Active Learning for Software Engineering

José P. Cambronero
CSAIL MIT
USA
jcamsan@mit.edu

Thurston H. Y. Dang
CSAIL MIT
USA
tdang@mit.edu

Nikos Vasilakis[*]
CSAIL MIT
USA
nikos@vasilak.is

Jiasi Shen
CSAIL MIT
USA
jiasi@csail.mit.edu

Jerry Wu[†]
Google
USA
werryju@mit.edu

Martin Rinard
CSAIL MIT
USA
rinard@csail.mit.edu

Abstract

Software applications have grown increasingly complex to deliver the features desired by users. Software modularity has been used as a way to mitigate the costs of developing such complex software. Active learning-based program inference provides an elegant framework that exploits this modularity to tackle development correctness, performance and cost in large applications. Inferred programs can be used for many purposes, including generation of secure code, code re-use through automatic encapsulation, adaptation to new platforms or languages, and optimization. We show through detailed examples how our approach can infer three modules in a representative application. Finally, we outline the broader paradigm and open research questions.

CCS Concepts • **Software and its engineering → Software development techniques**.

Keywords program inference, program modeling, active learning

ACM Reference Format:
José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin Rinard. 2019. Active Learning for Software Engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '19), October 23–24, 2019, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3359591.3359732

1 Introduction

Software engineering as practiced today is a laborious and error-prone process. Programmers are faced with large, poorly documented (and hard to inspect) systems. This makes working with such codebases an unwieldy exercise, negatively impacting correctness, runtime performance, and development cost.

We propose an alternative paradigm: using active learning to automatically infer program functionality. By feeding a program carefully crafted inputs and observing its behavior, we can capture the program's core semantics in a form amenable to analysis, augmentation, and engineering.

1.1 Dimensions

Our work aims to improve three key dimensions of software engineering artifacts: correctness, runtime performance, and development cost.

Correctness: Inferring the semantics of an application from its execution facilitates creating an inferred program that satisfies the *de facto* specification [8], which reflects the behavior experienced and expected by its end-users.

Depending on the application and our inference process, the inferred program will not identically replicate the original application. However, this behavior can be an advantage. For example, an application could be hard-coded to only process images of a fixed width and height, while the inferred application can be generalized to varying sizes.

Security can be considered a special case of correctness. Vulnerabilities such as buffer overflows can be exploited to corrupt the data or control-flow of the program, causing unintended behavior. Our framework can use a subset of inputs to reduce the attack surface and the regenerated code can be designed to be safe-by-construction.

Runtime Performance: Our systems infer the functionality of a program, divorced from its underlying implementation or language. This frees our techniques from the constraints of legacy codebases, allowing generation of new code that can be highly optimized by the latest compilers.

Moreover, in principle, our inference systems can perform more fundamental optimizations at the algorithmic level. For

[*] Work was done while the author was with the University of Pennsylvania.
[†] Work was done while the author was with MIT.

Onward! '19, October 23–24, 2019, Athens, Greece

ACM ISBN 978-1-4503-6995-4/19/10…$15.00
https://doi.org/10.1145/3359591.3359732

example, consider a legacy binary that performs bubble sort. An inference system that only observes the active behavior of the program could determine that a sorting operation was performed, and emit code that uses a more efficient sorting algorithm, such as mergesort.

Development Cost: Software engineering with active learning can help bridge the gap between the intent of the programmer and the concrete instantiation of their vision. By viewing the active behavior as the specification, our paradigm allows the programmer to code their system in any language and then have its semantics extracted and automatically re-expressed in any other target language.

1.2 Limitations of Current State of the Art

Prior efforts to improve on these dimensions have encountered difficulties—*e.g.*, non-trivial generalization challenges across languages [41]—addressed using a combination of manual annotations [24, 25], full formal specifications [45, 54], and significant human intervention or source code access for reverse engineering [44],

The active-learning-based approach to software engineering that we propose allows us to circumvent such difficulties: as we infer the application, there is no need for a specification beyond the one captured by the program's behavior; and since we can modify the inferred application freely, we can add (or remove) checks as appropriate, reducing performance costs. We provide further details on the current state of the art and how it compares with our approach in Section 9.

1.3 Structure of the Paper

We begin by introducing key background concepts in active learning and program inference (Section 2). We then outline our paradigm by considering a social network application (Section 3), and show how to infer a small, but diverse, set of its components using three different techniques and systems (Section 4–6):

- Section 4 presents a technique to infer and regenerate binary data parsers, similar to the one needed by the social network application.
- Section 5 discusses Shen and Rinard's [73] technique for inferring programs that use databases, and how this can be applied to the account management back-end of our social network application.
- Section 6 presents a technique for upgrading a program that uses in-memory data structures such as maps and lists to one that uses a database. This upgrade is done without changing the original program to use an external framework, such as one for Object-Relational Mapping (ORM).

These example components illustrate several benefits of our proposed approach. For example, parsers are a significant source of security vulnerabilities in programs [53, 61]; and database interfaces are known sources of development and maintenance cost. Our regenerated programs can be safe-by-construction—automatically augmented with security checks—while significantly simplifying the development process.

We present the general paradigm of active learning for software engineering (Section 7), including design choices for the observation, inference and generation models, discuss research challenges and threats to validity (Section 8), and compare with prior work (Section 9).

2 Background

We introduce background concepts employed throughout the remainder of the paper. We outline program inference, active learning broadly and its application to program inference.

2.1 Program Inference

We define *program inference* as the task of modeling a computation by repeatedly interacting with, and observing the behavior of, an existing implementation of that computation. The inference technique interacts with the existing implementation by executing a *command*, *i.e.*, a procedure or function, that has been exposed by the implementation's API (*e.g.*, the `main` function of a program written in C).

A *hypothesis* corresponds to a possible model of that computation (*e.g.*, a partial program), which is said to be *consistent* with an execution if the execution's behavior under the model matches the behavior in the original program. We say there is *inference ambiguity* when there are two or more hypotheses that are consistent with the implementation's observed executions and may not be equivalent to each other. Absent any additional information, either hypothesis could represent the target computation. To resolve this ambiguity we may attempt to induce an execution that can rule out competing hypotheses.

A crucial difference between program inference and other automated programming techniques, such as inductive program synthesis [62], is that program inference assumes an implementation of the desired computation already exists and is available during inference. This reference implementation eliminates the need for examples or a formal specification.

2.2 Active Learning

Active learning refers to a setting in which a learning algorithm is free to query an oracle or environment to obtain additional information for the learning task [69]; often this information comes in the form of ground-truth labels associated with examples. For example, a system may request labels for examples with low-confidence predictions [46] or

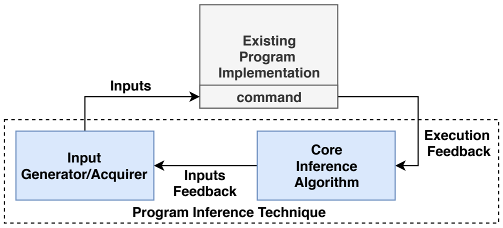

Fig. 1. Active learning in program inference. At a high-level, the program inference technique incorporates execution feedback from the existing program implementation and then actively picks new inputs to refine the hypothesized model of the underlying computation.

examples that most reduce the space of possible models [37]. The goal of active learning, broadly construed, is to improve the ability of the learning algorithm to efficiently use the available data. It is important to note that active learning is an addition to, not a replacement of, the core learning technique (*e.g.*, a standard classifier can be trained with or without active learning).

2.3 Program Inference and Active Learning

As discussed previously, a core challenge in program inference is to resolve the ambiguity between possible hypotheses. Rather than depending on a fixed corpus of inputs, program inference has the advantage of having access to a "cheap" oracle: the existing implementation. This oracle can be used to actively "label" different inputs with their corresponding ground-truth behavior. Furthermore, the inference technique can *actively* construct inputs that are devised to increase the likelihood that the hypothesized model can be refined. Active learning, thus, provides a grounded approach to exploring and refining the inferred program.

Figure 1 provides a schematic of this high-level algorithm, which motivates the rest of our paper. The program inference technique consists of a core inference algorithm, which incorporates execution feedback from the existing implementation, and a method for acquiring/generating inputs to improve the hypothesis. In the following sections we discuss three different techniques that instantiate this higher-level algorithm to infer different types of programs.

For reference purposes, Figure 2 presents a summary of key terms introduced.

3 Active Learning at Scale

We use a modern social network application to illustrate the challenges of active learning at the scale of a production-grade application (Section 3.1), explain how module-level decomposition can make such active learning tractable (Section 3.2), and showcase an end-to-end example of learning and regeneration (Section 3.3).

program inference: Producing a model of a computation, in the form of a program, by repeatedly interacting with, and deriving hypotheses from, an existing implementation of that computation.

command: An API function or procedure that can be invoked to interact with an existing program implementation and execute the target behavior (*e.g.*, the `main` function in a program written in C).

consistency: We say an inferred model is consistent with an execution, if the behavior when executing the inferred model matches that of the original program.

inference ambiguity: When two or more program hypotheses are consistent with observed executions, but may not be equivalent to each other.

active learning: Augmenting a learning algorithm to query/obtain information from an oracle or environment in a directed manner.

Fig. 2. Glossary. A summary of key background terms.

3.1 Challenges and Opportunities

Consider a modern social network application like Facebook. Such an application must provide a dizzying array of functionality—*e.g.*, user accounts, news feeds, photos, group events, search queries—accounting for millions of lines of code. For example, Facebook's source code in 2011 totaled more than nine million lines [63], spanning different logical subsystems and semantic levels, and supported by thousands of engineers (and open-source backing). How could active learning infer and regenerate such an application?

A key insight is that modern applications, even when seemingly monolithic, are highly modular—just the front-end code of the aforementioned Facebook example was over 10K modules in 2011 [67]. Software modules are development-time constructs that abstract away inessential details and present well-encapsulated interfaces to their consumers. They provide implicit, fine-grained guidelines for breaking down the complexity of active learning.

Pervasive modularization is a relatively recent phenomenon. Recent research [80, 89] shows that today's software is primarily composed of third-party modules, with large applications numbering in the thousands to tens of thousands of modules. Even more impressive are the statistics of language-specific package repositories. JavaScript hosts more than 1M packages from over 100K authors and serves billions of package downloads per day [68]. Such extensive modularity, the use of encapsulation, and pervasive reuse of code in today's applications translates to several opportunities for active learning:

- **Divide and Conquer:** Recursive module imports simplify active learning and regeneration, as complex modules are

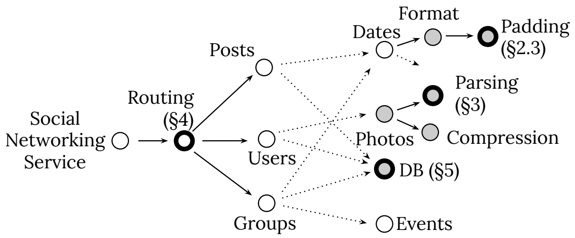

Fig. 3. Active learning of a complex application. Decomposition into smaller modules is accomplished top-to-bottom (root-to-leaves), whereas regeneration is bottom-to-top (leaves-to-root). Non-shaded vertices denote vanilla components, shaded vertices denote regenerated components, and dotted edges denote omission of layers. The modules highlighted in bold are exemplified in Sections 4–6.

composed by many simpler ones—to the point that leaf-level, elementary modules can be inferred directly.

- **Interface Reuse:** Module re-use within a single application translates to observing active program behavior in multiple settings, significantly aiding active learning.
- **Fine Granularity:** Multi-thousand-module dependency trees enable visibility into an application's behavior at a high resolution in time and space: (i) at every function call entering a module, and (ii) on thousands of modules across an application.
- **Clear Boundaries:** Modules today encapsulate state behind small and tight interfaces, with clear self-contained components. This structure simplifies capturing every observable behavior, and lowers the cost of interacting with the target program.
- **Parallel Learning:** Module state at the same level in a dependency tree is independent, allowing parallel inference and regeneration.

The next section returns to the social-network application, illustrating one possible way of putting these ideas into practice.

3.2 Breaking Down into Modules

We start by unfolding a complex application or module into smaller modules (top-down), so as to leverage recursive active learning and regeneration. Inferred modules are then composed into larger modules, leading to regeneration of the entire application.

Figure 3 presents an overview of this process using the (simplified) module structure of a social network application. Vertices represent modules. Edges depict their interdependencies: parent (caller) modules importing child (callee, consumer) modules. Dotted edges imply multiple levels of depth, and thicker vertices highlight modules used as detailed examples of active learning in the rest of the paper. Features such as user accounts are implemented as separate modules that draw their functionality from other modules further down in the dependency tree. For example, `Photos` imports functionality to parse and compress images.

```
txfm (e: Field) : Field ← match e with
  | λ.f → (...args).{v ← f(args) // alr(args, v)}
  | {(s, v) :: vs} → {(s, txfm v) :: txfm vs}
  | [v :: vs] → [(txfm v) :: txfm vs]
  | _ → interpose(e)
end
```

Fig. 4. Interposition transformation for module-level active learning. The type structure (in *turquoise*) of module interface elements is used for pattern matching. In the first pattern, `alr()` passes the input arguments and return value (`args`, `v`) of every function `f` at the module boundary to the active learning algorithm of choice; this learning executes concurrently with the main call without blocking calls into the module (*Cf.*§3.2). The next two patterns apply this transformation recursively, as module return values are objects of arbitrary nesting. Finally, `interpose` wraps any primitive-value entry points to the module (*e.g.*, assignments).

A first step towards module-level active learning is to interpose at the module boundaries. Such interposition can be achieved automatically, for example, by detecting and transforming module interfaces. The specifics of these transformations depend on the language setting, with compiled and dynamic languages providing appropriate mechanisms (*e.g.*, Template Haskell, Rust's macro system, name (re-)binding, and dynamic code evaluation).

In both language settings, a module's return value is transformed to automatically capture active program behavior, in this case in the form of input-output pairs. This transformation is structured similar to Figure 4: the module interface is augmented to pass call arguments and return values to the active learning infrastructure (`alr`). (The complexity of transformation stems from the fact that a module return value can be an object of arbitrary nesting.) The active learning infrastructure could (i) generate a preliminary template for the module, and (ii) instantiate the correct subsystem—*e.g.*, *Nero* (Section 6), *Konure* (Section 5), *etc.*

Further transformations (not detailed) aim to detect module accesses to their surrounding environment—including accesses to global variables, top-level objects, *etc.*—and capture a module's *full* observable behavior. One way to achieve this is by shadowing and transforming free variables in the module environment during the module-loading process [79, 81].

Once the learning and regeneration completes, the decomposed structure can be used to replace the module at runtime. To achieve this, the active learning subsystem sends a notification to the transformation system that a module has been generated. The process could overwrite the pointer to the wrapper-internal module so that (i) the consumer modules remain intact, and (ii) the system keeps feeding the active learning subsystem with data from subsequent calls.

3.3 A Teensy Module: String Padding

To further understand these transformations, we zoom into the `Format` module within the larger social networking application. The interaction between `Format` and one of its

```
1 let pad = import("Padding");
2 …
3 year: pad(year, 2, ' '),
4 day: pad(day, 2, ' '),
5 …
```

Format

```
1 f = (str, len, c) => {
2  let p = c.repeat(len);
3  return p + str;
4 }
5 export = f;
```

Padding

Fig. 5. Zooming into Figure 3's Format–Padding boundary. Padding exports a single function, which is used several times in Format (*Cf.*§3.3).

dependencies, Padding, is shown in Figure 5: Format calls Padding several times, padding different date-related strings.

Using the techniques described (Section 3.2), our system first interposes on all imports of the application—including the Padding import (Figure 5 left, line 1). Instead of returning the original return value, our system introspects on the return value and applies a transformation pass over it. A first pass infers a coarse type signature for Padding—essentially that the module returns a single function, as opposed to, say, a set of functions or a list of Date objects.

A second transformation pass wraps Padding with a function that interposes on all calls, recording inputs and outputs. The wrapper observes that the function is always provided two arguments (string and int), and at times a third argument (string). Leveraging active learning over the program's behavior, in this case captured through input-output pairs, the inference algorithm is able to detect that the first argument is always the suffix of the result, the second argument denotes the spaces prefixing the string, and that the third argument alters the padding character. Replacing Padding with, say, the recently ES7-standardized padEnd method would improve both the performance and the security of the larger application.

The Format–Padding duo is a trivial case—their implementation code amounts to less than 50 lines—used to illustrate the application of transformations and tangible benefits of regeneration. The next three sections walk through the active learning process of modules that are significantly more complex. These are highlighted in Figure 3: a binary data parser for photos (Section 4), a module that accesses the database of users' posts (Section 5), and a stateful *seed* for a new module with lists and maps (Section 6).

4 Inferring Binary Data Parsers

The social networking application, introduced in Section 3, provides image sharing functionality. These images are stored as binary data, with parsing details that vary by image type and their end-use. The social network application must validate input *and* populate key data structures, defined across different functions, with values from the input.

Writing parsers and data processors is a notoriously difficult software engineering task [11, 12, 29, 72]. Often this logic is scattered throughout a program, yielding what is commonly termed a "shotgun" parser, a known source of security vulnerabilities [53]. Other times, the input validation performed is insufficient to remove vulnerabilities [61].

Our active learning approach provides a framework to infer the original parsing functionality from the application and a collection of inputs. In particular, by exploring the data structure definitions produced by different inputs, we can build up a generalization for these definitions, which represents the desired parsing functionality.

4.1 Image Sharing Service

The image service uses a library, *stb*, to load bitmap (.BMP) images. The application calls the stbi__bmp_load function defined in stb_image.h. stbi__bmp_load parses the image file header using stbi__bmp_parse_header and then parses the remainder of the file using stbi__bmp_load_cont . stbi__bmp_load_cont populates two struct pointers and three integer pointers, and returns a pointer to an array containing the image pixel data.

Figure 6 shows a (simplified) excerpt of code from the stbi__bmp_load_cont function that highlights some of the parsing challenges. The code reads the image's pixel data and stores it in an output array out. To do so, the code must branch between different parsing definitions, extract the appropriate number of bytes (and keep track of the file pointer) based on the image's rows and columns, cast these bytes to the correct type, store them in the correct position in the output array, and keep track of certain image properties (in this case, the use of the alpha channel).

We now walk through the steps that an active learning technique can use to infer this parsing functionality. We implemented these steps in a prototype system called *BDP* (short for Binary Data Parser).

4.2 Approach

Figure 7 presents a block diagram outlining our technique. We begin with a set of example images. The original image parsing module, depicted in gray with bold dotted edges, takes input files (images), reads their data and populates the data structures used later on by the compression function. Our technique, depicted in blue boxes with dashed outlines, layers on additional components to extract behavioral information from this execution and recover patterns that yield the parsing function. We walk through each step in the pipeline and its use in the broader technique.

Dynamic Analysis: Each input file is executed through the parsing function. The application has been modified with dynamic instrumentation tools to produce execution traces. These traces contain concrete expressions relating the input and output data structure bytes. After executing each input file, we have a collection of such traces.

Our technique relies on observing a large space of input examples and their corresponding execution traces. However, rather than randomly acquiring new examples, the inference procedure guides the choice of examples that provide new information by choosing input files that cannot be parsed

```
for (j=0; j < (int) s->img_y; ++j) {
   // Branch between parsing cases.
   if (easy) { // No bitpacking
      for (i=0; i < (int) s->img_x; ++i) {
         out[z+2] = read8Bits(s); // blue
         out[z+1] = read8Bits(s); // green
         out[z+0] = read8Bits(s); // red
         z += 3;

         // Optional alpha channel
         unsigned char alpha = (easy == 2 ? read8Bits(s) : 255);
         all_alpha |= alpha;
         if (out_bytes_per_pixel == 4) out[z++] = alpha;
      }
   } else { // Bitpacking e.g., 16-bit images.
      // Use header information
      int in_bytes_per_pixel = info->bpp;
      for (i=0; i < (int) s->img_x; ++i) {
         // Decode appropriate number of bytes
         uint32 pixel = (in_bytes_per_pixel == 16
                         ? read16Bits(s)
                         : read32Bits(s));
         unsigned int a;
         // shiftsigned extracts the bits corresponding to the red/green/blue
         // channels, and scales it to between [0,255]
         out[z++] = shiftsigned(pixel & mask_red,
                            red_shift,   red_count));
         out[z++] = shiftsigned(pixel & mask_green,
                            green_shift, green_count));
         out[z++] = shiftsigned(pixel & mask_blue,
                            blue_shift,  blue_count));

         alpha = (mask_alpha
                      ? shiftsigned(v & mask_alpha,
                                    alpha_shift, alpha_count)
                      : 255);
         // Track properties of input read so far
         all_alpha |= alpha;
         if (out_bytes_per_pixel == 4) out[z++] = alpha;
      }
   }
   stbi__skip(s, pad);
}
return (out);
```

Fig. 6. Simplified excerpt from `stbi__bmp_load_cont`, which reads in the image’s pixel data into an array. This requires that the developer distinguish between different parsing cases, read at the appropriate file offset, cast sequences of bytes into the appropriate type, and store these in the correct order in the output array.

with the current inferred model, and input files that match conditions for which we have few existing examples.

Domain-Specific Language: Effective inference, across the techniques we present in this paper, relies on having a productive representation of the input and the application behavior we want to capture. As shown in this and other techniques here, carefully designing a domain-specific language (DSL) that enables expression of important program features and appropriately constrained generalization is a key technical contribution.

BDP uses such a DSL, which describes sequences of memory and maps input/output bytes, to describe the data structures built up by the target application. This DSL abstracts away implementation details such as field names and allows us to implement a simple inference procedure. We refer to programs in our DSL as *Data structure Definition Programs* (DDPs).

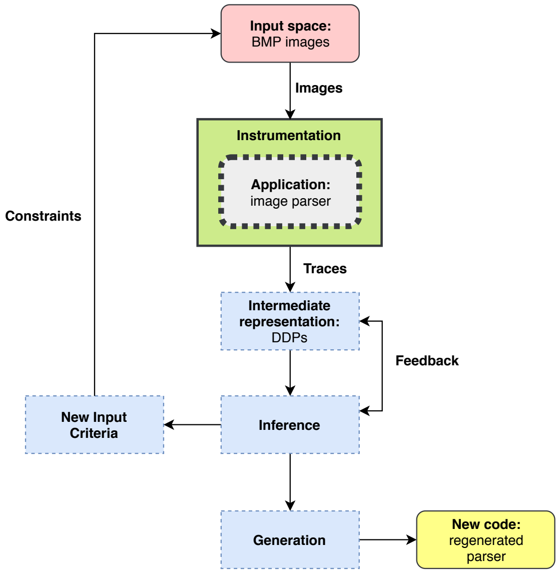

Fig. 7. *BDP* system. *BDP* is an active learning technique (and system) used to infer the image parsing functionality and generate code that implements that parsing function. Our technique combines: dynamic program analysis, DSL design, inference, and code generation. With respect to the high-level paradigm (Figure 1): the *Intermediate representation* and *Inference* boxes instantiate the *Core Inference Algorithm*, and the *New Input Criteria* box and *Constraints* arrow instantiate the *Input Generator/Acquirer*.

For example, the following DDP can describe a struct with two integer fields that are read directly from the input file

```
out[0] = ZeroExtend(64, Read(0, 4))
out[1] = ZeroExtend(64, Read(4, 8))
```

where the word at index 0 is mapped to an expression that reads an int stored in bytes 0 through 3 (inclusive) and extends it to a 64-bit word, similarly for index 1.

Single Input Inference: The DDPs produced by executing the application and analyzing its traces can then be used to infer common properties of the input files and how they are parsed by the original module.

We distinguish between properties that can be inferred on the basis of each individual DDP and properties that can only be resolved when multiple DDPs are considered.

For this parsing function, we can infer repetition in the way groups of bytes are defined, which must account for the strides in the data structure that is being populated and the strides in the original input file. Figure 8 shows the layout of pixel data in the array populated by the parsing function and compares this with the layout for that data in the original input file. Bitmaps are commonly stored “bottom-up” (upside-down), with the file starting with the bottom row of the image. The output array, however, is “top-down”, so it is necessary to iterate in the zig-zag pattern shown with dashed lines. Additionally, while the input file has color channel information laid out as blue–green–red–alpha (BGRA), this must be laid out as "RGBA" in the output array. Our inference

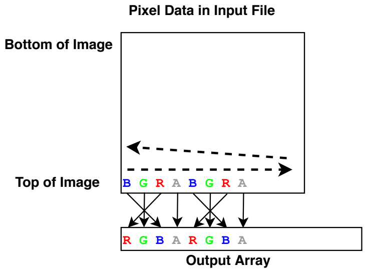

Fig. 8. Patterns between input files and data structures. Identifying repetitions requires inferring relationships between the DDP and the input file, in particular identifying shared expressions, strides in the data structure, and strides in the input file.

procedure learns these kinds of relationships and generalizes them where appropriate.

We can carry out the repetition inference procedure repeatedly to identify nested repetition patterns.

At this point, we can represent a portion of that input file as a DDP with repetition. We can also automatically annotate each number of repetitions with candidate expressions over the input images' bytes. This means that rather than provide only a concrete number of iterations for a repetition, we can associate it with values in the input file. However, there may be multiple expressions that yield the same concrete number of iterations, so this must be refined later on.

Multiple Input Inference: Multiple key parsing features cannot be inferred from a single example. A typical case is *parsing flags*, values in the input file that are used to switch between data structure definitions in the original parsing function. For example, Figure 6's `easy` is defined over file values (elided in the figure for simplicity), branching between two possible ways of populating the `out` output array.

Parsing flags are useful as they allow us to partition the input space. The intuition is that input files that share parsing flag values should be parsed using the same set of operations. Using this intuition, we formulate an iterative partitioning algorithm, that partitions the input files into groups, infers commonalities (*e.g.*, shared expressions), and validates these inferences.

The feedback automatically obtained by *BDP* allows us to progressively improve the inferred partitions and their shared representation. Once partitioning is finalized, combining the inferred model for each partition with the flag values fully represents the observed parsing behavior.

New Inputs for Active Learning: The advantage of active learning is that we can acquire or produce new inputs for our target application. For *BDP*, we can always produce a parser that is guaranteed to parse at least a subset of the training files. Rather than simply provide files at random to the inference algorithm, we can use the partial parser to identify files for which the current model fails, and we can choose input files in different ways, such as the smallest file or the file associated with a high-error partition.

Code Generation: Our model DSL can define the contents of data structures in terms of input bytes, and provides a limited set of control flow operations, such as bounded loops (with bounds determined by concrete values or values based on input file bytes) and branching on an equality expression over the values of particular input file bytes (*i.e.*, parsing flags). This language is used to define a set of parsers that can handle different input types, which are then grouped into a single parser by defining branching conditions that lead to each sub-parser.

During the code generation phase, we compile this DSL program into a C program using a simple translation approach, as every construct in the DSL maps in a straightforward fashion to constructs in popular imperative languages.

Figure 9 shows an excerpt of the C code generated to populate the pixel data array for a 16×16 32-bit bottom-up bitmap.

The generated program can now replace the original parsing function in the application. The code generated has multiple advantages over the original parsing function: it is more compact, the mapping between input bytes and data structure contents is more explicit, and it uses simple array expressions instead of pointer arithmetic.

Evaluation: We implemented the technique presented here in a system: *BDP*. We have successfully generated parsing functions using *BDP* for two popular image applications: `catimg` and `potrace`.

User Manual Input: Currently, to use *BDP*, the user must provide a corpus of inputs that can be used to exercise the application. The active learning process selects files to run from this corpus during inference. Additionally, the user must indicate the function (or functions) in the target application that build the data structures of interest to facilitate their instrumentation. Finally, the user must provide *BDP* with the command used to execute the application (and which *BDP* will repeatedly call with different input files).

5 Inferring Modules that Access Databases

Many applications access a database as an integral part of their execution. Databases provide advantages such as optimized data retrieval and storage, and abstraction of the underlying data structures and their layout. However, developing a database-backed module presents challenges: the core functionality must be mapped to database operations, database queries and inputs must be appropriately sanitized to reduce security vulnerabilities [70], and the developer must determine the appropriate database system to back their application.

Shen and Rinard [73] addressed this challenge with their *Konure* system, which observes the active behavior of the

```
int64_t expr_0 (int inputStartIndices [1], long file) {
    int8_t INPUT_0;
    readBytes (&INPUT_0, file, inputStartIndices [0], 1);

    // (assert (let ((?x0 ((_ sign_extend 56) INPUT_0)))
    //                (= __dfsan_top_level ?x0)))
    int8_t var3003 = INPUT_0;
    int64_t var3002 = ((int8_t) var3003); // sign_extend
    int64_t INTERNAL_VAR0 = var3002; // let clause

    return (INTERNAL_VAR0);
}

int MIN_X = 0;
int MIN_Y = 78;
int MIN_Y0_0 = MIN_Y + 2;

int LOOP_BOUND_A = 16;

int LOOP_BOUND_B = 16;
int MULTIPLIER_B_0 = 4;
int MULTIPLIER_B_2 = 4;

int MULTIPLIER_C_0 = 64;
int MULTIPLIER_C_1 = -64;
int ADDEND_C_1 = -15;

for (int indexB = 0; indexB < LOOP_BOUND_B; indexB++) {
    int NUMERIC_B_1 = indexB * MULTIPLIER_C_0;
    int NUMERIC_B_3 = (indexB + ADDEND_C_1) * MULTIPLIER_C_1;

    for (int indexA = 0; indexA < LOOP_BOUND_A; indexA++) {
        int NUMERIC_A_0 = indexA * MULTIPLIER_B_0 + NUMERIC_B_1;
        int NUMERIC_A_1 = indexA * MULTIPLIER_B_2 + NUMERIC_B_3;

        int x0 = NUMERIC_A_0 + MIN_X;
        int y0_0 = NUMERIC_A_1 + MIN_Y0_0;
        out [x0] = expr_0 (y0_0, file);

        int x1 = x0 + 1;
        int y1_0 = y0_0 + (-1);
        out [x1] = expr_0 (y1_0, file);

        int x2 = x0 + 2;
        int y2_0 = y0_0 + (-2);
        out [x2] = expr_0 (y2_0, file);

        int x3 = x0 + 3;
        int y3_0 = y0_0 + (1);
        out [x3] = expr_0 (y3_0, file);
    }
}
```

Fig. 9. Generated parsing code using *BDP*. Code to load pixel data from an image for the application's image sharing service, generated from the inferred model.

module and pattern of database accesses, in order to infer the module's functionality.

In this section, we show how *Konure* can be viewed as an instance of active learning for software engineering, by demonstrating how we can apply *Konure* to infer the social network application (Section 3) module that stores user login and post information to a database.

5.1 Users' Postings Service

The social network application uses a module to store users' login information and posts in a database. The module exposes a command line interface (CLI):

```
listuserposts -u <user-id> -p <password>
```

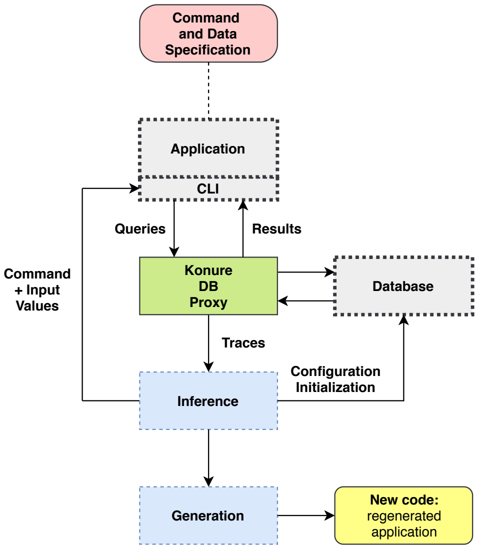

Fig. 10. *Konure* system diagram. *Konure* can be used to infer the functionality of applications that access a database. With respect to the high-level paradigm (Figure 1): the *Inference* box and *Configuration Initialization* arrow instantiate the *Core Inference Algorithm*, and the *Inference* box and *Command + Input Values* arrow instantiate the *Input Generator/Acquirer* box.

which lists a user's posting information, when given the user ID and corresponding password. To implement this command, the application queries the `users` table, which contains user ID (primary key), name and password information, and the `posts` table, which contains entries of the form (user ID, post ID) to reflect the application's post history across all users.

Such an application could be written in any language, as long as it reads from a MySQL database [83] and its behavior can be expressed in *Konure*'s DSL.

5.2 Approach

Figure 10 shows a high-level diagram of the *Konure* technique and how it infers the target commands. Similarly to the *BDP* pipeline (Figure 7), the original application (along with the database it uses) is detailed in grey boxes with bolded dotted edges, while the inference components are detailed in light blue boxes with dashed edges.

Application and Specifications: *Konure* takes the social network application along with a command and data specification. The command specification provides the command line interface through which *Konure* can interact with the application and the input parameter format expected. For example, *Konure* can execute `listuserposts` with varying values for user ID and password. The data specification provides database schema information, such as the tables and columns available with their datatypes, which *Konure* uses to generate configurations for the database and its contents. Figure 11 shows an example of such a command and data specification.

```
{
  // Data specification
  "schema": [
    {"table":"users", "columns": [
      {"name": "id", "type": "int", "key": "primary"},
      {"name": "password", "type": "str"},
      ...]
    },
    {"table":"posts", "columns": [...]},
    ...
  ],

  // Command specification
  "signature": {
    "platform": "shell",
    "path": "/big-org/userdb/",
    "apis": [
      {"name": "listuserposts", "command": "java -cp src/java...
     listuserposts -u [arg_u], -p [arg_p]"},
      ...]
  }
}
```

Fig. 11. ***Konure*** **specification example.** *Konure* takes a data and command specification, such as the one shown here, and uses the specification to configure interactions with the application being inferred.

Monitor Communications: *Konure* takes a greybox approach – monitoring a database proxy – to observe the application's execution. For each command execution, *Konure* observes the inputs, the outputs, and the application's communications with its database. These interactions are transparent to the original application.

Konure initially executes the command with parameters: `listuserposts -u 0 -p 1`. Future executions contain values that allow *Konure* to explore the application. The command is executed on an initially empty database. When it executes, *Konure*'s database proxy records the SQL query `SELECT * FROM users WHERE id = '0'`, as well as the fact that the query retrieves no data. This information constitutes the application's concrete trace for this command execution.

Domain-Specific Language: *Konure* uses a carefully designed domain-specific language to represent applications in a constrained search space and enable efficient inference. The DSL can capture sequences of queries (which may reference prior queries' results), conditional statements which execute based on a test condition for non-empty query results, and iteration over query results in the form of bounded loops.

During inference, *Konure* maintains a hypothesized program, expressed in this DSL, which is derived to be consistent with all observed traces. In this case, having observed the query `SELECT * FROM users WHERE id = '0'`, *Konure* hypothesizes that this query could either be a query followed by a yet-to-be derived subprogram of additional statements or be part of a conditional statement's guard. The latter would correspond to the following program hypothesis:

```
if y1 <- select users.id, users.password,
         users.firstname, users.lastname
         where users.id = u
then { ?? } else { ?? }
```

where each "??" is a yet-to-be-explored subprogram, or a DSL non-terminal. Note that the concrete value `'0'` has been replaced with a data origin (the input parameter `u`), and the columns for `users` are explicitly included in the select statement, rather than using the `*` wildcard. *Konure* identifies such rewrites using a trace abstraction procedure.

Trace Rewriting: In addition to abstracting out concrete values of a trace with their data origins, *Konure* also rewrites the trace to summarize loop information. For example, if a concrete trace contains the same query, repeated multiple times, then *Konure* may hypothesize that the query is executed as part of a loop body.

Inference Algorithm: *Konure*'s hypotheses contain unexplored subprograms (DSL non-terminals), marked ??. After executing the command a few times, *Konure* expands a ?? with the appropriate DSL production that is consistent with the execution traces. If this expansion produces more ??s, *Konure* executes the command more times and expands each ?? recursively. This recursive, hierarchical expansion continues until all non-terminals have been expanded and the final program has been identified.

For any program that conforms to the DSL, the algorithm terminates and produces a program that correctly models the original functionality.

New Inputs: To exercise unexplored subprograms and resolve inference ambiguity, *Konure* generates input values and database contents that induce different executions. In our user posting example, *Konure* produces values such that the select query from the `users` table does not produce empty results. In this case, that corresponds to providing an input parameter for user ID (`u`) that has a corresponding entry in the `users` table. In the hypothesis that contains the conditional statement, this would induce execution of the `then` branch (which was previously unexplored).

Konure selects these values by generating suitable constraints and solving them with an SMT solver.

Generating Code: Inference terminates once all non-terminals in the hypothesized program are expanded, indicating that no unexplored paths remain. *Konure*'s inferred program can express database queries, bounded loops over query results, and conditional statements based on non-empty query results. These DSL constructs map straightforwardly to constructs in popular imperative languages. *Konure* currently generates Python code.

Evaluation: *Konure* has been evaluated on five applications, including a chat room, task manager, blogging application, and student registration [73], which have been written in multiple programming languages, including Java and Ruby on Rails. *Konure* inferred a subset of their commands, which contain SQL queries, conditionals, loops, and outputs.

User Manual Input: To use *Konure* the user provides the necessary command and data specifications. The command specification identifies the API command to be inferred, and the data specification provides information on the database schema. *Konure* uses these to automate its exploration of the application's behavior and infer a model for it.

6 Automatically Replacing In-Memory Data Structures with Databases

As discussed in Section 5, developing a module that interacts with a database to access and persist data has many advantages, but also implementation challenges.

In this example, we show how program inference can infer a database replacement for existing in-memory data structures in a program. This approach allows developers to write a program in a plain imperative language, such as Python, using familiar, standard programming constructs and data structures (*i.e.*, no need to develop a database model or use database-related frameworks). The approach then infers the program and regenerates a new program that accesses data using a database (instead of the original data structures).

A key distinction here is that while our previous example demonstrated how *Konure* infers the use of an *existing* database in an imperative program, this example (and the corresponding system) focuses on a program that does not yet use a database but rather performs computations that *can* be modeled and replaced with a database.

We introduce *Nero*, an active learning system we developed to infer the use of standard in-memory data structures (*e.g.*, lists and dictionaries) in a program, model their use with database queries, and replace the in-memory data structures with a database in a regenerated program [84].

This approach presents multiple advantages: the developer need not be familiar with databases, and similarly to the case seen in Section 5, the module's development does not become tied to a particular database system, as the regenerated code can be adapted to other database systems.

6.1 A New User Post Querying Service

A developer has been tasked with writing a new variant of the module associated with users' posting history (Section 5). Figure 12 presents the module code written by the developer for this new version.

The module uses a list, `users`, to track each user's first/last names, and uses the corresponding index position as an id. Post information is stored in the `post_contents` dictionary, which maps a post title to the corresponding post contents (a string). The application uses another list, `posts`, to track tuples of user ID and post titles, representing users' posts.

Nero refers to the code written by the developer as a *seed program*, which satisfies certain program properties that enable efficient inference. In this case, the code has two functions: `do_post` and `do_listuserposts`. Function `do_post` takes a user ID, a string representing the post contents, and a post title. The user ID is used to query the `users` list, retrieving and printing the user's name. The function then registers the user's post by appending the user ID and the post title to the `posts` list. Finally, the function inserts the corresponding post contents (`post_str`) into a dictionary (`post_contents`) using the post title as a key. Function `do_listuserposts` takes as input a user ID, queries for the user's name and prints it, and then iterates over the `posts` list to identify the user's posts and prints out the corresponding post title and contents.

```
# List of user information.
# Index: user id
# Entries: (first_name, last_name)
users = []

# Dictionary of post information.
# Key: post title
# Entries: post contents
post_contents = {}

# List of user post information.
# Entries: (user_id, post_title) indicating a post made by a user
posts = []

def do_post(user_id_query, post_str, post_title):
    first_name, last_name = users[user_id_query]
    print(first_name, last_name)
    posts.append((user_id_query, post_title))
    post_contents[post_title] = post_str

def do_listuserposts(user_id_query):
    first_name, last_name = users[user_id_query]
    print("User:␣{},␣{}".format(first_name, last_name))
    for user_id, post_title in posts:
        if user_id_query == user_id:
            post_str = post_contents[post_title]
            print(user_id_query, post_title, post_str)
```

Fig. 12. Inferring programs with lists/dicts. *Nero* infers the data structures and their accesses used by a *seed* program and models their behavior using database queries.

6.2 Approach

Nero treats a program's control logic and data as two separate components. This conceptual separation often occurs in data-intensive programs. *Nero* uses data accesses to infer the program's control logic. To do so, *Nero* carefully chooses the input values, data structure contents, and executes the program multiple times.

Figure 13 provides an overview of the *Nero* system. As in the *BDP* (Figure 7) and *Konure* (Figure 10) pipelines, the grey box with bolded dotted edges corresponds to the original application, while the light blue boxes with dashed edges correspond to the inference components.

Our technique starts with a seed program. *Nero* restricts the space of programs that it can infer through a domain-specific language (DSL). In this case, that DSL can be expressed as a subset of Python. Additionally, *Nero* requires a command specification, which outlines the commands we want to infer, and a data specification, that details the types for the global data structures used by the module and which our inferred model will replace with a database.

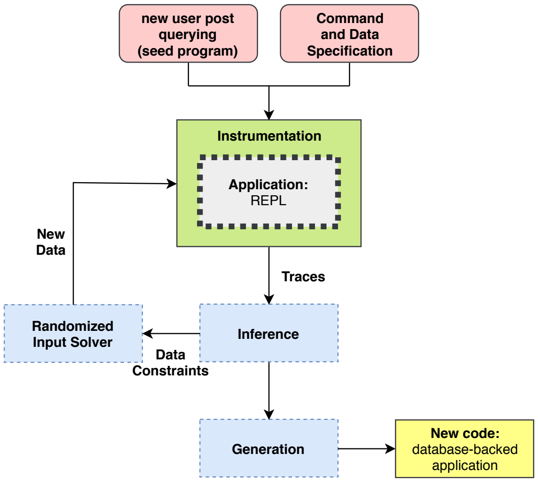
new user post querying (seed program)
Command and Data Specification
Instrumentation
Application: REPL
New Data
Traces
Randomized Input Solver
Inference
Data Constraints
Generation
New code: database-backed application

Fig. 13. *Nero* system diagram. *Nero* can be used to infer the behavior of programs that access data using lists/dicts and regenerate them to use a database instead. With respect to the high-level paradigm (Figure 1): the *Inference* box instantiates the *Core Inference Algorithm*, and the *Data Constraints* arrow and *Randomized Input Solver* box instantiate the *Input Generator/Acquirer*.

In the posting module, a (partial) command specification would correspond to

```
[
  ["post", "do_post", "int", "str", "str"],
]
```

This specification states that the command post is implemented by the do_post function in the seed program, and it takes three arguments: an integer and two strings.

A (partial) data specification would be

```
{
  "users": [
    [["str", "str"]],
    ["id", "first_name", "last_name"],
    "Users"
    ]
}
```

This specification states that the users data structure contains entries corresponding to two strings (first_name and last_name). The corresponding database schema is a table named Users, with three columns: a virtual primary key (id) and the first and last name columns (corresponding to the two strings in the original data structure’s entries).

Execution: *Nero* takes the seed program, along with the two specification files, and generates boilerplate code to produce a read-eval-print-loop (REPL) that can be used to interact with the application. This REPL is used to execute commands with new inputs and explore the application’s behavior. This has the benefit of reducing the code that the seed program developer has to write, as well as standardizing the interface that *Nero* uses for inference.

Dynamic Analysis: *Nero* relies on dynamic instrumentation to produce execution traces. These traces contain information on the data structures accessed and assigned to. To collect these traces *Nero* uses the REPL wrapper described previously, and records any accesses (and parameters used) to lists or dicts during execution (as well as certain key functions, like print) by using instrumented variants of lists/dicts.

For example, the expression users[user_id_query] in line 16 of Figure 12 would produce the following entry in the execution trace (assuming that user_id_query has a value of 2, the users data structure has 1 as an instrumentation identifier (UID), and the corresponding entry in the list contains ("john", "doe")):

```
list UID1 __getitem__ [2] [("john", "doe")]
```

Inferring Program Constructs: *Nero* collects the traces of data accesses and infers the program’s data and control flow. To do this *Nero* identifies the origin of different values observed in the trace, the presence of loops, and the constraints for any conditional loops. *Nero* repeatedly executes the program with new inputs to reduce ambiguity and explore new paths.

For example, *Nero* can infer data structure accesses, such as users[user_id_query], directly from the trace. If *Nero* initializes data structures and chooses input values carefully, it will also be able to infer that the argument to this lookup (user_id_query) corresponds to an input parameter to the post command (implemented by do_post). However, the origin of values can be ambiguous if multiple possible locations could have produced that value. This highlights the importance of repeated execution and inference for *Nero*’s success.

While the example presented here includes user identifiers and post identifiers, *Nero* does not rely on knowing that these are unique identifiers (and map to a database primary key). The inference algorithm, however, does rely on accessing the input parameters to the application and successfully identifying that the input parameter is used as a index/key to access a list/dict. We believe the number of programs that take an input parameter value and then use it to access a program data structure is significant.

Loops over data structures in the program are identified through execution trace events creating iterators. *Conditional* loops, on the other hand, require that *Nero* infer a predicate that induces execution of the body. To do so, *Nero* uses a randomized approach to inferring data constraints to generate input/data structure values. The algorithm is inspired by simulated annealing [40]. It fine tunes the temperature parameter to balance the inherent tension in value generation: *Nero* needs to induce repeated values across structures to lead to the conditional loop’s execution, but inference also requires unique values to resolve ambiguity in the program’s logic.

Randomized Input Solver: While values are generated randomly by *Nero*, they are subject to constraints that will 1) avoid application crashes, 2) reduce inference ambiguity, and 3) explore program paths (by inducing conditional loop execution). These constraints are discovered progressively during inference and are used in a graph-based algorithm to populate possible values.

Nero's randomized value generation demonstrates the flexibility of such an approach when the domain of values grows and the application complexity increases, compared to a deterministic, constraint-based solving approach.

Code Generation: *Nero*'s inferred model consists of database queries that can effectively capture the original program behavior. *Nero* can generate a skeleton program, consisting of database boilerplate such as schema definitions and database connection/cursor management, and integrate the queries into this skeleton to produce an executable program. The current implementation of *Nero* uses Python and its accompanying SQLite module [57].

Evaluation: We evaluated *Nero* by using it to infer four Python seed programs, including a task management program, a chat room program, and a blog program. These programs are adapted from database-backed applications [1–3], by rewriting a subset of their functionality in Python using standard in-memory data structures. For all these programs, *Nero* fully inferred the target functionality and produced regenerated programs that use a database.

Comparison to Object Relational Mapping: Object Relational Mapping (ORM) frameworks [4, 5] facilitate access to a database by providing objects which an imperative program can use to query/modify a database. While both ORMs and *Nero* share the goal of integrating database usage into applications, they take orthogonal approaches. Namely, an application that uses an ORM must *already* be implemented to use the corresponding API. In contrast, *Nero*'s core focus is the inference task: it infers the data usage information and its relational mapping directly from executions of the seed program, which can be written with standard data structures. Additionally, the model that *Nero* infers, while represented in terms of SQL queries, does not need to be limited to producing database executions, as highlighted in Section 7.3.

User Manual Input: To use *Nero* the user must provide a seed program, which has semantics corresponding to the desired database-backed application but is written in a subset of plain Python (*i.e.*, the *Nero*'s DSL) using standard in-memory data structures. The user also provides a data and command specification. The user may also specify the desired names to use for the database tables and columns in the regenerated program. The command specification describes the commands of the application, specifically, the names of the entry functions in the seed program, as well as the datatypes for their arguments.

7 Active Learning for Software Engineering as a General Paradigm

We discussed three systems, *BDP*, *Nero*, and *Konure* [73], which employ active learning to infer a target program. These three systems leverage domain knowledge to enable the design of effective inference algorithms. This approach is consistent with the history of development trends in programming languages and software engineering in general—compiler optimizations, for example, were first designed for specific code patterns and later developed into sophisticated systems effective for a significantly wider range of programs [26, 39, 50].

We anticipate that future research in active learning of programs will exploit modularity to extend to expressive domains, where target applications consist of multiple (inferrable) modules. Potential domains include software for data retrieval, image processing, embedded systems, distributed systems, network protocols, user interfaces, and certain numerical computations. Potential use cases include software migration, software understanding, security analysis, robust code generation, *etc.* There is a large design space for this line of research, as demonstrated by the different systems discussed.

Broadly speaking, we describe active learning for software engineering in terms of three components: an observation model, an inference model, and a generation model. We next place the three systems in the context of each of these paradigm components, point towards different research directions, and highlight open questions.

7.1 Observational Model

The observation model roughly outlines what aspects of the program's behavior the technique observes and how it does so. All three systems described use a greybox technique. *BDP* and *Nero* use program instrumentation to obtain execution traces. While the former is focused on tracking concrete executions and building byte-level expressions over the inputs, the latter only records instrumented method calls along with information on the concrete input and return values. *Konure*, yet another greybox approach, intercepts SQL queries as well as the observed input and outputs.

Other models of observation present different trade-offs. For example, blackbox interaction may limit the complexity of the target application that can be inferred, or it may lead to uncertain inference, but it also removes assumptions such as implementation language or direct access to the application's execution. For example, *Hecate* [71] uses a blackbox approach to interact and infer programs that store and retrieve data, as long as the applications conform to a model for programs using a relational database. Compared to this black box approach, the greybox approach where SQL queries are visible (*Konure*) infers more sophisticated database programs because of the richer information available.

A whitebox observation model is yet another alternative, which would allow incorporating additional program analysis, or might facilitate rewrites of the program that reduce (or remove) inference ambiguity (*e.g.*, using SSA).

Exciting research directions that target the observational model include identifying domains and applications that can be successfully inferred with limited interaction (*e.g.*, not just black box, but also limited numbers of queries). Similarly, we have assumed that the application’s behavior is observed truthfully and there is no adversarial interaction. However, one could imagine scenarios where we are trying to infer an application that is actively adding noise to the outputs or non-deterministically picking execution paths.

Producing New Inputs: A key advantage of active learning is that the technique can use new inputs to improve inference. How these inputs are produced may vary depending on the technique.

For example, *Nero* produces values using a constrained random value generator. *BDP*, in contrast, relies on having access to a large number of example inputs and filtering down to those that can not be parsed with the currently hypothesized program. *Konure* uses an SMT solver, along with constraints implied by program traces, to produce new values.

Depending on the application, generating new inputs can pose challenges. For example, a highly structured custom binary input would require a sophisticated generation process, in contrast with a common format such as bitmap images. Integrating input generation techniques developed in areas such as fuzzing [31, 34] into an active approach could broaden the range of applications that can be inferred.

7.2 Inference Model

The inference model roughly aims to answer the question: *“How do we represent the observations made in a way that we can identify and generalize common application patterns?”*. A key component to successfully identify these shared patterns is the use of an intermediate representation amenable to inference. A good representation should capture the behavior of interest and constrain the space of possible hypotheses derivable from an observation.

We have found carefully designed DSLs to be a particularly effective intermediate representation. For example, *BDP* uses a simple language that describes data structures as sequences of bytes (or words), allowing it to identify repetitions and generalizations. *Nero* constrains the seed program to a DSL so that it can identify key control flow. Similarly, *Konure*’s DSL represents hypothesized (partially inferred) programs and guides inference through hierarchical expansion of non-terminals (*i.e.*, unexplored subprograms).

DSLs have already been found useful in constraining the search space in other automated programming areas [7]. They are modular by definition [51], and thus are well matched to the modular approach we believe is critical to scale active learning-based program inference to large applications. The three systems presented all incorporate knowledge about the target domain to make inference more efficient; incorporating this information through a DSL is simple—for example, extend the grammar with a corresponding production or modify the semantics of an operator. Finally, a DSL facilitates the use of existing analysis techniques (such as structural induction proofs) to reason about the correctness of the inference, as done in *Konure* [73].

Different inference models may reason differently about the observed executions and their generalization. For example, *BDP* partitions the space of inputs (based on executions observed and their input files) and iteratively generalizes a DDP based on each partition’s members, which when combined with flag expressions, capture the application behavior. *Konure*, in contrast, maintains a program representation guaranteed to cover all observed executions, and modifies that hypothesis based on satisfiability criteria for newly observed traces.

Different approaches to inference may also yield varied properties. For example, *Konure*’s inference algorithm guarantees termination and that the paths in the application are fully explored if no non-terminals remain in the hypothesized program.

Identifying appropriate inference procedures for different intermediate representations, domains, and applications, is an exciting direction for future research. Other parts of the paradigm also play a role: for example, can inference be made more efficient under a whitebox observational model? Can we formulate a generalized inference procedure based on an input DSL and application? Or is efficient inference restricted to custom procedures for each group of applications we want to infer? Can we develop statistical techniques that can reason effectively for programs that violate many standard statistical assumptions (as might be the case for a stateful, long-running program)?

7.3 Generation Model

There are many possible uses for the inferred model of an application’s behavior. There is no reason, apriori, that these models need to be used for code generation. We term this portion of the paradigm the *generation model*, as we can generate different outputs based on the representation we have learned. For example, an inferred model could be used to generate documentation for a program (or be used as documentation itself). How best to integrate these inferred models, beyond code generation, into existing developer workflows remains an open question.

Assuming the goal is to generate code, a question about the possible impact on software maintenance arises. We believe that code generated from an inferred model has the potential to be simpler (for example, see the code generation discussion in Section 4), as restricted by the chosen generation model. Program inference, in this regard, holds particular promise in rejuvenating legacy code. We also believe inferred models can help identify application functionality that can be modularized, producing self-contained and re-usable components for software development. Jointly, these properties would contribute to reduce the burden of software maintenance through simpler, modular, and extensible (generated) programs.

8 Challenges and Threats to Validity

Sucessful inference must avoid inferring incorrect properties. A carefully designed representation, such as a DSL, can be used to invalidate particularly undesirable outcomes. For example, both *Nero* and *Konure* have DSLs that only allow bounded loops. Furthermore, the representation, when co-designed with the inference algorithm, can lead to desirable inference guarantees. *Konure*, for example, guarantees that its inference terminates and when it does so no unexplored paths remain.

We anticipate that the main difficulty in deploying our active learning-based approach to program inference will be in appropriately matching the target applications and the restrictions imposed by our techniques. In particular, developers using these systems in practice will need to understand the limitations of each technique. As such, clear documentation of these inference systems is a key design goal.

Active learning requires a supply of new inputs. Depending on the application, generating these examples may be a challenging problem in its own right (*e.g.*, highly structured inputs). However, we have already successfully inferred programs in two domains (data retrieval/storage and binary parsing) and believe that many productive applications can be handled with relatively simple inputs. Techniques from other research areas, such as fuzzing, can also be integrated to help.

To infer large-scale applications, we propose to exploit the modularity of modern software to enable effective (and scalable) inference. This direction leverages the trend of increasingly popular third-party library usage [6, 16, 52, 79, 81, 89]. Applications with less modular designs will likely present a cost-of-inference challenge and the need for refining the observation and inference models used. For example, *BDP* would incur a larger overhead when inferring a monolithic application, where parser data structures might be populated throughout the application (requiring broader instrumentation). However, existing work on program partitioning [21, 22, 56, 77] could be adapted to induce some degree of modularity and reduce the cost of inference. Other practices, such as the use of global variables, may also influence the inference strategy. For example, an existing prototype [65] successfully developed an inference strategy that handled reads and writes of a global data structure. The key to this success is identifying the effective boundary of the target behavior we are interested in inferring, interposing the appropriate observation model, and tailoring the inference model as needed.

9 Related Work

Program Synthesis: Program synthesis is an active area of research [7, 28, 36, 38, 60], particularly programming-by-example [64, 75, 85], where the user provides input-output examples as a form of specification. These techniques often define a DSL that captures the space of programs. Synthesizing a correct program (out of potentially many) then becomes a search over the DSL for a program that is consistent with the examples provided.

Oracle-guided synthesis [38, 43, 76] extends the search to include an oracle, often an SMT solver, which can validate proposed programs and provides counter-examples when the proposed program is incorrect.

The systems discussed here perform inference by (partially) observing the behavior of a full application. Rather than rely on a fixed corpus of input-output examples, the application's execution is used to define the target specification. Incorporating active learning allows these systems to successfully and efficiently expand the input-output examples automatically as needed.

Reverse Engineering and Transformations: Reverse engineering has been successfully used to understand, maintain and rewrite otherwise unfamiliar applications (*e.g.*, legacy code) [20, 49, 78]. However, many reverse engineering tools require significant human effort and expertise [44], and assume the tool has access to extensive static information, such as source code or the compiled binary, or dynamic information, such as whole-program instruction-level instrumentation [19, 27, 66]. Work automating the reverse engineering process [23, 48, 55] has focused on narrow inference tasks like communication protocols rather than whole programs.

Similarly, source code transformations, by definition, rely on significant access to the original application, which for example rules out transformation of remote applications. Recent work [41] has explored enabling source code transformations across multiple languages, but the scope of these transformations remains restricted.

Our vision for active learning for software engineering addresses the hurdles highlighted. For example, all three systems presented make limited use of human intervention and automate a significant portion of the inference process. The observational models presented enable inference on applications where the source code may not be available. Finally, representing the inferred model in a DSL allows us to employ transformations that can be re-used across possible target languages, as the program translation step is handled by an independent generation model.

Language Inference: Inference of regular and context free languages has been a well studied problem [32, 58, 87]. These techniques have been applied to, for example, the grammar for a program's inputs [13], rules of a board game [42], the partitioning of objects in an image [30], among others. The examples we presented here highlight the potential for inferring full program functionality.

Model-based Software Engineering: The use of models in software engineering has enabled diverse applications such as program verification [15], testing [18], generation [47], and reuse [35]. The user may be tasked with providing a model specification written in a higher-level language, or a simplified model may be extracted automatically (if the target software already exists for analysis or execution).

The models inferred by *BDP*, *Nero* and *Konure* reflect full functionality of particular application commands, rather than a simplified model (except where a pruned model is intentionally desired). All three systems use a DSL to constrain the search space for their inference procedures, but do not expose these directly to the user.

Active Learning: Active learning has been extensively explored in the machine learning community [10, 82, 86] and successfully applied to a variety of tasks, including complex image classification [88], data cleaning and labeling [33], symbolic systems [9], among others.

Previous work in the software engineering community has also explored applications of active learning. For example, active learning has been used to: derive program assertions from test suite executions [59], infer likely points-to specifications using black-box observation [14], improve program behavior classification [17], and synthesize Datalog programs [74].

The three systems we discussed here use active learning for the broader task of inferring full functionality of key commands in applications.

10 Conclusion

Pervasive modularity in modern software has created an opportunity for deploying active learning at scale, enabling a state-of-the-art approach to security, code generation, adaptation, documentation, and more. We walked through how three existing inference techniques (along with prototypes) infer key, and varied, modules in a representative social networking application. We outlined the general paradigm, contextualized the systems discussed, and identified open questions and challenges.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions. This research was funded in part by National Science Foundation grant CNS-1513687, and DARPA grant FA8650-15-C-7564.

References

[1] 2018. Getting Started with Rails. http://guides.rubyonrails.org/getting_started.html.

[2] 2018. Kanban. https://github.com/somlor/kanban.

[3] 2018. Kandan – modern open source chat. https://github.com/kandanapp/kandan.

[4] 2019. SQLAlchemy - The Database Toolkit for Python. https://www.sqlalchemy.org/.

[5] 2019. The Web framework for perfectionists with deadlines | Django. https://www.djangoproject.com/.

[6] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 385–395.

[7] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[8] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

[9] Garrett Andersen and George Konidaris. 2017. Active exploration for learning symbolic representations. In *Advances in Neural Information Processing Systems*. 5009–5019.

[10] Philip Bachman, Alessandro Sordoni, and Adam Trischler. 2017. Learning Algorithms for Active Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 301–310. http://proceedings.mlr.press/v70/bachman17a.html

[11] Godmar Back. 2002. DataScript-A specification and scripting language for binary data. In *International Conference on Generative Programming and Component Engineering*. Springer, 66–77.

[12] Julian Bangert and Nickolai Zeldovich. 2014. Nail: A practical tool for parsing and generating data formats. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 615–628.

[13] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

[14] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 678–692. https://doi.org/10.1145/3192366.3192383

[15] Ivan Bocic and Tevfik Bultan. 2017. Symbolic model extraction for web application verification. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 724–734.

[16] Jan Bosch. 2015. Speed, data, and ecosystems: the future of software engineering. *IEEE Software* 33, 1 (2015), 82–88.

[17] James F. Bowring, James M. Rehg, and Mary Jean Harrold. 2004. Active Learning for Automatic Classification of Software Behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 195–205. https://doi.org/10.1145/1007512.1007539

[18] Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. 2016. Testing the Untestable: Model Testing of Complex Software-intensive Systems. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 789–792. https://doi.org/10.1145/2889160.2889212

[19] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 317–329.

[20] Vitaly Chipounov and George Candea. 2010. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 5th European conference on Computer systems*. ACM, 167–180.

[21] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/1294261.1294265

[22] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 301–314. https://doi.org/10.1145/1966445.1966473

[23] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H Katz. 2006. Protocol-Independent Adaptive Replay of Application Dialog.. In *NDSS*. Citeseer.

[24] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *International Conference on Software Engineering and Formal Methods*. Springer, 233–247.

[25] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. 2009. VCC: Contract-based modular verification of concurrent C. In *2009 31st International Conference on Software Engineering-Companion Volume*. IEEE, 429–430.

[26] Jeffrey A Dean and Craig Chambers. 1996. *Whole-program optimization of object-oriented languages*. Citeseer.

[27] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, 4.

[28] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 229–239.

[29] Kathleen Fisher and Robert Gruber. 2005. PADS: a domain-specific language for processing ad hoc data. In *ACM Sigplan Notices*, Vol. 40. ACM, 295–304.

[30] Abram L Friesen and Pedro M Domingos. 2018. Submodular Field Grammars: Representation, Inference, and Application to Image Parsing. In *Advances in Neural Information Processing Systems*. 4307–4317.

[31] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 474–484.

[32] Pedro García, Enrique Vidal, and Francisco Casacuberta. 1987. Local Languages, the Succesor Method, and a Step Towards a General Methodology for the Inference of Regular Grammars. *IEEE Trans. Pattern Anal. Mach. Intell.* 9, 6 (1987), 841–845. https://doi.org/10.1109/TPAMI.1987.4767991

[33] R. A. Gilyazev and D. Yu. Turdakov. 2018. Active Learning and Crowdsourcing: A Survey of Optimization Methods for Data Labeling. *Programming and Computer Software* 44, 6 (2018), 476–491. https://doi.org/10.1134/S0361768818060142

[34] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. *SIGPLAN Not.* 43, 6 (June 2008), 206–215. https://doi.org/10.1145/1379022.1375607

[35] Thiago Gottardi, Rafael Serapilha Durelli, Oscar Pastor López, and Valter Vieira de Camargo. 2013. Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort. *J. Software Eng. R&D* 1 (2013), 4. https://doi.org/10.1186/2195-1721-1-4

[36] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, Vol. 46. ACM, 317–330.

[37] Jingrui He, Hanghang Tong, Mingjing Li, Hong-Jiang Zhang, and Changshui Zhang. 2004. Mean version space: a new active learning method for content-based image retrieval. In *Proceedings of the 6th ACM SIGMM international workshop on Multimedia information retrieval*. ACM, 15–22.

[38] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 215–224.

[39] Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 194–206.

[40] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[41] James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One tool, many languages: language-parametric transformation with incremental parametric syntax. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 122.

[42] Jakub Kowalski and Andrzej Kisielewicz. 2018. Regular Language Inference for Learning Rules of Simplified Boardgames. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 1–8.

[43] Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. *arXiv preprint arXiv:1703.03539* (2017).

[44] A Lee, A Payne, and T Atkison. 2018. A Review of Popular Reverse Engineering Tools from a Novice Perspective. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer âĂę, 68–74.

[45] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

[46] Mingkun Li and Ishwar K Sethi. 2006. Confidence-based active learning. *IEEE transactions on pattern analysis and machine intelligence* 28, 8 (2006), 1251–1261.

[47] Richard Lei Li, John G. Hosking, and John C. Grundy. 2008. MaramaEML: An Integrated Multi-View Business Process Modelling Environment with Tree-Overlays, Zoomable Interfaces and Code Generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. 477–478. https://doi.org/10.1109/ASE.2008.79

[48] Xiangdong Li and Li Chen. 2011. A survey on methods of automatic protocol reverse engineering. In *2011 Seventh International Conference on Computational Intelligence and Security*. IEEE, 685–689.

[49] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University, 5.

[50] Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 603–616.

[51] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[52] Tommi Mikkonen and Antero Taivalsaari. 2019. Software Reuse in the Era of Opportunistic Design. *IEEE Software* 36, 3 (2019), 105–111.

[53] Falcon Momot, Sergey Bratus, Sven M Hallberg, and Meredith L Patterson. 2016. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 45–52.

[54] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-Francois Petin, and Eric Levrat. 2010. Automatic code generation for embedded systems: From UML specifications to VHDL code. In *2010 8th IEEE International Conference on Industrial Informatics*. IEEE, 1085–1090.

[55] John Narayan, Sandeep K Shukla, and T Charles Clancy. 2016. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 40.

[56] Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden. 2009. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 395–408. http://dl.acm.org/citation.cfm?id=1558977.1559004

[57] Mike Owens. 2006. *The definitive guide to SQLite*. Apress.

[58] Tsyh-Wen Pao and John W. Carr III. 1978. A Solution of the Syntactical Induction-Inference Problem for Regular Languages. *Comput. Lang.* 3, 1 (1978), 53–64. https://doi.org/10.1016/0096-0551(78)90006-1

[59] Long H. Pham, Ly Ly Tran Thi, and Jun Sun. 2017. Assertion Generation Through Active Learning. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 155–157. https://doi.org/10.1109/ICSE-C.2017.87

[60] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices* 51, 6 (2016), 522–538.

[61] Erik Poll. 2018. LangSec revisited: input security flaws of the second kind. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 329–334.

[62] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 107–126.

[63] Evan Priestley. 2011. How many lines of code is Facebook? https://qr.ae/TWpXts Accessed: 2019-04-18.

[64] Mohammad Raza and Sumit Gulwani. 2018. Disjunctive Program Synthesis: A Robust Approach to Programming by Example. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[65] Martin C. Rinard, Jiasi Shen, and Varun Mangalick. 2018. Active Learning for Inference and Regeneration of Computer Programs That Store and Retrieve Data. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018)*. ACM, New York, NY, USA, 12–28.

[66] Chris Sanders. 2017. *Practical packet analysis: Using Wireshark to solve real-world network problems*. No Starch Press.

[67] Greg Schechter. 2011. Visualizing Facebook's PHP Codebase. https://bit.ly/2DkcL2R Accessed: 2019-04-18.

[68] Isaac Z. Schlueter et al. 2010. Node Package Manager. https://npmjs.com Accessed: 2017-02-17.

[69] Burr Settles. 2009. *Active learning literature survey*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[70] Lwin Khin Shar and Hee Beng Kuan Tan. 2012. Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 1293–1296. https://doi.org/10.1109/ICSE.2012.6227096

[71] Jiasi Shen and Martin Rinard. 2017. *Inference and Regeneration of Programs that Manipulate Relational Databases*. Technical Report. http://hdl.handle.net/1721.1/111067

[72] Jiasi Shen and Martin Rinard. 2017. Robust programs with filtered iterators. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 244–255.

[73] Jiasi Shen and Martin C. Rinard. 2019. Using Active Learning to Synthesize Models of Applications That Access Databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 269–285.

[74] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 515–527.

[75] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.

[76] Armando Solar-Lezama and Rastislav Bodik. 2008. *Program synthesis by sketching*. Citeseer.

[77] Eli Tilevich and Yannis Smaragdakis. 2009. J-Orchestra: Enhancing Java Programs with Distribution Capabilities. *ACM Trans. Softw. Eng. Methodol.* 19, 1, Article 1 (Aug. 2009), 40 pages. https://doi.org/10.1145/1555392.1555394

[78] Paolo Tonella. 2005. Reverse engineering of object oriented code. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. IEEE, 724–725.

[79] Nikos Vasilakis, Ben Karel, Yash Palkhiwala, John Sonchack, André DeHon, and Jonathan M. Smith. 2019. Ignis: Scaling Distribution-Oblivious Systems with Light-Touch Distribution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1010–1026. https://doi.org/10.1145/3314221.3314586

[80] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2017. Towards Fine-grained, Automated Application Compartmentalization. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS'17)*. ACM, New York, NY, USA, 43–50. https://doi.org/10.1145/3144555.3144563

[81] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_08-3_Vasilakis_paper.pdf

[82] Kai Wei, Rishabh Iyer, and Jeff Bilmes. 2015. Submodularity in Data Subset Selection and Active Learning. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, France, 1954–1963. http://proceedings.mlr.press/v37/wei15.html

[83] Michael Widenius and Davis Axmark. 2002. *Mysql Reference Manual* (1st ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[84] Jerry Wu. 2018. Using Dynamic Analysis to Infer Python Programs and Convert Them into Database Programs. In *Master's thesis*. Massachusetts Institute of Technology.

[85] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated migration of hierarchical data to relational tables using programming-by-example. *Proceedings of the VLDB Endowment* 11, 5 (2018), 580–593.

[86] Songbai Yan, Kamalika Chaudhuri, and Tara Javidi. 2018. Active Learning with Logged Data. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, StockholmsmÃďssan, Stockholm Sweden, 5521–5530. http://proceedings.mlr.press/v80/yan18a.html

[87] Takashi Yokomori. 1987. Inductive Inference of Context-free Languages - Context-free Expression Method. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy, August 23-28, 1987*. 283–286. http://ijcai.org/Proceedings/87-1/Papers/058.pdf

[88] Bangyan Zhu, Xiao Wang, Zhengwei Chu, Yi Yang, and Juan Shi. 2019. Active Learning for Recognition of Shipwreck Target in Side-Scan Sonar Image. *Remote Sensing* 11, 3 (2019), 243. https://doi.org/10.3390/rs11030243

[89] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. *arXiv preprint arXiv:1902.09217* (2019).