Execution* [37]: Symbolic execution assigns *symbolic names* to the program inputs, and computes symbolic expressions over them (referred to as *symbolic values*). The reachability of an error condition is modelled as a logical formula over the symbolic names, and a satisfiability checker is used to answer if the program can fail.

e) *Spectrum-based Fault Localization (SBFL)* [38]: SBFL uses a lightweight approach to rank program locations by their *suspiciousness* of causing a failure. It operates by recording execution spectrums—given a set of test inputs, it constructs a vector that records the program locations reached on the respective inputs; these vectors can be stacked as an *activity matrix*. Further, the outcome of each test (pass/fail) is also recorded in an *error vector*. An *activity matrix* along with the corresponding *error vector* is collectively referred to as a *spectrum*. The involvement of a component in a test that fails indicates its suspiciousness for causing the failure. There exist many metrics (like [39], [40]) to gauge such correlations and provide a *suspiciousness* score to rank program locations.

f) *Evolutionary search based test generation* [41]: Evolutionary search is used to synthesize test-suites, driven by certain *testing* goals (e.g. coverage, fault localization etc.) captured by an optimization objective (used as fitness function).

B. Course Overview

We share our experience of teaching program analysis using CHIRON in our semester-long (4 months) graduate level course. The course is quite mature and has evolved over more than ten years. We felt the need for such an infrastructure over these offerings and this paper shares our experience from the last two offerings of this course that we ran with CHIRON.

The course exposes the students to a variety of program analysis techniques and is meant to prepare students for research in this area. Most of the students who registered for the course had no exposure to program analysis, verification or testing; however, they had exposure to undergraduate level compiler design and theory of computation courses. The course starts off with control-flow and dataflow analysis [33], [34] techniques. Then, it exposes the students to abstract interpretation [35], fuzzing [36], symbolic execution [37], evolutionary search based software testing [41] and statistical fault localization [39], [40].

C. An Overview of the CHIRON Language

CHIRON is designed to analyze programs in CHIRONLANG. CHIRONLANG is essentially a subset of the popular graphics-based *Turtle* dialect of the LOGO programming language

family [42]. LOGO is popular for teaching programming to children and, being a fun but a Turing complete language, we felt it could be a good device to teach program analysis as well. CHIRONLANG is a dialect of Turtle/Logo that is closest to KTurtle [9] and TurtleAcademy [43].





On invocation, CHIRON launches a canvas with a small turtle () at some initial position. A CHIRONLANG program controls the movement of , and, thereby, draws interesting shapes on the canvas as  moves. Fig. 1 shows a CHIRONLANG program and the drawing it creates.

Fig. 2 summarizes the syntax of the CHIRONLANG language. The statements in CHIRONLANG can be categorized as: (1) assignment statements, (2) control-flow statements, (3) loop statements, (4) movement statements, (5) pen statements. The assignment, control-flow and loop statements are standard imperative language constructs, while the movement statements control the position of the  and the pen statements control if the pen is up (drawing off) or down (drawing on). Variables need to be prepended by a colon (':') and square brackets ('[', ']') are used for scoping.

IV. TASKS PROVIDED TO STUDENTS

We motivate CHIRON via a set of student assignments used in our graduate-level course. For the assignments, the students are provided the base framework that includes the CHIRONLANG parser, the IR generator and multiple utility functions to analyze/manipulate the IR. Each assignment is based on one of the program analysis techniques, like control/dataflow analysis, abstract interpretation, symbolic execution, fuzzing, and statistical testing. In these assignments, the primary algorithm is implemented with a few procedures left out for the students to complete. Each of these incomplete procedures have well-defined interfaces and detailed comments. Following are some of the functionality that the students implement:

- *Abstract Interpretation*. With the complete implementation of the fixpoint algorithm provided, the students are required to specify their analysis in terms of the lattice to describe the abstract domain and the abstract transfer functions for the language components;
- *Compiler Optimizations*. The students are provided the complete machinery for computing dataflow facts using the fixpoint algorithm; they describe their analysis and use the analysis results to transform the IR (utility functions for adding/deleting instructions in the IR were available as part of the base framework);
- *Coverage-guided Fuzzing*. The students are expected to design the coverage feedback and the mutation operations within the fuzzing engine provided by the framework;
- *Evolutionary search and fault localization*. The students implement a fitness function (like DDU [44], Ulysis [45]) to drive test generation and suitable metrics for fault localization (like Ochiai [46], DStar [47], Artemis [48]).
- *Symbolic execution and program synthesis*. The students use the symbolic execution engine in CHIRON it to build a program synthesis application.


```

1 :x = 20; :y = 100; :z = 60
2 repeat 6 [
3   if (:x != :y) [
4     if (:x > :y) [ right :x ]
5     else [ left :y ]
6   ] else [
7     if ((:x <= :z) || (:y <= :z))
8       :x = :z / :x
9       :z = :y / :z
10  ]
11 forward :x; right :y
12 forward :z; left :z
13 ]

```

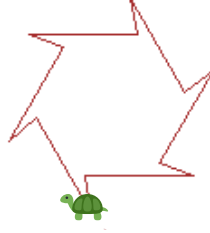


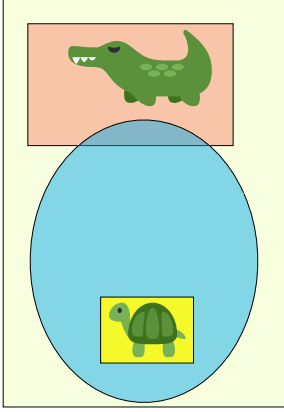
Figure 1: A CHIRONLANG program and the corresponding drawing created by the 🐢 movements

```

⟨stmt⟩ ::= ⟨stmt⟩ ; ⟨stmt⟩ | ⟨asgn⟩ | ⟨ifstmt⟩
        | ⟨loop⟩ | ⟨pen⟩ | ⟨move⟩
⟨asgn⟩ ::= var = ⟨expr⟩
⟨ifstmt⟩ ::= if ⟨cond⟩ [⟨stmt⟩]
⟨loop⟩ ::= repeat ⟨expr⟩ [⟨stmt⟩]
⟨pen⟩ ::= penup | pendown
⟨move⟩ ::= forward ⟨expr⟩ | backward ⟨expr⟩ | left ⟨expr⟩
        | right ⟨expr⟩ | go (⟨expr⟩, ⟨expr⟩)
⟨cond⟩ ::= true | false | ⟨cond⟩ ⟨blop⟩ ⟨cond⟩ | !⟨cond⟩
        | ⟨expr⟩ ⟨brop⟩ ⟨expr⟩ | (⟨cond⟩)
⟨expr⟩ ::= var | num | ⟨expr⟩ ⟨baop⟩ ⟨expr⟩ | -⟨expr⟩
        | (⟨expr⟩)
⟨baop⟩ ::= + | - | * | /
⟨blop⟩ ::= && | ||
⟨brop⟩ ::= < | > | == | != | <= | >=

```

Figure 2: Syntax grammar of CHIRONLANG



```

1 input (:sX, :sY)
2 go (:sX, :sY)
3 :posX ← :sX
4 :posY ← :sY
5 :θ ← 0
6 repeat 8 [
7   right 45
8   :θ += 45
9   repeat 90 [
10    forward 4
11    :posX += 4 * cos(:θ)
12    :posY += 4 * sin(:θ)
13    right 2
14    :θ += 2
15    repeat 4 [
16     forward 10
17     :posX += 10 * cos(:θ)
18     :posY += 10 * sin(:θ)
19     left 90
20     :θ += 90
21  ]]]

```

Figure 3: The scared turtle

The source code of the whole framework was available to the students in case they wanted a deeper understanding of the whole framework and how the functions they implement interfaces with the rest of the framework. Now, let us detail the setting of some of these student assignments on CHIRON.

A. The Scared Turtle

Concepts introduced: *Abstract interpretation*

1) *Problem statement:* In this assignment, the students attempt to track if the movements of 🐢, as dictated by a CHIRONLANG program, is guaranteed to be safe (Fig. 3). The 🐢 moves in accordance to a provided input program, and eventually rests at the final position that it reaches at the end of the program. However, a part of the canvas is inhabited by a dangerous alligator; so, 🐢 is scared as if 🐢 rests within this region, it will be devoured by the alligator. Given a CHIRONLANG program, the students need to verify if 🐢 could rest in the alligator's region. The alligator's region is rectangular in shape and can be described by x-coordinate interval and y-coordinate interval e.g., $[x1, x2], [y1, y2]$.

2) *Solution strategy:* The students use abstract interpretation with the interval domain to solve this problem. Let us consider the CHIRONLANG program shown in Fig. 3: to begin with, the program can be instrumented with ghost statements that track the position of 🐢 after every statement (shown in gray text) via the ghost variables $:posX$, $:posY$ and θ . Next, the students design appropriate abstract transformers for the movement statements (e.g. go, forward, left and right), and run abstract interpretation on the program to compute an overapproximation of the final position. Finally, if the overapproximation computed by the students does not intersect with the alligator's territory, 🐢 is verifiably safe.

B. The Lazy Turtle

Concepts introduced: *Dataflow analysis and optimizations*

1) *Problem statement:* In this task, the students are required to help out a lazy 🐢 by optimizing its movements so that it can draw the same shape but with fewer movements. For example, the program shown in Fig. 4a can be optimized to Fig. 4b, as the movements of the Turtle are reduced (or remain unaltered) along all paths.

2) *Solution strategy:* This problem can be formalized similar to solving for redundancy elimination [49]. The students are required to perform dataflow analysis [33], [34] on the program to identify redundant movements, i.e. design appropriate transfer functions, perform the fixpoint computation and finally perform the required transformations. CHIRON provides support for the above activities in the form of various helper functions that can add new instructions and delete existing instructions from the IR of the program.

C. The Turtle Explorer

Concepts introduced: *Coverage-guided fuzzing*

1) *Problem statement:* In this assignment, the students help 🐢 in its quest to explore. Given a CHIRONLANG program (in Fig. 5) that determines the movement of 🐢 on the canvas (canvas is the complete canvas region in Fig. 5), students find a set of *interesting* start positions (or inputs) that maximize the total canvas area covered (pink regions in Fig. 5) by 🐢.

```

1 input (:x, :w, :z)
2 pendown
3 :y1 = :x + 20
4 # Optimization possible!
5 # this move is redundant
6 forward :y1
7 if (:y1 > 42) [
8   :y2 = :z - 20
9   forward :y2
10 ] else [
11   :y2 = :w - 20
12   forward :y2
13 ]
14 penup

```

(a) Original program

```

1 input (:x, :w, :z)
2 pendown
3 :y1 = :x + 20
4 forward :y1
5 if (:y1 > 42) [
6   :y2 = :z - 20
7   :t = :y1 + :y2
8   forward :t
9 ] else [
10   :y2 = :w - 20
11   :t = :y1 + :y2
12   forward :t
13 ]
14 penup

```

(b) Optimized program

Figure 4: The lazy turtle

```

1 input (:x, :z1, :z2, :z3)
2 input (:sq1, :sq2)
3 forward :z1
4 left :z2
5 forward :z3
6 pendown
7 repeat :x [
8   forward :sq1
9   left 90
10 ]
11 repeat :y [
12   forward :sq2
13   left 90
14 ]
15 penup

```

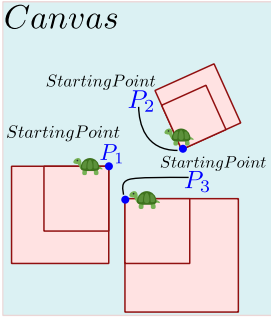


Figure 5: The turtle explorer

The movement of 🐢 for some start positions (P_1 , P_2 and P_3) along with a CHIRONLANG program is shown in Fig. 5.

2) *Solution strategy*: Students use fuzzing [36] to solve this problem with *canvas area coverage* as the optimization objective. The students design a set of mutation operators and the optimization objective that would plug into the fuzzer available in CHIRON.

D. The Lost Turtle

Concepts introduced: *Evolutionary search based test generation, spectrum based fault localization*

1) *Problem statement*: Two turtles friends, 🐢 and 🐢, want to ensure that their respective maps (input programs) will always allow them to reach the same destination when they start from the same point. However, the map of 🐢 is buggy (its map gives a wrong instruction at Line 9 i.e., [left 45] as shown in Fig. 6). Due to this “bug”, for some input directions :d, though 🐢 and 🐢 start from the same initial location, 🐢 is lost. The students help 🐢 identify the faulty instruction in its map?

2) *Solution strategy*: The students use the test generation strategy within CHIRON to execute the faulty program P_2 and identify the status of the test (pass/fail) by comparing the final destination against that of the oracle P_1 . This allows the construction of a program *spectrum*. Next, the students employ spectrum-based fault localization strategies (like Ochiai [39],

```

1 input (:d)
2 forward 100
3 right 90
4 forward 100
5 if (:d > 90) [
6   right 90
7 ]
8 else [
9   right 45
10  left 45
11 ]
12 forward 100

```

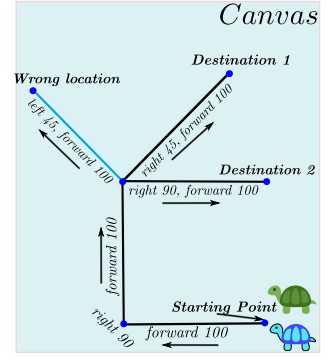


Figure 6: The lost turtle

DStar [40] etc.) to compute a *suspiciousness* of each instruction in P_2 . A final ranking reveals the likely culprit.

E. The Turtle Friends

Concepts introduced: *Symbolic execution, Prog. synthesis*

1) *Problem statement*: Continuing with the above problem of turtle friends, 🐢 and 🐢, with an incorrect map, we now wish to automatically *repair* the faulty map. The lost Turtle, having identified the incorrect statement (via the previous assignment), erases the faulty parameter to get a *partial* map. The students have to help the turtle identify the correct parameters such the two friends end up with (semantically) equivalent maps (i.e. if they start from the same position in the two maps, they reach the same destination). 🐢 and 🐢 move as per the instructions of P_1 and P_2 respectively. The program, P_2 is a partial program. The students complete the program, P_2 , such that if 🐢 and 🐢 start from the same position (program inputs), then, they always meet at the same position when their respective programs terminate.

Fig. 7a and Fig. 7b show two programs, P_1 and P_2 respectively, where :x is the input to both programs. Program, P_2 , is partial as the values of constants :c1 and :c2 are unknown. Here, the assignment aims to find values of :c1 and :c2 in program P_2 (Fig. 7b) such that, at the end of both program executions, 🐢 and 🐢 converge at the same position.

2) *Solution strategy*: The students solve this problem by program synthesis via sketching [50]—infer values corresponding to the *holes* to establish semantic equivalence across the two programs. To solve this assignment, the students use the symbolic execution engine provided by CHIRON to enumerate all paths across the programs and collect the corresponding *path conditions*. The programs can now be encoded symbolically as a disjunction over their respective path conditions. Now, it is a matter of solving the existential quantification for the holes that establishes equivalence over the symbolic encodings of the two programs. The students discharge the respective constraints to a theorem prover to solve for the existential quantification.

V. DESIGN OF CHIRON

We show the architecture of CHIRON in Fig. 8. CHIRON accepts input programs in two modes: as the input CHIRON-LANG program source or as its Intermediate Representation

```

1 input (:x)
2 pendown
3 :y = :x
4 if :x <= 42 [
5   :y = :y + 40
6 ] else [
7   :y = :y + 22
8 ]
9 forward :y

```

(a) Complete program

```

1 input (:x)
2 pendown
3 :y = :x
4 if :x <= 42 [
5   :y = :y + :c1
6 ]
7 :y = :y + :c2
8 forward :y

```

(b) Partial program

Figure 7: The turtle friends

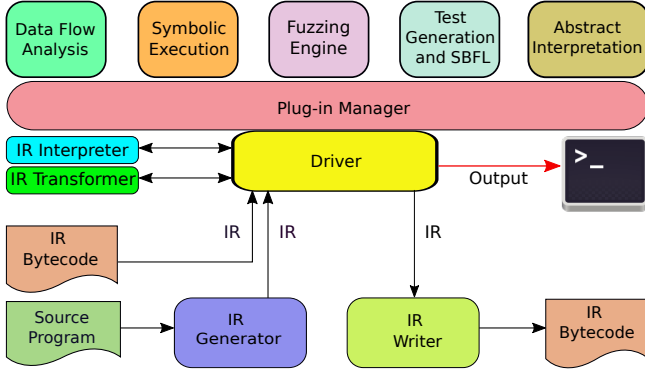


Figure 8: CHIRON architecture

(IR) bytecode. The driver loads the IR into a data-structure and makes it available to the various analysis modules. The IR interpreter allows execution of a CHIRONLANG program and the IR writer can write the (possibly transformed) IR as IR bytecode to disk. The plugin manager allows adding new analysis modules.

A. CHIRON Intermediate Representation (IR)

CHIRON translates CHIRONLANG programs to the CHIRONLANG IR on which the program analysis plugins operate. The IR is a list of `Instr` objects that store the instruction

```

:x = 0
go (34, 60)
repeat 5 [
  pendown
  if (:x > 2) [
    forward 30
    right 90
  ]
  else [
    backward 60
    left 90
  ]
  penup
  :x = :x + 1
]

```

(a) A CHIRONLANG program

```

1 :x = 0
2 go 34 60
3 :rcnt1 = 5
4 JMPIFN (:rcnt1 != 0) 12
5 pendown
6 JMPIFN (:x > 2) 4
7 forward 30
8 right 90
9 JMPIFN False 3
10 backward 60
11 left 90
12 penup
13 :x = (:x + 1)
14 :rcnt1 = (:rcnt1 - 1)
15 JMPIFN False -11
16 END

```

(b) IR for Fig. 9a

Figure 9: CHIRONLANG program containing different types of statements with its corresponding IR

opcode and a list of abstract syntax trees (ASTs) corresponding to the arguments (Fig. 10 has a sketch of the `Instr` class).

Our objective was to keep the IR short and close to the source program. We show the translation of a source program to its IR in Fig. 9; the statements are color-coded to map the source and IR statements in the translation.

The IR has the following category of instructions:

- **assignment** instruction. A variable can be assigned to an expression represented by its abstract syntax tree.

$\langle \text{var} \rangle = \langle \text{expr} \rangle$

- **jump** instructions. Branch and loop instructions in the source program are translated to (conditional) jump instructions with relative jump addresses (unconditional jumps are supported by setting the condition as False).

JUMPIFN $\langle \text{cond} \rangle$ $\langle \text{relative_offset} \rangle$

- **movement** and **pen** instructions. CHIRON supports movement instructions like forward, backward, left, right and go. It supports penup and pendown for pen instructions to disable/enable drawing.

B. Software Design of CHIRON

Fig. 10 shows a few of the important classes and methods in CHIRON. The primary object of its architecture is to keep the codebase small by ensuring high code reuse and provide generic interfaces that can be used across multiple analysis modules thereby reducing the cognitive load on the students.

The IR class provides functionalities to compile the source language, CHIRONLANG, to an intermediate representation (via `loadIR()` method) as a list of instructions (`stmtList`). It also provides methods that manipulate the IR, construct the control-flow graph and other utility methods. The `Instr` class provides the interface for instructions, while each instruction (like `Move` and `Asgn`) derive from `Instr` to provide their specialized functionalities.

The `Interpreter` abstract class provides the primary interfaces to interpreting the intermediate representation. The `Interpreter` class contains the IR of the input program (`ir`), the program counter (`pc`), and the program state (`state`) to support execution on the IR. `ConcreteInterpreter` implements this interface to provide functionalities for running the IR on concrete inputs. Static analysis tools derive from the `Interpreter` class to handle abstractions: `AbstractInterpreter` adds functionalities for interpretation on abstract inputs. `Dataflow Analysis (DFA)` further inherits from `AbstractInterpreter` to reuse the functionalities for fix-point computation while adding additional functionalities. Dynamic analysis algorithms like `Fuzzer` and `SBFLAnalysis` derive from the `ConcreteInterpreter` while overloading the instruction handlers (`handleStmt()`) for relevant instructions to hook instrumentation code to collect execution statistics. `SymbolicInterpreter` (that implements the concolic execution [51] variant of symbolic execution) inherits the functionalities of the `ConcreteInterpreter` for

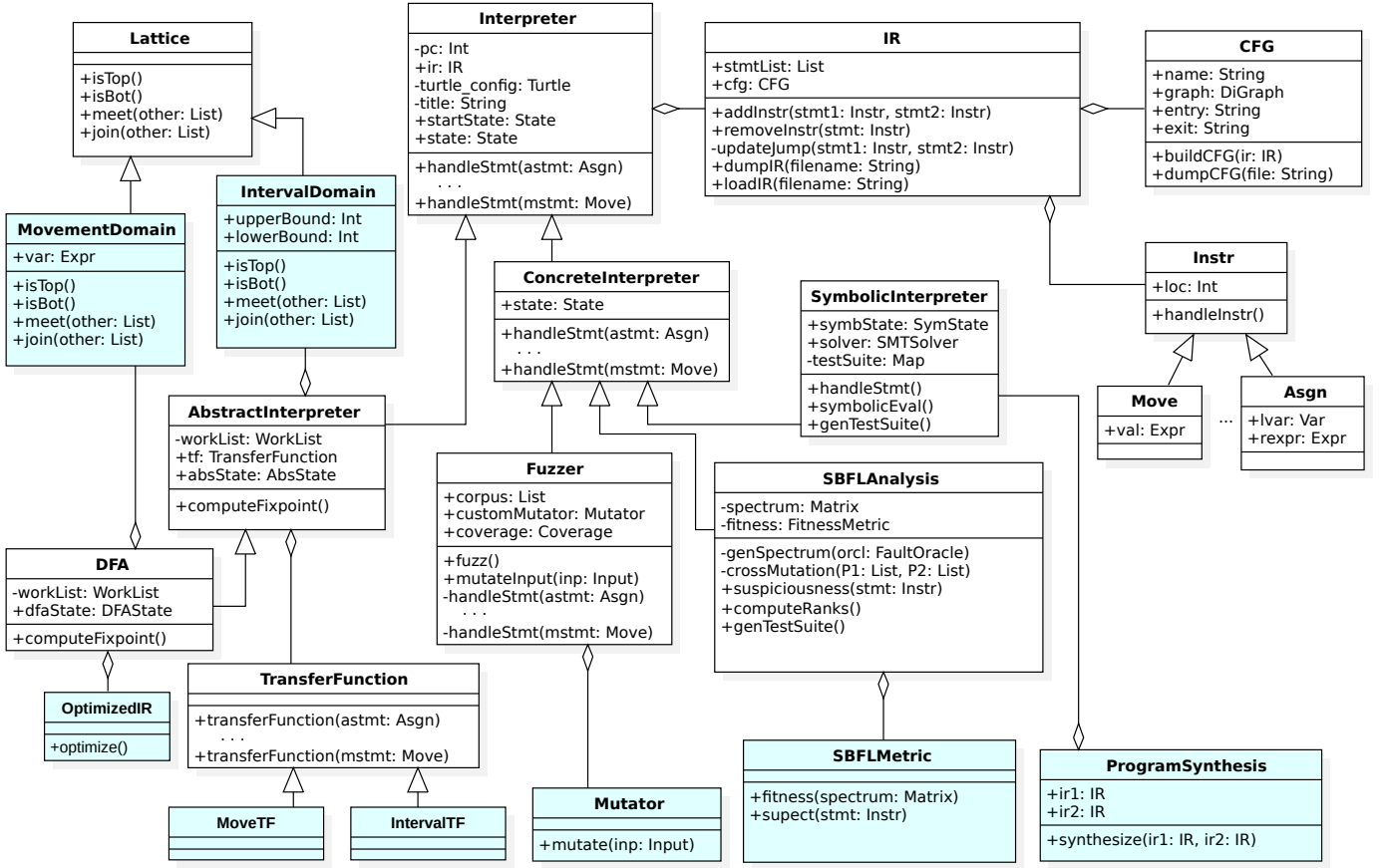


Figure 10: Software architecture of CHIRON

concrete execution while overloading the instruction handlers for maintaining the symbolic run.

The classes shown in cyan are the modules that the students needed to implement in their programming assignments.

a) Dataflow Analysis Based Compiler Optimizations:

The DFA class implements the worklist-based fixpoint algorithm for dataflow analysis. The students were supposed to define their own lattice (MovementDomain) and transfer function (MoveTF) to analyze the turtle’s movements and produce an optimized IR (using the IR manipulation utilities in IR class) to implement the optimize() method in the OptimizedIR class (see §IV-B).

b) Fuzzing:

The Fuzzer class provides a coverage-guided fuzzer. The students implement the Mutator class for their custom mutations. The animation of the turtle moving through the canvas can be seen while fuzzing is in progress, providing a visual feedback on the effectiveness of the designed mutation operators.

c) Symbolic Execution:

Symbolic execution generates symbolic states (symbState) and path conditions (pc) for paths traversed in the program (ir). The students use SymbolicInterpreter to solve their respective programming assignment (§IV-C) by implementing the ProgramSynthesis class.

d) Evolutionary Search Based Test Generation and Spectrum Based Fault Localization (SBFL): The SBFLAnalysis class uses genTestSuite() to generate a test-suite using evolutionary search, and suspiciousness() to perform SBFL. The students implement the SBFLMetric class: the fitness() function to define a custom fitness function for evolutionary search and the suspect() method to define an SBFL metric for fault localization.

e) Abstract Interpretation: AbstractInterpreter provides machinery to drive abstract interpretation. The students define their own abstract domain, IntervalDomain (subclassed from Lattice) to define the respective abstract domain (interval, in their case), and IntervalTF to define abstract transformers for each instruction.

VI. CASE STUDY

As of now, we have made two offerings of our program analysis course using CHIRON. Since, this is an advanced graduate level course, the number of students were not large: there were 13 students in the first offering of the course and 19 in the second, for a total of 32 students across the two offerings. At the end of each offering, the students were asked to participate in a voluntary survey. We summarize the feedback in response to our questionnaire below. Participation in the survey as well as learning program analysis via CHIRON

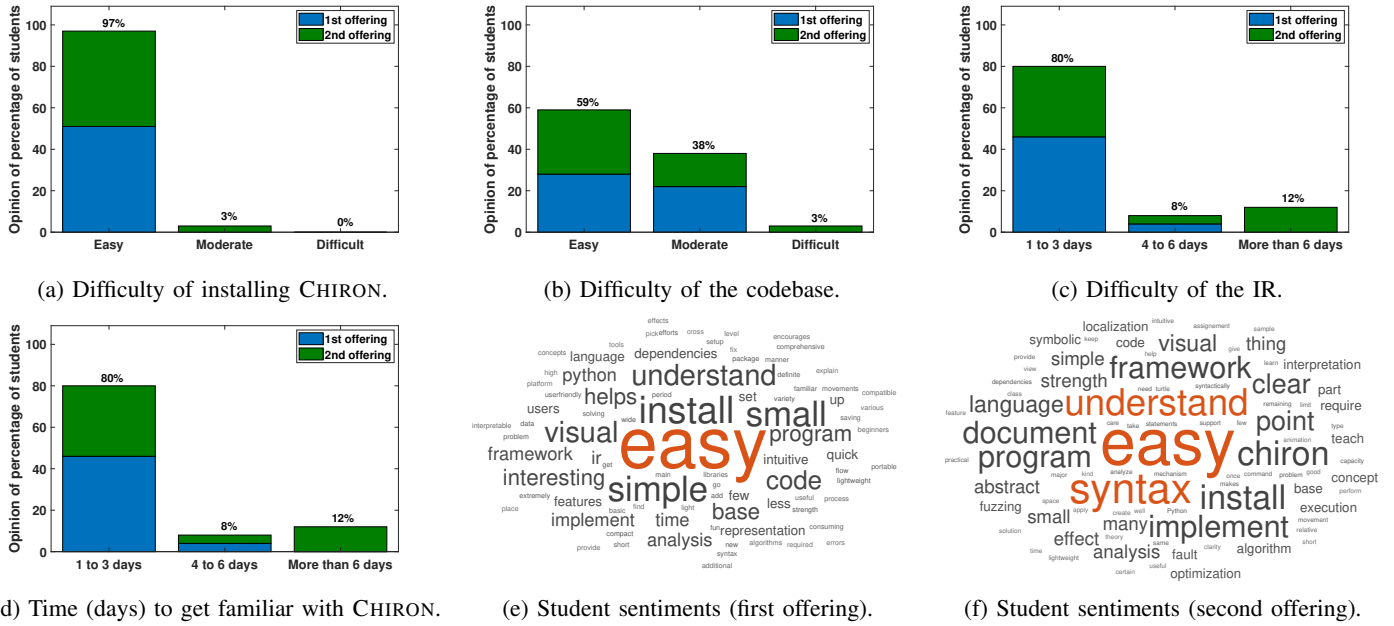


Figure 11: How does CHIRON fare as an introduction to program analysis?

was voluntary, and the students provided consent to use their anonymized responses to the questionnaire in publications.

The background of students who registered for the courses was as follows: 22 out of 32 students (69%) had no familiarity with program analysis; 10 of them (31%) had a little familiarity, and that too with at most one program analysis strategy such as fuzzing or fault localization.

As the studies were conducted in live courses, there were certain constraints. Our studies had to be designed such that they do not affect learning, the difficulty of the course remains uniform for all students, and that the students were graded in an unbiased manner. Also, being an advanced graduate level course, the courses had small class sizes. Therefore, statistically significant studies, that are meaningful only with a large number of data-points, was not possible. So, our survey had to be designed guardedly, and the feedback had to be interpreted carefully.

Given the constraints, we restrict ourselves to study *only* the effect of CHIRON in the context of teaching a first-level program analysis course. The objective of this case-study is *not* to pose the superiority of CHIRON over other program analysis tools. Our only argument is to establish the use of frameworks like CHIRON as a viable alternative approach to teaching first-level program analysis courses (in line with such frameworks for other courses like operating systems [12], [13], [52], [53] and compiler design [10], [11], [54]).

A. Introducing program analysis Through CHIRON

a) Installation: CHIRON introduces beginners to program analysis without the hassle of installing several large tools with multiple dependencies. CHIRON requires only an installation of Python. The feedback of the students (Fig. 11)

shows that CHIRON is easy to get started: 97% of the students were able to install CHIRON with little to no difficulty, whereas only 3% of the students found the installation to be moderately challenging (see Fig. 11a).

b) Complexity of codebase: Fig. 11b shows that 59% of students faced little to no trouble navigating the codebase of CHIRON, whereas 38% students found the codebase to be moderately difficult. Only a single student felt that the codebase was difficult.

c) Simplicity of IR: Fig. 11c shows that 91% of the students found the IR to be very simple and easy to understand as a beginner while only 9% felt it to be moderately complex.

All of the above responses demonstrate that CHIRON is a very effective entry point to learning the concepts of program analysis. Given its simplicity, it is not a surprise that a large majority of the students (80%) were able to get familiar with the inner workings of CHIRON within only 3 days (Fig. 11d), so that they could focus more on the algorithms rather than the intricacies of the codebase.

Importantly, the consistent trend in the survey responses over both offerings seems to present strong evidence of the effectiveness of CHIRON for teaching program analysis.

B. Are the assignments designed in CHIRON interesting?

a) Conceived difficulty of assignments: Our survey indicated that 78% of the students felt that the assignments were *moderately challenging*, whereas the remaining 22% students felt that the assignments were easy but not trivial. This was exactly the expectation from our design. We aimed for a balance in the difficulty between too trivial assignments (that may not buy enough learning from the assignments) and too difficult assignments (that demotivated the students). CHIRON

helped our purpose by ensuring that the students do not get sidetracked on the intricacies of the codebase, and rather, are able to focus on the algorithms.

b) *Visualization*: We asked the opinions of students regarding whether they liked the problem formulations in CHIRON (see Sect. IV) with respect to conventional program analysis tasks. *All but one student were unanimous* in their opinion that they liked the problem formulations in CHIRON. Furthermore, 91% students expressed that they enjoyed the visual representation of problem formulations.

C. Sentiment of Students Regarding Learning via CHIRON

Figs. 11e and 11f are word-clouds of student feedback on learning program analysis using CHIRON. The most prevalent feedback were that CHIRON makes learning program analysis “easy”—it is *easy* to install and the syntax is *easy* to understand on virtue of the codebase being *small, well-documented*. The feedback also mentions the visually interesting nature of CHIRON to be a prominent feature.

Overall, we found that the students not only had an overwhelmingly positive sentiment towards CHIRON, but this positive sentiment were consistent for both offerings of the course across different batches of students. Further, all students unanimously recommended that CHIRON should be used to teach program analysis courses in other institutions as well.

D. Feedback on CHIRON

We provide some feedback from anonymous reviewers and faculty/researchers working in the areas of programming languages/software engineering about CHIRON:

- “*This is a clear improvement of the state of the practice in teaching graduate program analysis. I will consider using CHIRON in my own classes.*”
- “*Interesting contribution from an educational perspective... The adoption of the framework has potential to improve the quality of the learning experience in program analysis courses... The contribution is not only technological (i.e., the actual program analysis framework) but also “methodological” (from the point of view of designing a course on program analysis/verification).*”
- “*I believe that what the authors are doing can be extremely useful for instructors teaching program analysis and testing classes.*”
- “*The framework includes a simple, “fun” target language inspired by LOGO, a standard IR, and a broad suite of program analyses. Once students understand the common aspects of Chiron, they can focus their attention on the particulars of each program analysis.*”
- “*It’s useful see how many problems (analysis, verification, optimization, testing) can all be cast in the context of analyzing a turtle language.*”
- “*Since CHIRON provides diverse analysis modules such as abstract interpretation and fuzzing and allows users to extend a new analysis as a plugin, students can focus on*

learning various analysis algorithms rather than spending much time on installing diverse tools implementing modules themselves.”

On the flip side, some of the faculty/researchers remarked that students, by using CHIRON, miss exposure to “real” program analysis tools. We agree (see §II) that a course that uses CHIRON is not necessarily ideal for all program analysis courses; the *master a few methodology* would be more suitable in courses where experience on certain “real” tools is an important learning outcome.

VII. RELATED WORK

a) *Technology in Education*: Interactive educational tools have become popular, not only for online but also in offline classroom courses taught in high school [55], [56], [57] and undergraduate [58], [59], [60], [61], [62], [63], [64] levels. These tools are interactive and with a visual output that helps grasp concepts better. CHIRON is a step in the same direction, but for a graduate-level (program analysis) course.

There has been a large body of work [65], [66], [67], [68], [69], [70], [71], [72] that targets teaching introductory programming. Some of these even fix bugs or suggest *possible* failing test-cases for student written programs. Moreover, some [73], [74] target at providing suggestions for bug fixes in the code using machine learning based techniques. Some efforts have also been made at teaching advanced computer science topics like computer architecture, compiler design, and operating system by developing tools and frameworks [75], [52], [10], [11], [12], [76], [77], [78], [54], [79], [53] that provide a close-to-real experience at implementing some of the concepts taught in these courses. Students of such courses are expected to either complete the interfaces of the tool or code a complete module from scratch. The students, thus, become better equipped to handle real-world implementations and get an opportunity to bridge the gap between theory and practice. CHIRON is an attempt in the same direction, but for education in graduate level program analysis.

To the best of our knowledge, CHIRON is the first such framework that is specifically designed to aid teaching program analysis, verification and testing concepts.

b) *Use of “fun” Programming Languages*: There are a plethora of courses and frameworks [80], [81], [82], [83], [84], [85], [86] that attempt to make programming exciting for student. SCRATCH [86] offers a gamified experience. We chose such a “fun” language to lighten up the teaching of serious topics like fuzzing, symbolic execution, abstract interpretation etc., by posing program analysis problems in a different setting via some lively 🐢 animations on a canvas.

VIII. DISCUSSION

CHIRON helps students grasp hard-to-understand program analysis concepts in an easy and fun way minus the overhead of learning many tools. The readers would agree that learning sophisticated frameworks, like LLVM, KLEE, AFL, EVOSUITE, CRAB (Seahorn), and Doop [87], requires considerable effort and time. We also want to emphasize that our

study is not meant to demonstrate the superiority of CHIRON w.r.t. overall utility of other tools mentioned above—these industry-strength tools and CHIRON are designed with different objectives (solving real-world problems versus education). All we want to demonstrate is that CHIRON, having been designed for education, is a better fit in classrooms where we are interested in educating students on the breadth of program analysis. While we do not advocate that CHIRON is a right fit for all program analysis courses, we believe that instructors who have taught such courses would see the value of simplicity and breadth of techniques on a common representation that CHIRON offers, and will consider using it in their courses.

There do exist threats to validity to our findings. The number of students in an advanced course is not large, and so the sample size is too small to draw any statistical inferences from our study. However, a small class size is a reality for all advanced graduate level courses; we felt that this should not make us stop making innovations in graduate level courses as their impact can only be evaluated qualitatively. Also, as CHIRON was employed in a live course, we had to be careful to design our studies in a manner so as to not impact learning objective (the students are well exposed to all topics), uniform distribution of course-load (ensure that all student groups need similar effort to complete their problems) and fairness (the studies do not introduce any bias in our grading).

The course in which CHIRON was introduced was not a new course—it had been offered for more than a decade using production-quality tools before the offerings where CHIRON was introduced. The instructors feel that there is a substantial improvement in the “quality” of the assignments with CHIRON, and hence, the students’ assimilation of the topics. In the previous offerings, we could give only 2-3 assignments using tools like LLVM/Soot, and that too in similar topics like control/dataflow analysis, optimizations, and abstract interpretation as we were limited by the suitability of the framework(s) for certain tasks. CHIRON allowed the students to complete 5 assignments on a variety of topics. In terms of side effects, it was a welcome change that many of the students who took the recent offerings of this course (that used CHIRON) opted for doing their masters thesis in program analysis—in much larger number than in any of the past offerings. This indeed showed that they were inspired by this offering of the course—the lectures were the same, and the only component that changed was CHIRON being used for the programming assignments.

We are interested in improving and enhancing CHIRON, especially by adding more program analysis techniques to encourage its wider adoption in program analysis courses. In the near future, we are interested in adding static value flow analysis [88], [89], bounded model checking [90], [91], [92], [93], deductive verification [94], analysis of probabilistic programs [95], [96], [97], concurrency [98], [99], synthesis [100], [101], [102], [103], [104], [105], [106], [107] and dynamic analysis [108], [109], [110]. We are also interested in adding program analysis algorithms that combine ideas from multiple techniques, like {Fuzzing, Symbolic Execution} [111], [112],

[113], {Fuzzing, Deductive Verification} [114], {Abstract Interpretation, Synthesis} [115].

We have a release CHIRON available open-source, along with code sketches for the programming assignments, for use in other such courses. Further, we will be happy to support courses that are interested in using CHIRON by suggesting new problem statements, servicing bug-fixes, and feature requests. We also welcome feature enhancements, new analysis modules, and new problem statements from the community. We hope that CHIRON can form a wide platform for teaching program analysis technique and also grow by imbibing ideas from other instructors of similar courses.

ACKNOWLEDGMENTS

This work was supported, in part, by doctoral fellowships from Tata Consultancy Services and Intel, and a research fellowship award from Google. Any opinions, findings and conclusions in this article are those of the authors and do not necessarily reflect the views of the supporting organizations.

REFERENCES

- [1] UCLA, “CS 232 Static Program Analysis,” <http://web.cs.ucla.edu/~palsberg/course/cs232/F09/>, September 2009.
- [2] MIT-OCW, “Fundamentals of program analysis,” <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-820-fundamentals-of-program-analysis-fall-2015/index.htm>, September 2015.
- [3] CMU, “15-819 O: Program Analysis,” <https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/>, April 2013.
- [4] —, “17-819 Program Analysis,” <https://cmu-program-analysis.github.io/2022/index.html>, March 2022.
- [5] Cornell, “CS 6120: Advanced Compilers,” <https://www.cs.cornell.edu/courses/cs6120/2020fa/self-guided/>, September 2020.
- [6] UPenn, “CIS 700: Software Analysis and Testing,” <https://www.cis.upenn.edu/~mhnaiik/edu/cis700/index.html>, March 2018.
- [7] —, “CIS 673: Computer-Aided Verification,” <https://www.cis.upenn.edu/~alur/cis673.html>, March 2021.
- [8] UW, “CSE 501: Principles and Applications of Program Analysis,” <https://courses.cs.washington.edu/courses/cse501/15sp/index.html>, March 2022.
- [9] KDE/kturtle, “Educational programming environment that uses turtle-speak,” <https://github.com/KDE/kturtle>, March 2022.
- [10] Cornell, “Introduction - Bril: A Compiler Intermediate Representation for Learning,” <https://capra.cs.cornell.edu/bril/>, March 2022.
- [11] C. P. Language, “Cool tool,” <http://theory.stanford.edu/~aiken/software/cool/cool.html>, (Accessed on 02/17/2023).
- [12] P. Project, “Pintos introduction,” https://web.stanford.edu/class/cs140/projects/pintos/pintos_1.html, (Accessed on 02/17/2023).
- [13] W. A. Christopher, S. J. Procter, and T. E. Anderson, “The nachos instructional operating system,” in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, 1993.
- [14] R. T. Hays, J. W. Jacobs, C. Prince, and E. Salas, “Flight simulator training effectiveness: A meta-analysis,” *Military Psychology*, vol. 4, no. 2, 1992. [Online]. Available: https://doi.org/10.1207/s15327876mp0402_1
- [15] J. Whaley and M. S. Lam, “Cloning-based context-sensitive pointer alias analysis using binary decision diagrams,” in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004. [Online]. Available: <https://doi.org/10.1145/996841.996859>
- [16] The Coq Development Team, “Coq,” 2022. [Online]. Available: <https://coq.inria.fr>
- [17] G. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, 1997.
- [18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot - a java bytecode optimization framework,” in *CASCON*, 1999, p. 13.

- [19] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, 2010.
- [20] CodeQL, "CodeQL," <https://codeql.github.com/>, March 2022.
- [21] Polyglot, "Polyglot extensible compiler framework," <https://www.cs.cornell.edu/projects/polyglot/>, March 2022.
- [22] Randoop, "Randoop: Automatic unit test generation for java," <https://randoop.github.io/randoop/>, March 2022.
- [23] M. Zalewski, "American fuzzy lop," 2019. [Online]. Available: <http://lcamtuf.coredump.cx/afl>
- [24] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, Mar 2004.
- [25] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*.
- [26] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. [Online]. Available: <https://doi.org/10.1145/566172.566191>
- [27] D. S. Wilkerson and S. McPeak, "Delta," <https://github.com/dsw/delta>, June 2003.
- [28] B. R. Liblit, "Cooperative bug isolation," Ph.D. dissertation, University of California, Berkeley, 2004.
- [29] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, vol. 2988.
- [30] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems," in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, no. 1066, Oct. 1995.
- [31] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, Mar 2000. [Online]. Available: <https://doi.org/10.1007/s100090050046>
- [32] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, 2011.
- [33] G. A. Kildall, "A unified approach to global program optimization," in *POPL '73*, 1973.
- [34] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the ACM*, vol. 23, no. 1, jan 1976.
- [35] P. Cousot and R. Cousot, "Abstract Interpretation: A unified lattice method for static analysis of programs by construction of approximation of fixed points," in *POPL*, 1977.
- [36] OWASP, "Fuzzing," <https://owasp.org/www-community/Fuzzing>, April 2021.
- [37] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.
- [38] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, 2016.
- [39] R. Abreu, P. Zoetewij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, 2009.
- [40] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, 2013.
- [41] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium 13th ESEC/FSE*, 2011.
- [42] L. Foundation, "Logo (programming language)," https://el.media.mit.edu/logo-foundation/what_is_logo/logo_programming.html, 2000.
- [43] T. Academy, "Logo turtle," <https://turtleacademy.com/>, March 2022.
- [44] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.
- [45] P. Chatterjee, A. Chatterjee, J. Campos, R. Abreu, and S. Roy, "Diagnosing software faults using multiverse analysis," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, 2020, main track. [Online]. Available: <https://doi.org/10.24963/ijcai.2020/226>
- [46] J. Campos, A. Ribeiro, A. Perez, and R. Abreu, "Gzoltar: An eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012. [Online]. Available: <https://doi.org/10.1145/2351676.2351752>
- [47] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, 2014.
- [48] P. Chatterjee, J. Campos, R. Abreu, and S. Roy, "Augmenting automated spectrum based fault localization for multiple faults," in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023*, 2023. [Online]. Available: <https://doi.org/10.24963/ijcai.2023/350>
- [49] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," *ACM SIGPLAN Notices*, vol. 29, no. 6, 1994.
- [50] A. Solar-Lezama, "The sketching approach to program synthesis," in *Proceedings of the 7th APLAS*, 2009.
- [51] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005. [Online]. Available: <https://doi.org/10.1145/1065010.1065036>
- [52] J. W. Buck and S. Perugini, "An interactive, graphical simulator for teaching operating systems," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE, 2019. [Online]. Available: <https://doi.org/10.1145/3287324.3293756>
- [53] C. Dall and J. Nieh, "Teaching operating systems using code review," in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 2014. [Online]. Available: <https://doi.org/10.1145/2538862.2538894>
- [54] J. S. Mallozzi, "Thoughts on and tools for teaching compiler design," *J. Comput. Sci. Coll.*, vol. 21, no. 2, dec 2005.
- [55] A. J. Rossman and B. L. Chance, "Welcome to rossmanchance.com," <http://www.rossmanchance.com/>, March 2022.
- [56] Desmos, "Desmos," <https://www.desmos.com/>, March 2022.
- [57] GeoGebra, "Geogebra," <https://www.geogebra.org/>, March 2022.
- [58] JFLAP, "JFLAP Tool," <https://www.jflap.org/>, March 2018.
- [59] P. Tool, "Pate Tool," <https://users.cs.duke.edu/~rodger/tools/pateweb/>, March 2022.
- [60] A. Simulator, "Automaton simulator," <https://automatonsimulator.com/>, March 2022.
- [61] W. D. simulator, "Web DFA simulator," <https://web.cs.ucdavis.edu/~doty/automata/>, March 2022.
- [62] JELLRAP, "JELLRAP," <https://users.cs.duke.edu/~rodger/tools/jellrap/index.html>, March 2008.
- [63] LR(1) Parser Generator, "LR(1) Parser Generator," <http://jsmachines.sourceforge.net/machines/lr1.html>, March 2022.
- [64] L. P. Generator, "LL(1) parser generator," <https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/>, March 2022.
- [65] Y. Malysheva and C. Kelleher, "Assisting teaching assistants with automatic code corrections," in *CHI*, 2022. [Online]. Available: <https://doi.org/10.1145/3491102.3501820>
- [66] D. D. Garcia, M. P. Rogers, and A. Stefik, "Fun and engaging pre-cs1 programming languages," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '21. [Online]. Available: <https://doi.org/10.1145/3408877.3432570>
- [67] M. Fowler, E. T. Kraemer, Y.-S. Sun, M. Sitaraman, J. O. Hallstrom, and J. E. Hollingsworth, "Tool-aided assessment of difficulties in learning formal design-by-contract assertions," in *ECSE*, 2020. [Online]. Available: <https://doi.org/10.1145/3396802.3396807>
- [68] M. Ozkaya, "Teaching design-by-contract for the modeling and implementation of software systems," in *ICSOFT*, 2019. [Online]. Available: <https://doi.org/10.5220/0007950904990507>
- [69] D. Carvalho, R. Hussain, A. Khan, M. Khazeev, J. Lee, S. Masiagin, M. Mazzara, R. Mustafin, A. Naumchev, and V. Rivera, "Teaching programming and design-by-contract," 2020.
- [70] S. Strong, "Introduction to OOP (abstract): Teaching object-oriented programming concepts and skills using multi-media, computer-assisted instructions," in *OOPSLA*, 1992, p. 237. [Online]. Available: <https://doi.org/10.1145/157709.157817>
- [71] B. Alshaigy, S. Kamal, F. Mitchell, C. Martin, and A. Aldea, "Pilet: An interactive learning tool to teach python," in *WiPSCE*, 2015. [Online]. Available: <https://doi.org/10.1145/2818314.2818319>

- [72] B. Alshaikh, "Development of an interactive learning tool to teach python programming language," in *ITiCSE*, 2013. [Online]. Available: <https://doi.org/10.1145/2462476.2465601>
- [73] S. Combéfis, "Automated code assessment for education: Review, classification and perspectives on techniques and tools," *Software*, 2022. [Online]. Available: <https://www.mdpi.com/2674-113X/1/1/2>
- [74] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "Bugfix: A learning-based tool to assist developers in fixing bugs," in *ICPC*, 2009. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICPC.2009.5090029>
- [75] A. Aiken, "Cool: A portable project for teaching compiler construction," *SIGPLAN Not.*, vol. 31, no. 7, jul 1996. [Online]. Available: <https://doi.org/10.1145/381841.381847>
- [76] T. Golemanov and E. Golemanova, "A set of tools to teach language processors construction," ser. CompSysTech, 2020. [Online]. Available: <https://doi.org/10.1145/3407982.3407991>
- [77] B. Appelbe, "Teaching compiler development," in *SIGCSE '79*, 1979, p. 23–27. [Online]. Available: <https://doi.org/10.1145/800126.809546>
- [78] S. R. Vegdahl, "Using visualization tools to teach compiler design," in *CCSC*, 2000, pp. 72–83.
- [79] D. O'Brien, "Teaching operating systems concepts with systemtap," in *ITiCSE*, 2017. [Online]. Available: <https://doi.org/10.1145/3059009.3059045>
- [80] E. Isohanni and H.-M. Järvinen, "Are visualization tools used in programming education? by whom, how, why, and why not?" in *Koli Calling '14*, 2014.
- [81] E. Vidal Duarte, "Teaching the first programming course with python's turtle graphic library," in *ITiCSE '16*, 2016.
- [82] I. Arawjo, C.-Y. Wang, A. C. Myers, E. Andersen, and F. Guimbretière, "Teaching programming with gamified semantics," in *CHI '17*, 2017.
- [83] M. M. McGill and A. Decker, "Tools, languages, and environments used in primary and secondary computing education," in *ITiCSE 2020*.
- [84] M. Sharfuddeen Zubair, D. Brown, M. Bates, and T. Hughes-roberts, "Are visual programming tools for children designed with accessibility in mind?" in *ICETC'20*, 2020.
- [85] S. Suh, "Codetoon: A new visual programming environment using comics for teaching and learning programming," in *SIGCSE 2022*.
- [86] S. Framework, "Scratch - imagine, program, share," <https://scratch.mit.edu/>, March 2022.
- [87] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *OOPSLA*, 2009. [Online]. Available: <https://doi.org/10.1145/1640089.1640108>
- [88] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in llvm," in *CC*, 2016. [Online]. Available: <https://doi.org/10.1145/2892208.2892235>
- [89] —, "SVF Tool, GitHub," 2016. [Online]. Available: <https://svf-tools.github.io/SVF/>
- [90] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *TACAS*, 1999.
- [91] P. Chatterjee, J. Meda, A. Lal, and S. Roy, "Proof-guided underapproximation widening for bounded model checking," in *CAV*, 2022.
- [92] P. Chatterjee, S. Roy, B. P. Diep, and A. Lal, "Distributed bounded model checking," in *FMCAD*, 2020.
- [93] —, "Distributed bounded model checking," *Formal Methods in System Design*, 2022. [Online]. Available: <https://doi.org/10.1007/s10703-021-00385-1>
- [94] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, Oct. 1969. [Online]. Available: <https://doi.org/10.1145/363235.363259>
- [95] J. Bao, N. Trivedi, D. Pathak, J. Hsu, and S. Roy, "Data-driven invariant learning for probabilistic programs," in *CAV*, 2022.
- [96] S. Roy, J. Hsu, and A. Albarghouthi, "Learning differentially private mechanisms," in *42nd IEEE Symposium on Security and Privacy, SP 2021*, 2021. [Online]. Available: <https://doi.org/10.1109/SP40001.2021.00060>
- [97] Z. Susag, S. Lahiri, J. Hsu, and S. Roy, "Symbolic execution for randomized programs," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2. [Online]. Available: <https://doi.org/10.1145/3563344>
- [98] A. Verma, P. K. Kalita, A. Pandey, and S. Roy, "Interactive debugging of concurrent programs under relaxed memory models," in *CGO*, 2020. [Online]. Available: <https://doi.org/10.1145/3368826.3377910>
- [99] P. Deligiannis, A. Senthilnathan, F. Nayyar, C. Lovett, and A. Lal, "Industrial-Strength Controlled Concurrency Testing for C# Programs with COYOTE," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2023.
- [100] S. Roy, "From concrete examples to heap manipulating programs," in *Static Analysis - 20th International Symposium, SAS 2013*, vol. 7935. [Online]. Available: https://doi.org/10.1007/978-3-642-38856-9_9
- [101] A. Garg and S. Roy, "Synthesizing heap manipulations via integer linear programming," in *Static Analysis - 22nd International Symposium, SAS*, vol. 9291, 2015. [Online]. Available: https://doi.org/10.1007/978-3-662-48288-9_7
- [102] S. Verma and S. Roy, "Synergistic debug-repair of heap manipulations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, 2017. [Online]. Available: <https://doi.org/10.1145/3106237.3106263>
- [103] S. Verma and Subhajit Roy, "Debug-localize-repair: A symbiotic construction for heap manipulations," *Formal Methods Syst. Des.*, vol. 58, no. 3, 2021. [Online]. Available: <https://doi.org/10.1007/s10703-021-00387-z>
- [104] P. K. Kalita, M. J. Kumar, and S. Roy, "Synthesis of semantic actions in attribute grammars," in *2022 Formal Methods in Computer-Aided Design (FMCAD)*, 2022.
- [105] P. K. Kalita, D. Singal, P. Agarwal, S. Jhunjhunwala, and S. Roy, "Symbolic encoding of ll(1) parsing and its applications," *Formal Methods in System Design*, 2023. [Online]. Available: <https://doi.org/10.1007/s10703-023-00420-3>
- [106] G. Takhar and S. Roy, "Sr-sfl: Structurally robust stripped functionality logic locking," in *Computer Aided Verification*, C. Enea and A. Lal, Eds. Cham: Springer Nature Switzerland, 2023, pp. 190–212.
- [107] G. Takhar, R. Karri, C. Pilato, and S. Roy, "Holl: Program synthesis for higher order logic locking," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 3–24.
- [108] E. Jain and S. Roy, "Phase directed compiler optimizations," in *23rd IEEE International Conference on High Performance Computing, HiPC*, '16. [Online]. Available: <https://doi.org/10.1109/HiPC.2016.039>
- [109] R. Chouhan, S. Roy, and S. Baswana, "Pertinent path profiling: Tracking interactions among relevant statements," in *CGO*, 2013.
- [110] S. Roy and Y. N. Srikant, "Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm," in *Proceedings of the CGO, The Seventh International Symposium on Code Generation and Optimization*, 2009. [Online]. Available: <https://doi.org/10.1109/CGO.2009.11>
- [111] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016. [Online]. Available: <https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf>
- [112] A. Pandey, P. R. G. Kotcharlakota, and S. Roy, "Deferred concretization in symbolic execution via fuzzing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019. [Online]. Available: <https://doi.org/10.1145/3293882.3330554>
- [113] S. K. Muduli and S. Roy, "Satisfiability modulo fuzzing: A synergistic combination of smt solving and fuzzing," *Proc. ACM Program. Lang.*, no. OOPSLA2, 2022. [Online]. Available: <https://doi.org/10.1145/3563332>
- [114] S. Lahiri and S. Roy, "Almost correct invariants: Synthesizing inductive invariants by fuzzing proofs," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022. [Online]. Available: