A Sound Static Analysis Approach to I/O API Migration

SHANGYU LI, Hong Kong University of Science and Technology, China
ZHAOYANG ZHANG, Hong Kong University of Science and Technology, China
SIZHE ZHONG, Hong Kong University of Science and Technology, China
DIYU ZHOU, Peking University, China
JIASI SHEN, Hong Kong University of Science and Technology, China

The advances in modern storage technologies necessitate the development of new input/output (I/O) APIs to maximize their performance benefits. However, migrating existing software to use different APIs poses significant challenges due to mismatches in computational models and complex code structures surrounding stateful, non-contiguous multi-API call sites. We present Sprout, a new system for automatically migrating programs across I/O APIs that guarantees behavioral equivalence. Sprout uses flow-sensitive pointer analysis to identify semantic variables, which enables the typestate analysis for matching API semantics and the synthesis of migrated programs. Experimental results with real-world C programs highlight the efficiency and effectiveness of our approach. We also show that Sprout can be adapted to other domains, such as databases.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Software evolution**; *File systems management*.

Additional Key Words and Phrases: Static Analysis, Program Synthesis, Program Migration.

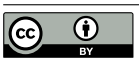
ACM Reference Format:
Shangyu Li, Zhaoyang Zhang, Sizhe Zhong, Diyu Zhou, and Jiasi Shen. 2025. A Sound Static Analysis Approach to I/O API Migration. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 293 (October 2025), 31 pages. https://doi.org/10.1145/3763071

1 Introduction

Computer software is ubiquitous and represents a vast amount of human effort, knowledge, and creativity. This software often relies on application programming interfaces (APIs) that are closely tied to specific platforms. As new platforms emerge, the software that depends on existing APIs can become obsolete. However, migrating existing software to modern APIs has been complex and challenging. This situation forces a choice: either cling to outdated systems that are unable to leverage the full benefits of modern technologies, or discard valuable existing software to develop new programs from scratch, which is costly and inefficient. Despite decades of research and significant effort invested in developing automated migration techniques [4, 85, 89, 90], success has been limited, causing substantial costs to society [25, 31, 61–64, 72].

One prominent example is associated with the evolution of storage technologies. Modern storage technologies offer substantial benefits in speed and efficiency. To maximize these benefits, it is crucial to design abstractions that leverage insights about the optimal ways to use the hardware. For example, traditional POSIX interfaces [30] are widely recognized as suboptimal for the latest

Authors' Contact Information: Shangyu Li, Hong Kong University of Science and Technology, Hong Kong, China, sliew@connect.ust.hk; Zhaoyang Zhang, Hong Kong University of Science and Technology, Hong Kong, China, zhaoyang.zhang@connect.ust.hk; Sizhe Zhong, Hong Kong University of Science and Technology, Hong Kong, China, sizhe.zhong@connect.ust.hk; Diyu Zhou, Peking University, Beijing, China, diyu.zhou@pku.edu.cn; Jiasi Shen, Hong Kong University of Science and Technology, Hong Kong, China, sjs@cse.ust.hk.



ACM 2475-1421/2025/10-ART293
https://doi.org/10.1145/3763071

hardware, and there is a consensus that existing system calls are inadequate for modern computing environments [28, 43, 68, 88, 96]. This issue arises mainly because POSIX interfaces closely resemble those of the original Unix [73], which rely on foundational assumptions that have become obsolete due to advances in computer speed and capacity, such as the assumption that memory is significantly faster than input/output (I/O) devices, which led to I/O system calls that involve copying data to or from memory. However, with advances in high-speed I/O devices such as 100 Gb/s network cards [50, 70] and non-volatile memory (NVM) [3, 32], this assumption no longer holds, and additional memory copying introduces considerable performance overhead. As a result, much research advocates for the redesign of storage APIs to maximize potential hardware benefits [9, 10, 26, 28, 38, 43, 51, 52, 68, 74, 88, 96]. Adopting the redesigned APIs involves migrating existing user applications, but unfortunately, this task has been difficult, tedious, and expensive.

1.1 Challenges

Migrating programs that rely on I/O APIs is crucial but challenging due to different computational models in various storage platforms. For example, SubZero [43] offers new file system APIs for NVM to replace POSIX APIs [30], and experiments indicate significant performance gains [43]. Migrating a program to SubZero involves transforming the POSIX `(p)read` and `(p)write` calls into SubZero's `peek` and `patch` API. This task requires understanding the underlying computation and the semantic differences between the APIs. Several notable challenges arise:

- **Semantic mismatch between old and new APIs.** To maximize performance benefits of modern storage technologies, it is often necessary to change programming paradigms and adopt different conventions [43, 88, 96]. Consequently, APIs may not align perfectly in semantics.
- **Complex control flow in non-contiguous structures.** Even with precise API mappings, migration often requires changes in multiple locations. The migration task often requires replacing multiple API invocations interwoven in complex code structures such as loops, conditional branches, and layers of implicit and explicit function calls due to abstraction and encapsulation.
- **Entangled value flows surrounding API invocation sites.** To ensure the migrated program is correct, the synthesized code must fit seamlessly into the existing codebase. It should pass the correct incoming variables and expressions to the new APIs and assign the new API return values to the correct outgoing variables, which requires an appropriate approximation of the meanings of existing variables that may involve both intra- and inter-procedural contexts.

Thus, techniques based on string replacements or local transformations [20, 23, 35, 46, 53, 59, 75, 77, 90] are inadequate for ensuring the correctness of migration across diverse APIs and codebases.

1.2 Our Approach

We aim to bridge the gap in migrating software across different I/O APIs by proposing a novel automated technique based on static analysis. Our approach builds on the following key ideas:

- **Decomposing API semantics into atomic operations.** To map APIs with slightly different semantics, we decompose API functionality into atomic operations. This decomposition enables fine-grained control to align APIs through overlapping operations and synthesizes new expressions and statements to handle discrepancies. We focus on ensuring behavioral equivalence, disregarding minor mismatches in corner cases that can be tolerated.
- **Simplifying complex code structures.** To make the migration task tractable, we preprocess the original program by simplifying loops and collapsing layers of aliased functions to reduce the complexity of the code structures often found in real-world codebases.
- **Approximating value flows and semantic types based on pointer analysis.** To reason about the meanings of existing variables, we approximate the value flows by employing a flow-

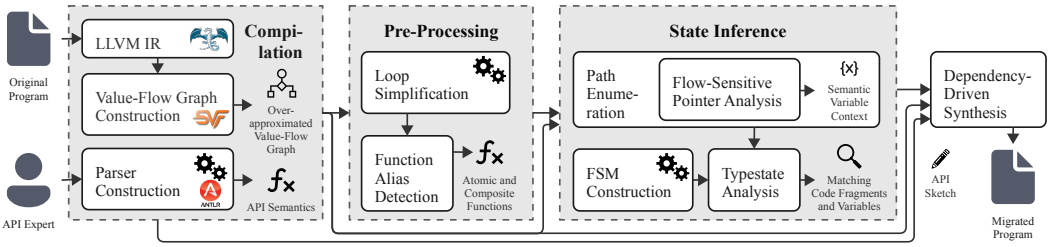


Fig. 1. Overview of the Sprout workflow. Parser construction, loop simplification, and FSM construction are customizable for other application domains (discussed later in Sec. 6.4).

and field-sensitive pointer analysis to identify semantic variables intra-procedurally, along with Andersen's pointer analysis to over-approximate value flows inter-procedurally [82].

We instantiate these ideas in a tool named Sprout. The design of Sprout consists of four stages: compilation, pre-processing, state inference, and synthesis (Fig. 1). The compilation stage compiles the original program source code into LLVM IR [45] to construct the over-approximated value-flow graph [83], then parses the expert-provided specifications into API semantics and sketches. The pre-processing stage performs loop simplification (Sec. 5.1.1) and function alias detection (Sec. 5.1.2). The state inference stage enumerates paths in composite functions (Def. 5.8), using a flow-sensitive pointer analysis to identify semantic variables (Sec. 4.1). It then constructs a finite state machine (FSM) from the parsed API semantics (Sec. 5.2.1) and performs typestate analysis [24, 81] for the enumerated paths (Sec. 5.2.3). Finally, Sprout uses the identified code fragments and variables to synthesize a migrated program (Sec. 5.3) that is equivalent to the original program (Sec. 5.4).

1.3 Contributions

We highlight the following contributions in this paper:

- **Formalization.** We formalize the migration problem and introduce the concept of behavioral equivalence. We also formalize the I/O behavior of APIs and use it to specify the semantics of a range of traditional POSIX APIs and modern I/O APIs.
- **Algorithm.** We present a novel and sound algorithm for migrating programs to use different I/O APIs. The algorithm is based on static analysis and involves reasoning about the I/O behavior of programs, identifying semantic variables, inferring I/O states, and synthesizing new code.
- **Experimental results.** We implement a prototype for Sprout and evaluate it using a range of real-world open-source C programs. The experimental results indicate that our algorithm is both efficient and effective in enabling automated migration.
- **Generalization.** We adapt our Sprout implementation to work with the domain of database access APIs, which highlights the generality of our core approach.

To the best of our knowledge, Sprout represents the first approach capable of automatically migrating stateful, non-contiguous multi-API invocations with formal correctness guarantees.

2 Motivating Example

Consider the problem of migrating programs that originally read files through POSIX APIs to instead read files through the `get` API proposed by Aerie [88] and ArckFS [96].

API Specification. An expert provides the specification and sketch for the `get` API as shown in Fig. 3a. It specifies that `get` initially establishes a connection to the file specified by `arg0` (Line 3), transfers the entire contents of the file into the buffer `arg1` (Line 4), disconnects from the file (Line 5), and finally returns the size transferred (Line 6). The sketch on Line 9 specifies how to

```
struct stat statbuf;
- fd = Uopen(ss, O_RDONLY, 0);
- if (fd < 0) {
-   *log_msgptr = string_sprintf("failed to
  open ACL file \"%s\": %s", ss, strerror(
  errno));
-   return ERROR;
-   }
- int flag = fstat(fd, &statbuf);
if (flag != 0) {
  *log_msgptr = string_sprintf("failed to
    fstat ACL file \"%s\": %s", ss,
    strerror(errno));
  return ERROR;
  }
acl_text=store_get(statbuf.st_size+1,ss);
acl_text_end=acl_text+statbuf.st_size+1;
- size_t bytes = read(fd, acl_text, statbuf
  .st_size);


if (bytes != statbuf.st_size) {
  *log_msgptr = string_sprintf("failed to
    read ACL file \"%s\": %s", ss, strerror
    (errno));
  return ERROR;
  }
acl_text[statbuf.st_size] = 0;
- (void)close(fd);
acl_name=string_sprintf("ACL \"%s\"",ss);
```

(a) Function body extracted from Exim.

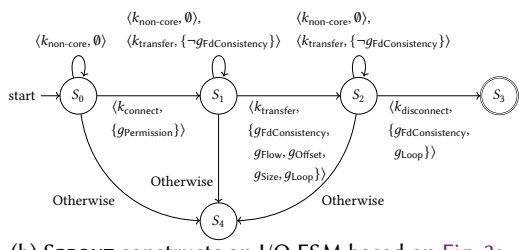
```
struct stat statbuf;




+ int flag = stat(ss, &statbuf);
if (flag != 0) {
  *log_msgptr = string_sprintf("failed to
    fstat ACL file \"%s\": %s", ss,
    strerror(errno));
  return ERROR;
  }
acl_text=store_get(statbuf.st_size+1,ss);
acl_text_end=acl_text+statbuf.st_size+1;
+ size_t bytes = 0;
+ if (!is_tainted(ss)) {
+   bytes = get(ss, acl_text);
+ }
if (bytes != statbuf.st_size) {
  *log_msgptr = string_sprintf("failed to
    read ACL file \"%s\": %s", ss, strerror
    (errno));
  return ERROR;
  }
acl_text[statbuf.st_size] = 0;

acl_name=string_sprintf("ACL \"%s\"",ss);
```

(b) Function body migrated by SPROUT.

Fig. 2. SPROUT migrates a function from Exim [27] to use the get API instead of POSIX APIs.

```
int get(const char *path, void* buf) {
ParamType: (arg0:FilePath,arg1:Buffer);
Semantics: {f=connect_file(arg0,Read),
            n=transfer(f,0,f.size,arg1,0),
            disconnect(f)};
Return: n:SizeTransferred;
}
Sketch: {
Sketch1: size_t #0 = get(#1, #2);
}
```

(a) API specification for the get API proposed by Aerie [88] and ArckFS [96].



(b) SPROUT constructs an I/O FSM based on Fig. 3a.

```
int open(const char* path, int oflag,...) {
ParamType:(arg0:FilePath,
           arg1:FilePermission);
Semantics:{f=connect_file(arg0, arg1)};
Return: f:FileDescriptor;
}
```

(c) API specification for POSIX open.

```
static inline int exim_open(const char*
  pathname, int flags, mode_t mode) {
if (!is_tainted(pathname))
  return open(pathname, flags, mode);
log_write( ... );
errno = EACCES; return -1; }
```

(d) Function in Exim that is aliased with open under the liftable conditional `!is_tainted(arg0)`.

```
int exim_open(const char* pathname, int flags, mode_t mode) {
ParamType:(arg0:FilePath, arg1:FilePermission);
Condition: {!is_tainted(arg0);}
Semantics:{f=connect_file(arg0, arg1)};
Return: f:FileDescriptor;
}
```

(e) SPROUT infers the semantics of exim_open (Fig. 3d).

```
size_t #0;
if (!is_tainted(#3)) {
  #0 = get(#1, #2);
}
```

(f) SPROUT inserts a sketch to prepare for synthesis.

Fig. 3. Example specifications, state machine, aliased function, and sketch. The third parameter of open is omitted for simplicity, as it is optional and specifies the permission mode for file creation [29].

invoke the `get` API within the program. The placeholders `#0`, `#1`, and `#2` represent holes, indicating that `get` requires two input parameters and stores its return value in an integer variable.

Original Program. Fig. 2a presents a code snippet extracted from a function in the Exim email server [27]. This function first opens a file using the `Uopen` macro (Line 2), then reads the file into a buffer by invoking `read` (Line 14), and finally closes the file with `close` (Line 20). This function is not straightforward to migrate by hand. One reason is that there are various conditional branches that perform tasks such as error handling and file status checking, interspersed throughout the function, making the code difficult to understand and modify. Another reason is that the file opening operation is not directly visible in the current function but nested in a macro, which invokes a wrapper function named `exim_open` (see Fig. 3d), which further invokes the `open` API with even more conditional checks. These complications make the file open, read, and close operations separated in the source code and difficult to reason about as a whole.

Simplifying Aliased Functions. Sprout analyzes the original program to characterize the function semantics that indirectly invoke POSIX APIs. In this example, Sprout determines that the function `exim_open(arg0,arg1)` is aliased with the POSIX `open` under the liftable condition `!is_tainted(arg0)` (discussed later in Def. 5.4). Because the POSIX `open` has the semantics as in Fig. 3c, Sprout infers that the semantics of the `exim_open` function is as in Fig. 3e.

Identifying Behaviorally Equivalent Statements. Based on the specification in Fig. 3a, Sprout constructs a finite state machine as shown in Fig. 3b (discussed later in Ex. 5.11), which accepts behaviors equivalent to those emitted by `get` as specified in Fig. 3a. Sprout then enumerates all behavior paths B in the behavior tree of the program, where B goes through the function, to match the FSM in Fig. 3b. In Fig. 2a, the code fragment "`fd = Uopen(ss,O_RDONLY,0)`" emits a behavior node $\beta_1 = \boxed{\alpha_1}$, where $\alpha_1 = v \leftarrow \underline{\texttt{connect\_file}}(v_1, v_2)$ (discussed later in Def. 4.2), causing the FSM to transition from state S_0 to S_1. The code fragment "`read(fd,acl_text,statbuf.st_size)`" emits a behavior node $\beta_2 = \boxed{\alpha_2}$, where $\alpha_2 = v \leftarrow \underline{\texttt{transfer}}(v_1, v_2, v_3, v_4, v_5)$, causing a transition from state S_1 to S_2. Finally, "`close(fd)`" emits a behavior node $\beta_3 = \boxed{\alpha_3}$, where $\alpha_3 = \underline{\texttt{disconnect}}(v_1)$, causing a transition from state S_2 to the accept state S_3.

Synthesizing the Migrated Program. Once the FSM accepts the behavior, Sprout inserts the extracted conditions into the sketch to obtain Fig. 3f. Hole `#3` is expected to contain a variable of type τ_{FilePath} according to Fig. 3e. Next, Sprout collects the minimal accepting code fragments, specifically those on Lines 2, 14, and 20, and the semantic variable context. Sprout removes the minimal accepting code fragments and replaces them with the sketch (discussed later in Alg. 3). In particular, Sprout inserts the sketch at the first original statement that emits the behavior node $\beta = \boxed{\alpha}$ where α is a `transfer` operation—the core goal of various I/O APIs—which is the call to `read` (Line 14). To fill in the holes, Sprout infers their semantic types (discussed later in Def. 3.2) based on the specification in Fig. 3a. In this example, hole `#0` has the semantic type $\tau_{\text{SizeTransferred}}$, hole `#1` has type τ_{FilePath}, and hole `#2` has type τ_{Buffer}. Based on these semantic types, Sprout searches for candidate semantic variables in the semantic variable context to fill in the holes.

Removing the original statements causes cascading changes to the code. Since Line 2 is removed, variable `fd` no longer holds the original value, which requires changes to Line 3 and Line 7. Sprout identifies these affected locations by tracking the value flows [84] and generates a new sketch for each such location. For Line 3, since the expression `fd < 0` represents an I/O error, the sketch simply removes the error-handling branch (discussed later in Def. 4.13). Line 7, on the other hand, emits behavior that needs to be preserved in the migrated program. Sprout searches among the

user-provided APIs and the existing POSIX APIs to identify one that emits behavior equivalent to `fstat` but does not rely on `fd`. The resulting migrated code invokes `stat` as the replacement.

After synthesis, Sprout produces the migrated function in Fig. 2b. It differs from the original function in Fig. 2a in several places: (1) The original invocations to `Uopen` (Line 2), `read` (Line 14), `close` (Line 20), and `fstat` (Line 7), along with the error-handling branch on Lines 3–6, have all been removed; (2) the synthesized code in Fig. 2b contains an invocation to `stat` (Line 2), an invocation to `get` (Line 11), auxiliary code based on the sketch (Lines 9–12), and a liftable condition extracted from `exim_open` (Line 10). By using the `get` API instead of POSIX APIs to perform I/O, the migrated program is expected to exhibit improved execution speed [88, 96].

3 Syntax of Core Language & Specification

Sprout works with programs written in a subset of C. We present formal notation for precisely specifying the semantics of I/O APIs and for reasoning about the correctness of migration.

Definition 3.1 (Abstract Syntax of Programs). Fig. 4 presents the abstract syntax of function definitions of an LLVM-like language that Sprout works with. A *program* p is a set of function definitions. A *function definition* d implements a user function y with a list of parameters $v_1, \ldots v_n$ and a block b as the function body. A *block* b is empty (Empty) or consists of *statements* (Single Statement), sequential structures (Sequential), conditional structures (Conditional), and loop structures (Loop). An evaluation statement assigns the result of the expression e to the variable v (Evaluation). An address statement creates a new abstract object o and assigns its address to the variable v (Address). A Get Element Pointer (GEP) statement points the variable v to the data field named c_{field} in the abstract object o, where the variable v' points to o (GEP). A load statement retrieves the data from the address stored in the variable v' and assigns the data to the variable v (Load). A store statement assigns the data stored in the variable v' to the address stored in the variable v (Store). A Phi statement assigns the value of variable v_i to the variable v when the control flow's predecessor block label is ι_i (Phi). A call statement invokes another function y with the parameters $v_1, \ldots, v_n$ and stores the return value in the variable v (Call). Function calls cannot be recursive. A return statement terminates the function and returns the value stored in the variable v (Return). An *expression* e is either a literal c (Constant), a variable v (Variable), or the computational results of an operation on sub-expression(s) (Unary Operation and Binary Operation).

In programs that invoke I/O APIs, variables often store values from several known categories, which we refer to as *semantic types*. For example, an `int` variable may represent the size of transferred data, a buffer size, an offset, or a file descriptor. Additionally, such a variable may hold values of different semantic types over its lifespan. Sprout leverages this insight to enhance the efficiency of the analysis and synthesis algorithms and to ensure the soundness of migration.

Definition 3.2 (Semantic Types). A *semantic type* τ_i is a type annotation for a variable v_i. Each semantic type falls into one of the following categories:

- **FileDescriptor:** Identifier for a system resource, used to perform I/O operations.
- **Buffer:** Memory region designated for storing data during I/O operations.
- **SizeToTransfer:** Number of bytes to read from or write to a resource.
- **SizeTransferred:** Number of bytes successfully read or written to a file or stream.
- **ErrorFlag:** Integer that signals the occurrence of an error during I/O operations.
- **Offset:** Position within a file, often used to specify where to begin reading or writing.
- **FileInfo:** Metadata about a file that provides essential details for management of the file.
- **FileSize:** The total amount of bytes in a file that a file descriptor points to.
- **FilePath:** A string that denotes the file's location within the file system.

$$
\begin{array}{lll}
d ::= \underline{\texttt{func}}\ y(v_1, \ldots, v_n)\ \{b\} & & \text{(Function Definition)} \\
\\
b ::= \varepsilon & & \text{(Empty)} \\
\mid s & & \text{(Single Statement)} \\
\mid b_1; b_2 & & \text{(Sequential)} \\
\mid \underline{\texttt{if}}\ (e)\ \{b_1\}\ \underline{\texttt{else}}\ \{b_2\} & & \text{(Conditional)} \\
\mid \underline{\texttt{while}}\ (e)\ \{b\} & & \text{(Loop)} \\
\\
s ::= v \leftarrow e & & \text{(Evaluation)} \\
\mid v \leftarrow \underline{\texttt{new}}\ o & & \text{(Address)} \\
\mid v \leftarrow \&(v' \rightarrow c_{\text{field}}) & & \text{(GEP)} \\
\mid v \leftarrow *v' & & \text{(Load)} \\
\mid *v \leftarrow v' & & \text{(Store)} \\
\mid v \leftarrow \Phi(\iota_1 : v_1, \ldots, \iota_n : v_n) & & \text{(Phi)} \\
\mid v \leftarrow \underline{\texttt{call}}\ y(v_1, \ldots, v_n) & & \text{(Call)} \\
\mid \underline{\texttt{return}}\ v & & \text{(Return)} \\
\\
e ::= c & & \text{(Constant)} \\
\mid v & & \text{(Variable)} \\
\mid \odot e & & \text{(Unary Operation)} \\
\mid (e_1 \oplus e_2) & & \text{(Binary Operation)}
\end{array}
$$

$d \in \text{FunctionDefinitions} \quad y \in \text{FunctionNames}$
$b, b_1, b_2 \in \text{Blocks} \quad s \in \text{Statements}$
$e \in \text{Expressions} \quad c, c_{\text{field}} \in \text{Literals} \subseteq \text{AbstractObjects}$
$o \in \text{AbstractObjects} \quad v, v', v_1, \ldots, v_n \in \text{Variables}$
$\iota_1, \ldots, \iota_n \in \text{BlockLabels} \quad \odot \in \{-, \sim, \neg\}$
$\oplus \in \text{Arithmetic} \cup \text{Comparors} \cup \text{Bitwise} \cup \text{Logical}$

Fig. 4. Abstract syntax of function definitions, adapted from FGS [7].

$$
\begin{array}{lll}
\psi ::= \underline{\texttt{spec}}\ y(v_1 : \tau_1, \ldots, v_n : \tau_n) : \tau_{\text{return}}\ \{\chi\} & & \text{(Behavioral Specification)} \\
\\
\chi ::= \varepsilon & & \text{(Empty)} \\
\mid \alpha & & \text{(Single)} \\
\mid \alpha; \chi & & \text{(List)} \\
\\
\alpha ::= v_{\text{buffer}} \leftarrow \underline{\texttt{alloc}}(v_{\text{size}}) & & \text{(Allocate)} \\
\mid v_{\text{handle}} \leftarrow \underline{\texttt{connect\_file}}(v_{\text{name}}, v_{\text{perm}}) & & \text{(Connect File)} \\
\mid v_{\text{handle}} \leftarrow \underline{\texttt{connect\_pipe}}() & & \text{(Connect Pipe)} \\
\mid v_{\text{handle}} \leftarrow \underline{\texttt{connect\_socket}}() & & \text{(Connect Socket)} \\
\mid \underline{\texttt{disconnect}}(v_{\text{handle}}) & & \text{(Disconnect)} \\
\mid \underline{\texttt{move\_offset}}(v_{\text{handle}}, v_{\text{offset}}) & & \text{(Move Offset)} \\
\mid v_{\text{offset}} \leftarrow \underline{\texttt{query\_offset}}(v_{\text{handle}}) & & \text{(Query Offset)} \\
\mid v_{\text{size\_transferred}} \leftarrow \underline{\texttt{transfer}}(v_{\text{location\_src}}, v_{\text{offset\_src}}, v_{\text{size\_to\_transfer}}, v_{\text{location\_dst}}, v_{\text{offset\_dst}}) & & \text{(Transfer)} \\
\mid \underline{\texttt{return}}\ v & & \text{(Return)}
\end{array}
$$

$\psi \in \text{BehavioralSpecifications} \quad y \in \text{FunctionNames}$
$\tau_1, \ldots, \tau_n, \tau_{\text{return}} \in \text{SemanticTypes}$
$\chi \in \text{OperationSequences} \quad \alpha \in \text{AtomicOperations}$
$v, v_1, \ldots, v_n, v_{\text{buffer}}, v_{\text{handle}}, v_{\text{name}}, v_{\text{offset}}, v_{\text{size}},$
$v_{\text{size\_transferred}}, v_{\text{location\_src}}, v_{\text{offset\_src}},$
$v_{\text{location\_dst}}, v_{\text{offset\_dst}}, v_{\text{size\_to\_transfer}} \in \text{Variables}$

Fig. 5. Abstract syntax of behavioral specifications for I/O APIs.

Another key insight is that the primary concern in I/O operations is ensuring the correct amount of data is transferred between specific locations. We therefore design a specification language that enables reasoning about the data and locations associated with I/O operations.

Definition 3.3 (Abstract Syntax of Specifications). Fig. 5 presents the abstract syntax of the specification for the functionality of I/O APIs and relevant memory operations. A *specification suite* ϱ is a set of behavioral specifications. A *behavioral specification* ψ specifies the functionality of an API function y as a sequence of I/O operations χ. The API function takes $v_1, \ldots, v_n$ as parameters. Each parameter v_i has a semantic type τ_i. The API returns a value of semantic type τ_{return}. An *operation sequence* χ contains a sequence of atomic operations. An *atomic operation* α may allocate an array with the length of v_{size} and assign its pointer to v_{buffer} (Allocate), open a file named v_{name} with permission v_{perm} and assign the file descriptor to v_{handle} (Connect File), open a pipe or a socket and assign the file descriptor to v_{handle} (Connect Pipe and Connect Socket), close the resource associated with the file descriptor v_{handle} (Disconnect), set the offset of the cursor for file v_{handle} as v_{offset} (Move Offset), get the offset of the cursor for file v_{handle} and assign it to v_{offset} (Query Offset), transfer at most $v_{\text{size\_to\_transfer}}$ bytes from a source location $v_{\text{location\_src}}$ starting at position $v_{\text{offset\_src}}$ to a destination location $v_{\text{location\_dst}}$ starting at position $v_{\text{offset\_dst}}$—a location can be a buffer or a

resource indicated by a file descriptor—and assign the number of bytes that are actually transferred to $v_{\text{size\_transferred}}$ (Transfer), or terminate the API function with a return value v (Return).

In a well-formed program, each function definition and specification declares a function with a distinct function name. Therefore, p is equivalent to a partial map from FunctionNames to FunctionDefinitions. In addition, ϱ is also equivalent to a partial map from FunctionNames to BehavioralSpecifications. That is, $p(y) \in \text{FunctionDefinitions}$ and $\varrho(y) \in \text{BehavioralSpecifications}$.

Let $\text{dom}(p)$ and $\text{dom}(\varrho)$ represent the sets of function names that are defined in $p$and ϱ, respectively. We have $\text{dom}(p) \cap \text{dom}(\varrho) = \emptyset$.

Example 3.4. Fig. 3a presents an example specification ψ. The function name y is `get` (Line 1). The parameters v_1 and v_2 are `arg0` and `arg1`, with semantic types $\tau_1 = \tau_{\text{FilePath}}$ and $\tau_2 = \tau_{\text{Buffer}}$, respectively (Line 2). The return type is $\tau_{\text{return}} = \tau_{\text{SizeTransferred}}$ (Line 6). The operation sequence χ consists of the following atomic operations: `f=connect_file(arg0,Read)`, `n=transfer(f,0,f.size,arg1,0)`, `disconnect(f)`, and `return n` (Lines 3–6).

Definition 3.5 (Non-essential Functions). A *non-essential function* y_n is a function that performs only auxiliary tasks and is not crucial to the core functionality of the program.

Example non-essential functions may perform logging, printing, or memory initialization.

In a well-formed program, the function in each call statement is distinctly a user function from a function definition d, an API function from a behavioral specification ψ, or a non-essential function. That is, $\text{dom}(p) \cup \text{dom}(\varrho) \cup Y_n \supseteq \text{called}(p)$ where $\text{called}(p) \subseteq \text{FunctionNames}$ is the set of all function names that appear in a call statement in the program p, and Y_n is a finite set of non-essential functions that may be used in the program.

Definition 3.6 (Program Context). A *program context* $P = \langle p, \varrho, Y_n \rangle$ is a tuple of a program p, a specification suite ϱ, and a set of non-essential functions Y_n. It is equivalent to a map from each function name $y \in \text{called}(p) \setminus Y_n$ to either a function definition $d = p(y)$ or a behavioral specification $\psi = \varrho(y)$, i.e. $P(y) \in \text{FunctionDefinitions} \cup \text{BehavioralSpecifications}$.

4 Formalizing I/O Behaviors

We introduce notation for reasoning about behavioral equivalence of the migrated programs.

4.1 I/O Behaviors of Programs

Definition 4.1 (Intermediate States). An *intermediate state* $\sigma = \langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_{\Phi} \rangle$ consists of six parts:

(1) The pointer analysis result $\sigma_{\text{PTR}} : \text{Variables} \rightharpoonup \text{Variables} \cup \text{AbstractObjects}$.
   - $v \mapsto v'$ denotes that the variable v is a pointer that points to another variable v'.
   - $v \mapsto o$ denotes that the variable v is a pointer that points to an abstract object o.

(2) The object field map $\sigma_{\text{OFM}} : \text{AbstractObjects} \times \text{FieldNames} \rightharpoonup \text{Variables} \cup \text{AbstractObjects}$, where $\text{FieldNames} \subseteq \text{Literals}$. The object field map is used mainly to handle the GEP operation.
   - $o.c_{\text{field}} \mapsto v$ denotes that the field indicated by c_{field} of the object o is not an object but equivalent to a variable v. Thus, the GEP operation returns a pointer to v.
   - $o.c_{\text{field}} \mapsto o'$ denotes that the field indicated by c_{field} of the object o is another object o', so the GEP operation returns a pointer to o'.

(3) The semantic typing map $\sigma_{\text{STM}} \subseteq \text{Variables} \times \text{SemanticTypes}$. For each pair $\langle v, \tau \rangle \in \sigma_{\text{STM}}$, the variable v has the semantic type τ. It annotates a program variable with some predefined semantics to facilitate the analysis and synthesis in Sec. 5. A variable may have more than one semantic type. For example, an integer variable may simultaneously have two semantic types τ_{size} and τ_{offset} in a single semantic typing map.

(4) The file offset map σ_{FOM} : FileDescriptiors $\rightharpoonup$ AbstractObjects $\cup$ Unknown, where FileDescriptors $\subseteq$ AbstractObjects. The file offset map records the cursor offset for each connected file descriptor, where the offset is either a known value, which is represented by an abstract object o, or is unknown. It enables Sprout to determine the file offsets of file descriptors associated with different aliased variables by using the atomic operations defined in Move Offset and Query Offset, along with σ_{PTR}.
(5) The loop context $\sigma_{\mathsf{LC}} \in \mathsf{LoopContexts}$, where $\mathsf{LoopContexts} = \{\bot\} \cup \{\langle \iota_b, \sigma_{\mathsf{LC}} \rangle \mid b \in \mathsf{Blocks} \wedge \sigma_{\mathsf{LC}} \in \mathsf{LoopContexts}\}$. The loop context is a stack where each element represents a nesting loop to which the intermediate state belongs. $\sigma_{\mathsf{LC}} = \bot$ indicates that the stack is empty so the intermediate state is not in any loop. $\sigma_{\mathsf{LC}} = \langle \iota_b, \sigma'_{\mathsf{LC}} \rangle$ indicates that b is the loop at the top of the stack, which is the closest loop, and the tail of the stack is σ'_{LC}. ι_b is a unique label of the loop block b. Such labels are unique for each block in the program. For those blocks that are syntactically identical but appear in different positions, their labels are distinct.
(6) The Phi context $\sigma_\Phi \in \{\iota_b \in \mathsf{BlockLabels} \mid b = s\}$, where $\sigma_\Phi = \iota_s$ indicates that the last executed statement in the behavior path is s. The Phi context is used to evaluate the Phi statement.

Definition 4.2 (Behavior Trees). A *behavior tree* η describes the I/O behavior of a program p and is composed of a set of behavior nodes. A *behavior node* β may be one of the following:

- A begin node $(\top)$, which is the root of the behavior tree and represents the program entry.
- An end node $(\bot)$, which is the only kind of leaf nodes in the behavior tree and represents the program termination.
- A statement node $\boxed{s}$, which represents a non-call statement s that does not emit I/O operation.
- An operation node $\boxed{\alpha}$, which represents an atomic operation α emitted by a call statement that calls a function $y \in \mathrm{dom}(\varrho)$ in the program p.
- A branching node $\langle e \rangle$, which represents a condition expression e. The branching node is the only kind of internal node that has two children, which are the successors when e is evaluated to true and false, respectively. All other internal nodes, including statement nodes, operation nodes, loop entry nodes, and loop exit nodes, all have one single child.
- A loop entry node $\boxed{b}\rangle$, which represents the start of the body of a loop block b. It pushes the label ι_b of the loop block into the loop context in the intermediate state.
- A loop exit node $\langle\boxed{b}$, which represents the end of the body of a loop block b. It pops the label ι_b from the loop context. Loop entry and exit nodes always appear in pairs in a behavior path.

For notation convenience, $\hat{\eta}$ denotes a sub-tree of a behavior tree. For a behavior node β that is not a branching node, $\beta\hat{\eta}$ denotes a behavior (sub-)tree where the root node is β and root node of the sub-tree $\hat{\eta}$ is the only child of β.

Definition 4.3 (Construction of Behavior Trees). Given a program context $P = \langle p, \varrho, Y_n \rangle$, consider a block b whose successor emits the behavior sub-tree $\hat{\eta}$. $\langle p, \varrho, Y_n \rangle \vdash b, \hat{\eta} \rightsquigarrow \hat{\eta}'$ denotes that the behavior sub-tree emitted by both the block b and its successor combined is $\hat{\eta}'$.

Given a program context $P = \langle p, \varrho, Y_n \rangle$, entry($p$) denotes the body block of the entry function in the program p. When executing from entry(p) to termination emits $\hat{\eta}'$, i.e. $\langle p, \varrho, Y_n \rangle \vdash \mathrm{entry}(p), (\bot) \rightsquigarrow \hat{\eta}'$, the behavior tree of the program is $\eta = (\top)\hat{\eta}'$.

Intuitively, the behavior tree of the program p consists of the root node $(\top)$ and the sub-tree emitted by the entry function body entry(p). Since the program terminates after executing the entry function body, the successor sub-tree $\hat{\eta}$ to construct the sub-tree of entry(p) is $(\bot)$, which indicates the program termination.

Example 4.4. Consider a program whose entry function body is as in Fig. 9a, which was extracted from Exim [27]. Given the specification of `read` in Fig. 9f, the behavior tree of Fig. 9a is Fig. 9b.

Definition 4.5 (Behavior Paths). A *behavior path* $B \in \eta$ is a path from the root node (begin node) to a leaf node (end node) in a behavior tree η.

For notation convenience, B may also denote a continuous sub-path of a behavior path.

Definition 4.6 (Path Conditions). For a behavior path B, the *path condition* of B is an expression $\text{cond}(B)$ such that the program will execute along the path B iff $\text{cond}(B)$ is satisfied. Let $\langle e_1 \rangle \dots \langle e_n \rangle \in B$ be the branching nodes in B where the path takes their true branches. Let $\langle e'_1 \rangle \dots \langle e'_m \rangle \in B$ be the branching nodes in B where the path takes their false branches. The path condition of the behavior path B is $\text{cond}(B) = \left(\bigwedge_{i=1}^{n} e_i\right) \wedge \left(\bigwedge_{j=1}^{m} \neg e'_j\right)$.

```
s0: *len = len0;
...
s1: len1 = *len;
s2: ret = read(fd,buf,len1);
s3: *bytes = ret;
...
s4: bytes1 = *bytes;
s5: len2 = *len;
```

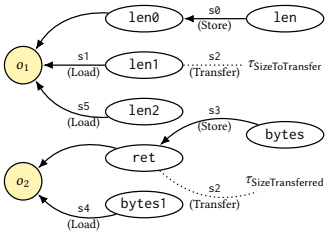


Fig. 6. Semantic types of variables in the intermediate state.

Definition 4.7 (Core Operations). Atomic operations directly involved in I/O and resource lifespan management, including `connect_file`, `connect_pipe`, `connect_socket`, `disconnect`, and `transfer`, are classified as *core operations*. An operation node $\boxed{\alpha}$ where α is a core operation is a *core operation node*. Other behavior nodes β are classified as *non-core operation nodes*.

Definition 4.8 (Behavior Node Kinds). For each core operation node $\boxed{\alpha}$, its *behavior node kind* k is the operation kind of α, which may be $k_{\text{connect\_file}}$, $k_{\text{connect\_pipe}}$, $k_{\text{connect\_socket}}$, $k_{\text{disconnect}}$, or k_{transfer}. The behavior node kind of any non-core operation node is $k_{\text{non-core}}$.

Definition 4.9 (Evaluation of Intermediate States). Fig. 7 presents the inference rules for evaluating intermediate states along behavior paths. $\sigma, \beta \Downarrow \sigma'$ denotes that given a behavior node β and the intermediate state σ before β, the intermediate state after β is σ'.

Remark 4.10. The behavior tree fully determines the intermediate state between any two adjacent behavior nodes according to the rules in Def. 4.9.

Example 4.11. Fig. 6 presents an intermediate state after evaluating statements on the left. On the right, each node represents an abstract object (yellow) or a variable (white). Solid lines represent the points-to relations and dashed lines represent the semantic types of variables. The labels on the edges represent the statements and the rules in Fig. 7 that SPROUT uses to establish the relation. Based on the intermediate state, SPROUT deduces that the variable `len2` shares the semantic type $\tau_{\text{SizeToTransfer}}$ with `len1` since both of them point to the same abstract object o_1. Similarly, `bytes1` is aliased with `ret` and is thus inferred to share the semantic type $\tau_{\text{SizeTransferred}}$ with `ret`.

4.2 API Migration Problem

As outlined in Fig. 1, SPROUT takes the following inputs to perform migration: the original program p_1 and its compiled LLVM IR containing debug information, the specification suite ϱ_1 for the existing I/O APIs in p_1, and the specification suite ϱ_2 for the new I/O APIs. SPROUT modifies p_1 to produce a migrated program p_2 that is behaviorally equivalent to p_1. We formalize the API migration problem below and present SPROUT's algorithms in Sec. 5.

$$\frac{s = v \leftarrow \underline{\texttt{new}}\ o \quad \sigma'_{\text{PTR}} = \sigma_{\text{PTR}}[v \mapsto o]}{\langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle, \boxed{s} \Downarrow \langle \sigma'_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle} \quad \text{(New)}$$

$$\frac{s = v \leftarrow \&(v' \rightarrow c_{\text{field}}) \quad o = \sigma_{\text{PTR}}(v') \quad o_{\text{field}} = \sigma_{\text{OFM}}(o, c_{\text{field}}) \quad \sigma'_{\text{PTR}} = \sigma_{\text{PTR}}[v \mapsto o_{\text{field}}]}{\langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle, \boxed{s} \Downarrow \langle \sigma'_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle} \quad \text{(GEP Object)}$$

$$\frac{\begin{array}{c} s = v \leftarrow *v' \quad v_{\text{target}} = \sigma_{\text{PTR}}(v') \quad \sigma'_{\text{PTR}} = \sigma_{\text{PTR}}[v \mapsto \sigma_{\text{PTR}}(v_{\text{target}})] \\ \sigma'_{\text{STM}} = \sigma_{\text{STM}} \cup \{\langle v, \tau \rangle \mid \langle v_{\text{target}}, \tau \rangle \in \sigma_{\text{STM}}\} \end{array}}{\langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle, \boxed{s} \Downarrow \langle \sigma'_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma'_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle} \quad \text{(Load)}$$

$$\frac{\begin{array}{c} s = *v \leftarrow v' \quad v_{\text{target}} = \sigma_{\text{PTR}}(v) \quad \sigma'_{\text{PTR}} = \sigma_{\text{PTR}}[v_{\text{target}} \mapsto \sigma_{\text{PTR}}(v')] \\ \sigma'_{\text{STM}} = \sigma_{\text{STM}} \cup \{\langle v_{\text{target}}, \tau \rangle \mid \langle v', \tau \rangle \in \sigma_{\text{STM}}\} \end{array}}{\langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle, \boxed{s} \Downarrow \langle \sigma'_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma'_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle} \quad \text{(Store)}$$

$$\frac{\begin{array}{c} \alpha = v_{\text{handle}} \leftarrow \underline{\texttt{connect\_file}}(v_{\text{name}}, v_{\text{perm}}) \quad o \text{ is a fresh object} \quad \sigma'_{\text{PTR}} = \sigma_{\text{PTR}}[v_{\text{handle}} \mapsto o] \\ \sigma'_{\text{STM}} = \sigma_{\text{STM}} \cup \{\langle v_{\text{handle}}, \tau_{\text{FileDescriptor}} \rangle, \langle v_{\text{name}}, \tau_{\text{FilePath}} \rangle\} \quad \sigma'_{\text{FOM}} = \sigma_{\text{FOM}}[o \mapsto 0] \end{array}}{\langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle, \boxed{\alpha} \Downarrow \langle \sigma'_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma'_{\text{STM}}, \sigma'_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle} \quad \text{(Connect File)}$$

$$\frac{\begin{array}{c} \alpha = \underline{\texttt{disconnect}}(v_{\text{handle}}) \quad \langle v_{\text{handle}}, \tau_{\text{FileDescriptor}} \rangle \in \sigma_{\text{STM}} \\ o = \sigma_{\text{PTR}}(v_{\text{handle}}) \quad \sigma'_{\text{FOM}} = \sigma_{\text{FOM}}[o \mapsto \text{null}] \end{array}}{\langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle, \boxed{\alpha} \Downarrow \langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma'_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle} \quad \text{(Disconnect)}$$

$$\frac{\begin{array}{c} \alpha = v_{\text{size\_transferred}} \leftarrow \underline{\texttt{transfer}}(v_{\text{location\_src}}, v_{\text{offset\_src}}, v_{\text{size\_to\_transfer}}, v_{\text{location\_dst}}, v_{\text{offset\_dst}}) \\ o \text{ is a fresh object} \quad \sigma'_{\text{PTR}} = \sigma_{\text{PTR}}[v_{\text{size\_transferred}} \mapsto o] \\ \sigma'_{\text{STM}} = \sigma_{\text{STM}} \cup \{\langle v_{\text{size\_transferred}}, \tau_{\text{SizeTransferred}} \rangle, \langle v_{\text{size\_to\_transfer}}, \tau_{\text{SizeToTransfer}} \rangle\} \end{array}}{\langle \sigma_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle, \boxed{\alpha} \Downarrow \langle \sigma'_{\text{PTR}}, \sigma_{\text{OFM}}, \sigma'_{\text{STM}}, \sigma_{\text{FOM}}, \sigma_{\text{LC}}, \sigma_\Phi \rangle} \quad \text{(Transfer)}$$

Fig. 7. Representative inference rules for evaluation of intermediate states.

$$\frac{\begin{array}{c} B_1 = \boxed{\alpha_1} \quad \sigma_1 = \text{the intermediate state before } \boxed{\alpha_1} \\ B_2 = \boxed{\alpha_2} \quad \sigma_2 = \text{the intermediate state before } \boxed{\alpha_2} \\ \forall \text{ corresponding variables } v_1, v_2 \text{ in } \alpha_1, \alpha_2,\ \sigma_1^{\text{PTR}}(v_1) = \sigma_2^{\text{PTR}}(v_2) \end{array}}{B_1 \equiv B_2} \qquad \frac{\begin{array}{c} B_1 = \beta \\ \beta \text{ is not an operation node} \\ B_2 = \varepsilon \end{array}}{B_1 \equiv B_2 \quad B_2 \equiv B_1} \qquad \frac{\begin{array}{c} B_1 \equiv B_2 \\ B_3 \equiv B_4 \end{array}}{B_1 B_3 \equiv B_2 B_4}$$

Fig. 8. Big-step inference rules of the equivalence between two behavior sub-paths.

Definition 4.12 (Equivalence between Behavior Paths). $B_1 \equiv B_2$ denotes that two behavior paths B_1 and B_2 are behaviorally equivalent. Fig. 8 presents the big-step inference rules of the equivalence between two behavior sub-paths.

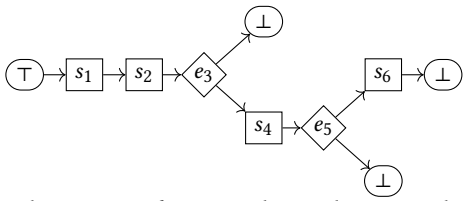
Definition 4.13 (Equivalence between Behavior Trees). Two behavior trees η_1 and η_2 are behaviorally equivalent iff $\forall\, B_1 \in \eta_1,\ \big(\text{cond}(B_1)$ does not represent I/O errors $\implies \exists\, B_2 \in \eta_2,\ \big(\text{cond}(B_1) \implies \text{cond}(B_2)\big) \wedge (B_1 \equiv B_2)\big)$ and $\forall\, B_2 \in \eta_2,\ \big(\text{cond}(B_2)$ does not represent I/O errors $\implies \exists\, B_1 \in \eta_1,\ \big(\text{cond}(B_2) \implies \text{cond}(B_1)\big) \wedge (B_2 \equiv B_1)\big)$.

Remark 4.14. To identify expressions that represent I/O errors, we establish a set of predefined evaluation rules for variables based on their semantic types. These rules are implemented with a flow-sensitive pointer analysis to identify the semantic types of variables. For example, variables of semantic type $\tau_{\text{FileDescriptor}}$ must be non-negative integers, while variables with semantic type τ_{Buffer} must not be null. Expressions that violate these rules are deemed to represent I/O errors.

Remark 4.15. Def. 4.13 treats error-handling behavior paths separately, due to the nature of I/O APIs. Modern storage systems often provide different abstractions that lead to various types of potential errors. The errors that may arise in the original programs, such as programs that invoke POSIX APIs, may no longer occur under new storage systems. For example, the `get` API in Fig. 3a eliminates the need to use file descriptors [88, 96]. As a result, any original error-handling code that

```
1 uschar * next = buffer;
2 uschar * end = next + len;
3 while (next < end) {
4   ssize_t got = read(fd, next, end-next);
5   if (got == -1 && errno == EINTR) return;
6   next += got;
7 }
8 return len;
```

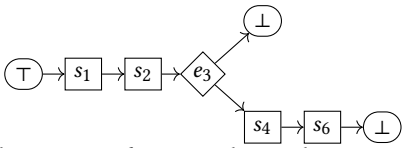
(a) Code snippet extracted from Exim [27].



(b) Behavior tree of Fig. 9a, where subscripts indicate the line numbers in Fig. 9a.

```
1 uschar * next = buffer;
2 uschar * end = next + len;
3 while (next < end) {
4   ssize_t got = read(fd, next, end-next);
5   next += got;
6 }
7 return len;
```

(c) Code snippet obtained by removing error-handling branches from Fig. 9a.



(d) Behavior tree of Fig. 9c, where subscripts indicate the line numbers in Fig. 9a. This tree is equivalent to Fig. 9b according to Def. 4.13.

```
1 *have = 0;
2 do {
3   get = len - *have;
4   if (get > max) get = max;
5   ret = read(state->fd, buf + *have, get);
6   if (ret <= 0) break;
7   *have += (unsigned)ret;
8 } while (*have < len);
```

(e) Loop extracted from `gz_load` in zlib [49].

```
1 int read(int fildes, void *buf, int nbyte) {
2 ParamType:(arg0:FileDescriptor,arg1:Buffer);
3 Semantics:{n=transfer(arg0,arg0.offset,
4                       arg2,arg1,0),
5            move_offset(arg0,arg2)};
6 Return: n:SizeTransferred;
7 }
```

(f) API specification for POSIX `read`.

Fig. 9. Example code snippets, behaviors, and specification.

deals with a missing file descriptor is no longer meaningful. It is therefore natural and reasonable to disregard similar error-handling code when migrating across different storage systems.

Example 4.16. Fig. 9b and Fig. 9d present a pair of equivalent behavior trees emitted by Fig. 9a and Fig. 9c, respectively. Fig. 9a differs from Fig. 9c with the presence of error-handling conditional blocks on Line 5. Since the condition expression e_5 = `got == −1 ∧ errno == EINTR` in Fig. 9b represents I/O errors, the behavior path ⊤ s_1 s_2 e_3 s_4 e_5 ⊥ is discarded. The remaining program paths in these two trees are correspondingly equivalent.

Definition 4.17 (API Migration Problem). Given two specification suites ϱ_1 and ϱ_2, a program p_1, and a set of non-essential functions Y_n, where $\text{called}(p_1) \cap \text{dom}(\varrho_1) \neq \emptyset$ and $\text{called}(p_1) \cap \text{dom}(\varrho_2) = \emptyset$, an *API migration problem* is to find a new program p_2 such that $\text{called}(p_2) \cap \text{dom}(\varrho_2) \neq \emptyset$ and $\exists\, \eta_1, \eta_2 \in \text{BehaviorTrees},\ \big(\langle p_1, \varrho_1, Y_n\rangle \rightsquigarrow \eta_1\big) \wedge \big(\langle p_2, \varrho_1 \cup \varrho_2, Y_n\rangle \rightsquigarrow \eta_2\big) \wedge \big(\eta_1 \text{ is equivalent to } \eta_2\big)$.

Remark 4.18. Def. 4.17 requires that p_2 invokes at least one new API in $\text{dom}(\varrho_2)$ but does not require eliminating all invocations to the old APIs in $\text{dom}(\varrho_1)$. Partially migrated programs are allowed, which captures a realistic scenario for the migration of I/O APIs. For modern storage systems, many new APIs in the file I/O domain have been purposefully designed to interoperate seamlessly with existing POSIX APIs [38, 43, 88, 96]. Thus, partially migrated programs can execute correctly, albeit with suboptimal performance—the greater the reliance on legacy APIs, the slower the execution. For example, the files accessed with `get` in Fig. 3a can still be accessed with POSIX APIs `open`/`read`/`close` [88, 96]. By migrating core APIs as extensively as possible, Sprout aims to maximize the benefits of advancements in storage systems while preserving the program semantics.

5 Automated Migration Guided by Semantic Types of Variables

Conceptually, Sprout first derives the behaviors of functions in the original program, then performs typestate analysis to match the relevant code fragments that invoke the old APIs, and finally synthesizes new code to invoke the new APIs. All these steps leverage the inferred semantic types that Sprout infers for each variable. We present these algorithms below.

5.1 Deriving Function Behaviors

Ideally, all relevant invocations of the existing APIs should appear within a single functional scope. In practice, software code often contains multiple abstraction layers that encapsulate the APIs in ϱ_1 to enhance readability and maintainability. To address this complication, we introduce the concept of aliased functions, to collapse the multiple abstraction layers surrounding the I/O APIs. The goal is to derive I/O behaviors for a broader range of functions encountered in practice and to identify groups of indirect invocations of the existing APIs that would be difficult to migrate otherwise.

5.1.1 Simplifying Loops. We begin by pre-processing common loop structures and summarizing them with their equivalent I/O behaviors. We collected a set of representative loop templates that invoke I/O APIs. For each of these loops, we manually proved that (1) the loop is guaranteed to terminate and (2) the loop transfers a specified amount of data from one location to another at a designated offset, while adjusting the offset accordingly. Therefore, the loop block is equivalent to a sequence of two atomic operations: a `transfer` operation and a `move_offset` operation.

Example 5.1. Fig. 9e presents a representative loop b_{Loop} for which we prove its termination and it is behaviorally equivalent to a single call to `read` in Fig. 9f, which reads `len` bytes at once without requiring a loop. Sprout concludes that the I/O behavior emitted by b_{Loop} is equivalent to that of the atomic operations $v \leftarrow \underline{\texttt{transfer}}(\texttt{state->fd}, o_c, n, \texttt{buf}, 0)$ and $\underline{\texttt{move\_offset}}(\texttt{state->fd}, v)$ such that $v = \texttt{len}$ if $n \geq \texttt{len}$, otherwise $v = n$.

For each loop block in the program p, Sprout attempts to iteratively apply a series of refinement steps. Each step ensures that the refined loop block emits I/O behavior equivalent to that of the original loop. These refinement steps may involve: (1) swapping the branches and reversing the condition of a conditional block, or (2) replacing a conditional block with its false branch if the condition evaluates to true iff it represents I/O errors. This process continues until the refined loop block conforms to a predefined template or until all possible refinements have been exhausted. When Sprout determines that the original loop block is equivalent to a template, it concludes that the loop emits certain I/O behaviors proven to be equivalent to those of the template.

Example 5.2. One step of the refinement simplifies the loop in Fig. 9a to the loop in Fig. 9c.

5.1.2 Merging Aliased Functions. Sprout identifies aliased functions to facilitate migration.

Definition 5.3 (Local Scopes). In a function definition $d = \underline{\texttt{func}}\ y(v_1, \ldots, v_n)\ \{b\}$, the *local scope* consists of local variables assigned within the function body b, along with inter-procedural values such as global variables and function parameters modified by b.

Definition 5.4 (Liftable Conditions). For a conditional block "`if` (e) $\{b_1\}$ `else` $\{b_2\}$", the condition expression e is a *liftable condition* iff e has no data dependencies on the local scope of the function. Otherwise, e is a *non-liftable condition.*

Moving liftable conditions outside the function does not alter the program's semantics.

Definition 5.5 (Aliased Functions). Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$ and two functions y_1, y_2, where y_1 is a user function defined by a function definition $d = \underline{\texttt{func}}\ y_1(v_1, \ldots, v_n)\ \{b\}$ in

the program p_1, and some statements in b call the function y_2. We say y_1 is an *aliased function* of y_2 if IsAliased(y_1, y_2, Y_n) in Alg. 1 returns true.

Alg. 1 checks whether a function $y_1 \in \text{dom}(p_1)$ is an alias of another function $y_2 \in \text{dom}(p_1) \cup \text{dom}(\varrho_1)$. The algorithm in IsAliased starts by splitting the essential paths of y_1. In particular, SplitEssentialPaths first simplifies the loops in the function body and collects all paths in the control-flow graph of the simplified function body. If any path contains non-liftable conditions, y_1 is not identified as an alias of y_2. If some path with liftable conditions E invokes only the function y_2 and non-essential functions in Y_n, y_1 is identified as an alias of y_2 under the liftable conditions E.

Example 5.6. Fig. 3d presents an example function in Exim [27] that Sprout identifies as an aliased function of the POSIX API `open()` function.

Definition 5.7 (Atomic I/O Functions). Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$. A function $y \in \text{dom}(p_1) \cup \text{dom}(\varrho_1)$ is an *atomic I/O function* if $y \in \text{dom}(\varrho_1)$ or there exists some $y' \in \text{dom}(\varrho_1)$ where y is an alias of y', i.e. $\exists\, y' \in \text{dom}(\varrho_1),\ \text{IsAliased}(y, y', Y_n)$.

Definition 5.8 (Composite I/O Functions). Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$. A function $y \in \text{dom}(p_1)$ is a *composite I/O function* if y is not an atomic I/O function and its definition $p_1(y)$ contains at least one call statement to an atomic I/O function or another composite I/O function.

A call statement emits I/O behaviors iff it calls an atomic I/O function or a composite I/O function.

Given an initial set of atomic I/O functions $Y_a = \text{dom}(\varrho_2)$, Sprout derives the aliased functions for each atomic I/O function and stores the analysis result. For each $y_a \in Y_a$, Sprout analyzes the call graph of the program p_1 to identify all caller functions of y_a and determines if each caller $y \in \text{dom}(p_1)$ is an alias of y_a. Once y is identified as aliased with y_a, y is added to Y_a. This process is repeated iteratively in topological order until Y_a reaches saturation. At this point, the functions in Y_a are identified as atomic I/O functions, while the functions in $\text{dom}(p_1) \setminus Y_a$ that call atomic I/O functions are identified as composite I/O functions.

5.2 Identifying Code Fragments to Modify

To identify the code fragments in p_1 that emit behaviors equivalent to that of an API $\psi \in \varrho_2$, Sprout matches the behaviors of the paths in p_1 against the FSM that accepts the behavior of the new API ψ. For each accepted behavior, the corresponding code fragment in p_1 will be replaced with synthesized code that invokes ψ (discussed later in Sec. 5.3) to produce the new program p_2.

5.2.1 *Constructing State Machines that Represent New API Semantics.*

A key step is identifying a code fragment in p_1 that exhibits behavior equivalent to that of a new API $y \in \text{dom}(\varrho_2)$. Sprout achieves this goal by matching the behavior paths in the behavior tree of p_1 against an FSM that accepts the behavior sub-paths emitted from the new API specification $\varrho_2(y)$. For each accepted behavior path, Sprout replaces the corresponding code fragment in p_1 with the synthesized code that invokes the new API y instead, to produce the new program p_2.

Definition 5.9 (Semantic Guards). A *semantic guard* g is a predicate that enforces a constraint on the intermediate state to ensure the legality of an I/O API invocation. SemanticGuards denotes the non-empty finite set of possible semantic guards.

Definition 5.10 (State Machines). Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$. An *I/O finite state machine (FSM)* for a new API specification $\psi \in \varrho_2$ is a quadruple $\langle \mathcal{S}, S_0, \mathcal{F}, \delta \rangle$, where:

- the alphabet consists of all behavior nodes from the behavior tree of the original program p_1.
- $\mathcal{S}$ is a non-empty set of machine states. $S_0 \in \mathcal{S}$ is the start state. $\mathcal{F} \subseteq \mathcal{S}$ is a non-empty set of accept states.

Algorithm 1 Function alias detection.

Input: $y_1 \in \mathrm{dom}(p_1)$, a user function.
Input: $y_2 \in \mathrm{dom}(p_1) \cup \mathrm{dom}(\varrho_1)$, a function called by y_1.
Input: $Y_n \subseteq$ FunctionNames, a set of non-essential functions.
Output: A boolean value that indicates whether y_1 is an alias of y_2.

function IsAliased(y_1, y_2, Y_n)
  return $\exists \langle b_{\text{path}}, E\rangle \in$ SplitEssentialPaths(y_1), $y_2 \in \text{called}(b_{\text{path}}) \wedge \text{called}(b_{\text{path}}) \subseteq Y_n \cup \{y_2\}$

Input: $y \in$ FunctionNames, a user-defined function.
Output: A set of tuples $\langle b_{\text{path}}, E\rangle$ where b_{path} is a path and E is a set of liftable conditions.

function SplitEssentialPaths(y)
  $d \leftarrow p(y)$
  func $y(v_1, \ldots, v_n)$ $\{b\} \leftarrow d$
  $b' \leftarrow b$ with loops simplified according to Sec. 5.1.1
  if b' contains loops that cannot be simplified **then return** $\emptyset$
  $\mathcal{B} \leftarrow$ all paths in the control-flow graph of b'
  $M \leftarrow \emptyset$
  $E_l \leftarrow$ all liftable conditions in b'
  for $b_{\text{path}} \in \mathcal{B}$ **do**
    $E_{\text{path}} \leftarrow$ all `if` conditions in b_{path}
    if $E_{\text{path}} \nsubseteq E_l$ **then return** $\emptyset$
    $E \leftarrow \emptyset$
    for $e \in E_{\text{path}}$ **do**
      if b_{path} takes the true branch of e **then** $E \leftarrow E \cup \{e\}$
      else if b_{path} takes the false branch of e **then** $E \leftarrow E \cup \{\neg e\}$
    $M \leftarrow M \cup \{\langle b_{\text{path}}, E\rangle\}$
  return M

Algorithm 2 FSM construction for an I/O API.

Input: $\psi \in \varrho_2$, the specification of a new API.
Output: an I/O finite state machine $\langle \mathcal{S}, S_0, \mathcal{F}, \delta\rangle$ for ψ.

function ConstructFSM(ψ)
  spec $y(v_1 : \tau_1, \ldots, v_n : \tau_n) : \tau_{\text{return}}$ $\{\chi\} \leftarrow \psi$
  return ConstructRecursively(χ)

Input: $\chi \in$ OperationSequences, the operation sequence in a new API specification ψ.
Output: an I/O finite state machine $\langle \mathcal{S}, S_0, \mathcal{F}, \delta\rangle$ for ψ.

function ConstructRecursively(χ)
  $S_0 \leftarrow$ a fresh state
  if $|\chi| = 0$ **then return** $\langle \{S_0\}, S_0, \{S_0\}, \emptyset\rangle$
  $\alpha; \chi' \leftarrow \chi$
  if α is not a core operation **then**
    return ConstructRecursively(χ')
  $\langle \mathcal{S}', S_0', \mathcal{F}', \delta'\rangle \leftarrow$ ConstructRecursively(χ')
  $\mathcal{S} \leftarrow \mathcal{S}' \cup \{S_0\}$
  $G \leftarrow$ the semantic guards that correspond to α
  $k \leftarrow$ the kind of α
  $\delta \leftarrow \delta' \cup \{\langle S_0, k, G, S_0'\rangle, \langle S_0, k_{\text{non-core}}, \emptyset, S_0\rangle\} \cup \{\langle S_0, k', \{\neg g_{\text{FdConsistency}}\}, S_0\rangle \mid k' \in \text{BehaviorNodeKinds}\}$
  ▷ $g_{\text{FdConsistency}}$ *is the predicate that "β operates on the same* v_{handle}*"*
  return $\langle \mathcal{S}, S_0, \mathcal{F}', \delta\rangle$

Algorithm 3 Dependency-driven synthesis.

Input: $\psi \in \varrho_2$, the specification of new I/O API.
Input: $L = \langle s_1 \ldots s_n\rangle \subseteq$ Statements, the minimal accepting code fragment of a behavior sub-path B accepted by the I/O finite state machine of ψ.
Input: $\Gamma \in$ SemanticVariableContexts, the semantic variable context of a behavior sub-path B accepted by the I/O finite state machine of ψ.
Output: a boolean value indicating whether the synthesis is successful.

function Synthesize(ψ, L, Γ)
  $\kappa \leftarrow$ the sketch that corresponds to ψ
  $\kappa' \leftarrow$ FillSketchHoles(κ, Γ)
  if κ' is nil **then return** false
  $s \leftarrow$ the statement in L that corresponds to κ'
  $K \leftarrow \{\langle s, \kappa'\rangle\}$
  $V \leftarrow$ the variables defined in L but not defined in κ'
  while $V \neq \emptyset$ **do**
    $v \leftarrow$ a variable in V
    $V \leftarrow V \setminus \{v\}$
    for $s \in$ all use sites of v **do**
      ▷ *based on an over-approximated value-flow analysis*
      $\kappa \leftarrow$ a sketch that corresponds to s
      $\kappa' \leftarrow$ FillSketchHoles(κ, Γ)
      if κ' is nil **then return** false
      $V' \leftarrow$ the variables defined in s but not defined in κ'
      $V \leftarrow V \cup V'$
      $K \leftarrow K \cup \{\langle s, \kappa'\rangle\}$
  remove all statements in L from the program p_1
  for $\langle s, \kappa'\rangle \in K$ **do**
    apply κ' to the program p_1 at the location of s
  return true

Input: $\kappa \in$ Sketches, a provided sketch.
Input: $\Gamma \in$ SemanticVariableContexts, the semantic variable context of a behavior sub-path B accepted by the I/O finite state machine of ψ.
Output: A filled sketch or nil, which indicates that it is impossible to fill the sketch.

function FillSketchHoles(κ, Γ)
  if κ has been completed and has no holes **then return** κ
  $h \leftarrow$ the first hole from κ
  $\tau \leftarrow$ the semantic type of h
  if no variable has semantic type τ in Γ **then**
    $\kappa' \leftarrow$ a sketch that returns a variable whose type is τ
    if κ' is nil **then return** nil
      ▷ *no sketch returns a variable whose type is* τ
    $v \leftarrow$ a fresh variable
    $h_{\text{return}} \leftarrow$ the hole of the returned variable in κ'
    return FillSketchHoles($\kappa'[v \,/\, h_{\text{return}}], \Gamma$) ⧺ FillSketchHoles($\kappa[v \,/\, h], \Gamma$)
  else
    $v \leftarrow$ a variable that has semantic type τ in Γ
    return FillSketchHoles($\kappa[v \,/\, h], \Gamma$)

- $\delta \subseteq \mathcal{S} \times \text{BehaviorNodeKinds} \times \mathcal{P}(\text{SemanticGuards}) \times \mathcal{S}$ is the non-deterministic transition relation that indicates which next state to enter after consuming a symbol in the alphabet. A transition $\langle S_1, k, G, S_2\rangle \in \delta$ denotes that when the FSM is currently in S_1, it will transition to the state S_2 if the behavior node β that it consumes satisfies: (1) the kind of β is k; (2) $\bigwedge_{g_i \in G} g_i$ evaluates to true on the corresponding intermediate state of β.

The FSM accepts a behavior sub-path B iff the FSM transitions from the start state S_0 to an accept state $S_f \in \mathcal{F}$ by consuming all behavior nodes in B.

Alg. 2 presents the algorithm for constructing an FSM that represents the behavior of an I/O API, where the semantic guards are predefined for atomic operations and semantic types.

Example 5.11. Given the behavioral specification of the `get` API in Fig. 3a, SPROUT constructs the FSM in Fig. 3b, which accepts behavior sub-paths that are equivalent to the ones emitted by the `get` API. SPROUT first constructs states S_0 and S_1 along with their transitions, incorporating the guard $g_{\text{Permission}}$ based on the atomic operation `f=connect_file(arg0,Read)`. This guard ensures that the file permission includes the `Read` attribute. Subsequently, based on the behavior tuple `n=transfer(f,0,f.size,arg1,0)`, SPROUT constructs state S_2 with transitions from S_1 to S_2 and introduces the following guard conditions: (1) $g_{\text{FdConsistency}}$, which validates file descriptor consistency; (2) g_{Flow}, which specifies the direction of data flow, verifying that data is transferred from a file to a buffer; (3) g_{Offset}, which ensures the file offset equals zero, in accordance with user-specified constraints; (4) g_{Size}, which verifies that the transfer size matches the file size; and (5) g_{Loop}, which ensures that the consumed behavior tuples are emitted from the same loop context. Next, SPROUT constructs state S_3 and its transitions, applying the guard conditions $g_{\text{FdConsistency}}$ and g_{Loop} again, based on the behavior tuple `disconnect(f)`. Finally, SPROUT introduces state S_4 to handle reject transitions, completing the FSM construction.

Definition 5.12 (Semantic Variable Contexts). Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$ and a behavior sub-path B that belongs to the behavior tree of p_1 and is accepted by an FSM for a new API specification $\psi \in \varrho_2$. Let $\langle S_0, k_1, G_1, S_1 \rangle, \ldots, \langle S_{n-1}, k_n, G_n, S_f \rangle$ be the transition path of the FSM to accept B. Let σ_i^{STM} be the semantic type map in the intermediate state σ_i on which $\bigwedge_{g_{i,j} \in G_i} g_{i,j}$ evaluates. The *semantic variable context* Γ is the set of all tuples $\langle v, \tau \rangle$ such that the variable v appears in some $g_{i,j}$ and τ is the corresponding semantic type in σ_i^{STM}, i.e. $\Gamma = \bigcup_{i=1}^{n} \left\{ \langle v, \tau \rangle \in \sigma_i^{\text{STM}} \mid \exists\, g_{i,j} \in G_i,\ v \text{ appears in } g_{i,j} \right\}$.

Definition 5.13 (Minimal Accepting Code Fragments). Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$ and a behavior sub-path B that belongs to the behavior tree of p_1 and is accepted by an FSM for a new API specification $\psi \in \varrho_2$. Let $\beta_1, \ldots \beta_n$ be the sequence of core operation nodes in B, which are matched by the FSM. The *minimal accepting code fragment* $\langle s_1 \ldots s_n \rangle$ is the sequence of statements in the program p_1 such that each core operation node β_i is emitted by the statement s_i.

Example 5.14. For a transition $\langle S_1, k_{\text{transfer}}, G, S_2 \rangle$ in Fig. 3b, SPROUT first checks if $\beta = \boxed{\alpha}$ and $\alpha = v \leftarrow \underline{\texttt{transfer}}(v_1, v_2, v_3, v_4, v_5)$ to verify the behavior node kind. If so, SPROUT then evaluates the semantic guards in G. Recall from Def. 5.10 that when the FSM consumes a behavior node β, the corresponding intermediate state σ is used for evaluating guard conditions. $g_{\text{Size}} \in G$ evaluates to true if the semantic type of the program variable v_3 has the semantic type τ_{FileSize}. To check this property, SPROUT uses the points-to results σ_{PTR} and the semantic type map σ_{STM} in the intermediate state σ. Similarly, $g_{\text{Offset}} \in G$ evaluates to true if the current offset value associated with the file descriptor that v_1 points to equals zero according to the file offset map σ_{FOM}.

5.2.2 Conservative Analyses on Behavior Paths. To ensure that removing a minimum accepting code fragment does not break functionality unrelated to the migration, SPROUT conservatively ensures that all the behavior sub-paths emitted by the code fragment are behaviorally equivalent regardless of the behavior path that the sub-path lies in. To ensure this property, SPROUT uses an over-approximated value-flow graph based on Andersen's pointer analysis [66, 82] to check for value consistency and resource type consistency.

Definition 5.15 (Value Consistency). For a variable v, the value of v is *consistent* at a use site if v holds the same value regardless of the behavior path that passes through the use site. For any statement that emits a behavior node consumed by the FSM, SPROUT performs an intra-procedural

analysis on all behavior paths that reach the statement. SPROUT conducts reachability analysis on the value-flow graph, where the sink is the current use site of v and the sources are the re-definition sites of v within the function. The value consistency of v holds if no sources can reach the sink. A semantic guard is deemed satisfied if the values from the intermediate state satisfy the guard and the values of variables are consistent.

Definition 5.16 (Resource Type Consistency). For a variable v with the semantic type $\tau_{\text{FileDescriptor}}$, the resource type of v is *consistent* at a use site if v is connected to the same type of resource R, regardless of the behavior path that passes the use site. Because resources often outlive the function to which the code fragment belongs, SPROUT conducts reachability analysis on the over-approximated value-flow graph of the entire program, where the sink is the use site of v and the sources are all I/O API calls that connect to resources whose types are not R. The resource type consistency of v holds if no sources can reach the sink. The guard g_{Flow} is deemed satisfied if the variable from an intermediate state σ holds resource type consistency.

LEMMA 5.17. *Given a semantic variable context Γ of a behavior sub-path accepted by an FSM,*

- *A variable v must preserve value consistency intra-procedurally if $\exists\, \tau \in \text{SemanticTypes},\ \langle v, \tau \rangle \in \Gamma$.*
- *If the semantic guard g_{Flow} is satisfied during FSM transitioning, for any variable v such that $\langle v, \tau_{\text{FileDescriptor}} \rangle \in \Gamma$, v must preserve resource type consistency inter-procedurally.*

RATIONALE. The over-approximated value-flow graph encompasses all potential value flows in the program, and no such flow exists from the sources to the sink. □

5.2.3 I/O State Inference of Behavior Paths. SPROUT enumerates all behavior paths B in the behavior tree of p_1 such that B goes through a composite I/O function, which serves as a potential scope for identifying code fragments that may require migration. A naïve brute-force approach to path enumeration scales exponentially as the number of condition nodes increases. To enhance scalability, SPROUT prunes behavior paths whose conditions represent I/O errors according to Def. 4.13.

Pruning paths solely based on I/O errors is insufficient, as a significant portion of path conditions may contain expressions that do not represent I/O errors. SPROUT further deploys another pruning strategy based on the common post-dominator block in the control-flow graph (CFG) of the composite I/O function. Fig. 10 presents an example that applies the pruning strategy. After enumerating a behavior path that goes through the CFG blocks b_0, b_1, and b_p, SPROUT prunes the behavior path B if (1) B goes through the CFG blocks from b_0 to b_p via b_2 and (2) the sub-path of B emitted by the CFG blocks $b_2, \ldots, b_3$ contains no operation nodes.

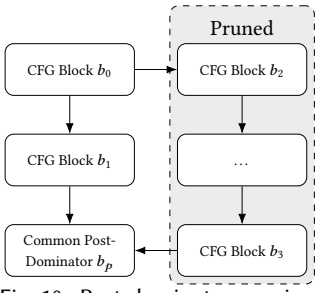


Fig. 10. Post-dominator pruning example.

For each enumerated behavior path, SPROUT computes the intermediate state between each pair of adjacent behavior nodes on the behavior path using the rules outlined in Fig. 7. In particular, SPROUT identifies the semantic types of variables along each behavior path through flow-sensitive pointer analysis. SPROUT then tests the behavior paths against the FSMs for the new API specifications. For a sub-path that is accepted by the FSM for a new API specification $\psi \in \varrho_2$, SPROUT collects the semantic variable context and the minimal accepting code fragment for use in the synthesis stage.

LEMMA 5.18. *Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$. Let the behavior tree of p_1 be η_1. For any new API specification $\psi \in \varrho_2$, if the FSM for ψ accepts some behavior sub-path B from η_1 and the minimal accepting code fragment of B is $\langle s_1 \ldots s_n \rangle$, any behavior sub-path B' from η_1 whose core operation nodes are emitted by $s_1 \ldots s_n$ will also be accepted by the FSM for ψ.*

Proof Sketch. For any behavior sub-path B'' accepted by the FSM, B'' is equivalent to the behavior sub-path emitted by the invocation of ψ. The semantic guards always evaluate to the same results across different behavior paths of B', which follows from Lem. 5.17. □

Lem. 5.18 is important because a code fragment may emit behaviors that vary depending on the incoming program state. Sprout ensures that any minimal accepting code fragment will emit behaviors that are always accepted by the FSM regardless of which behavior path reaches the code.

5.3 Synthesizing the Migrated Program

After identifying a behavior sub-path whose minimal accepting code fragment is equivalent to a new API call during the state inference stage, Sprout synthesizes a migrated program that invokes the new API instead of the old ones.

Alg. 3 presents the algorithm to synthesize a migrated program that invokes the new I/O API. For each behavior sub-path B accepted by the FSM for a new API specification $\psi \in \varrho_2$, Sprout extracts B's minimal accepting code fragment and semantic variable context, then passes them along with ψ to the synthesis algorithm. When the function Synthesize returns true, the original program p_1 modified by the algorithm is the migrated program that Sprout synthesizes.

Synthesize first retrieves a provided sketch for ψ, then completes the sketch (see below) and uses it to replace the original code fragment in the program p_1. This replacement may cause certain original variables to become undefined. Sprout iteratively identifies the original use sites of these deprecated variables intra-procedurally based on over-approximated value flows [84]. Each use site is replaced with another sketch, selected by searching for a provided sketch that performs the same I/O atomic operations as the code to be replaced. Sprout completes these sketches so that they implement the same functionality as the deprecated variables but no longer rely on the deprecated variables. Sprout repeats this process for all deprecated variables until none of them remains.

To complete each sketch, FillSketchHoles uses a dependency-driven approach to recursively fill all the holes in the sketch. Sprout tries to fill each hole with an existing variable that has the same semantic type. If no such variable exists, Sprout creates a fresh variable and prepends a prerequisite sketch that assigns a value to this new variable. This prerequisite sketch is selected by searching for a provided sketch that returns a variable of the same semantic type as the hole. Sprout then recursively completes the prerequisite sketch.

5.4 Soundness

A migration algorithm is considered *sound* if, whenever it produces a migrated program, the migrated program correctly preserves the behavior of the original program. We establish Thm. 5.19, which states that the program migrated by Sprout is correct and emits a behavior tree equivalent to that of the original program, thereby satisfying the definition of soundness.

Theorem 5.19. *Consider a migration problem $\langle \varrho_1, \varrho_2, p_1, Y_n \rangle$, where the program p_2 is synthesized by Alg. 3. We have: (1) p_2 is syntactically correct, (2) p_2 invokes some new API $y \in \mathrm{dom}(\varrho_2)$, and (3) p_2 emits a behavior tree equivalent to that of the original program p_1.*

Proof Sketch. The correctness of the theorem is implied by three key facts: (1) All holes in the sketches are filled with variables with the appropriate semantic types; (2) The behavior sub-paths accepted by the FSM for $\varrho_2(y)$ are equivalent to the behavior sub-paths emitted by an invocation of y, as established in Lem. 5.18; (3) All variables in the minimal accepting code fragment preserve value consistency over all behavior paths according to Lem. 5.17. □

6 Experimental Results

We implemented a Sprout prototype on top of SVF [83], a static value-flow analysis tool based on LLVM [45] and ANTLR [1]. Our newly implemented functionality contains ~25k lines of C++ code.

6.1 Applicability to Real-World Programs and APIs

We acquired 30 benchmark programs from open-source C projects on GitHub. The program sizes range from hundreds to over 100k lines of code (LoC). The benchmark programs perform I/O operations by invoking traditional POSIX APIs.

We developed specifications for the following I/O APIs:

- Over 30 POSIX APIs, such as `open`, `close`, `(p)read`, `(p)write`, `(f)stat`, `lseek`, `fopen`, `fread`, `fclose`, `fseek`, and `ftell`.
- Three types of modern I/O APIs proposed by the operating-system community: userspace I/O APIs for NVM such as `get` and `put` [88, 96], which eliminate file descriptors, zero-copy I/O interfaces on NVM such as `peek` and `patch` [43], and Linux zero-copy I/O APIs such as `mmap` [40], `splice` [42], and `sendfile` [41].

Sprout takes the benchmark programs as input, then replaces the invocations to the POSIX I/O APIs with modern ones to generate migrated programs.

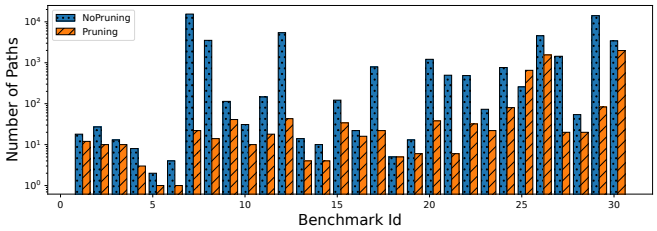
Tab. 1 presents statistics from applying Sprout to these benchmark programs, with a timeout limit of 30 minutes. Each row corresponds to a program. The first two columns (**Id** and **Name**) present the benchmark number, name, and GitHub link from which the program was sourced. The next two present statistics from the program. $\mathbf{P_{loc}}$ presents the program's lines of code, and $\mathbf{P_{IO\text{-}f}}$ presents the total number of I/O-related functions including atomic functions and composite functions in the program. The next three present statistics from the analysis. $\mathbf{A_{a\text{-}f}}$ presents the number of identified aliased functions, $\mathbf{A_{s\text{-}var}}$ presents the number of identified semantic variables in the composite functions analyzed, and $\mathbf{A_{path}}$ presents the number of enumerated paths. "$x \rightarrow y$" denotes that x is obtained without pruning, while y is obtained with pruning enabled. We focus our discussion on the pruned results y in this section and defer the discussion of x to Sec. 6.2. The next two columns present statistics from the synthesized code. $\mathbf{S_{diff}}$ presents the lines of code that differ before and after migration as reported by DiffChecker [15]. $\mathbf{S_{equiv}}$ presents the number of migrated function invocations. It counts only those invocations to atomic functions identified as behaviorally equivalent to the new API invocations. These two columns are obtained with pruning enabled. The last column (**Time**) presents the wall-clock time, in seconds, required to migrate each program. "TO" denotes a timeout. The experiments are conducted on an Ubuntu 22.04 LTS server with two 16-core CPUs of Intel(R) Xeon(R) Gold 6444Y Processor (45M Cache, base clock 3.60 GHz) and 256 GiB of DDR5-4800 ECC RDIMM RAM.

Sprout successfully migrates 29 of the 30 benchmark programs (97%) in the time limit. On average, Sprout synthesizes programs that differ in 32.1 lines after migrating 8.2 function calls. Even for the benchmark program #30 (exim) for which Sprout does not terminate in 30 minutes, Sprout successfully synthesizes a program that differs in 115 lines after migrating 17 function calls. These results indicate that Sprout is effective in migrating real-world I/O programs.

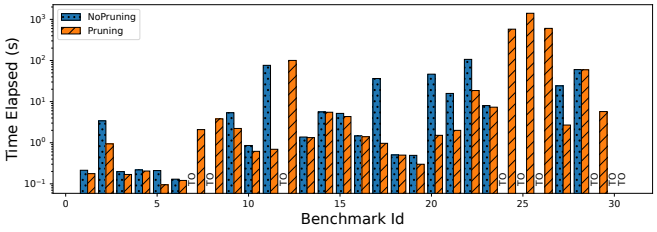
For the 24 smaller programs under 10k LoC, Sprout completes migration in 30.967 seconds on average. Fig. 11c visualizes the scalability of Sprout for the 6 larger programs that exceed 10k LoC. The horizontal axis represents the value-flow graph construction time, which indicates the complexity of the graph. The vertical axis represents the migration time as reported in Tab. 1. Each data point represents a program, with a line connecting the points from left to right. The migration time is positively correlated with the complexity of the value-flow graph. We attribute this phenomenon to the increased overhead associated with performing source-sink reachability

Table 1. Summary statistics for the results of applying SPROUT to benchmark I/O programs.

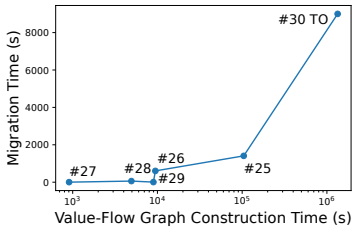
Id	Name	P_{loc}	$P_{IO\text{-}f}$	$A_{a\text{-}f}$	$A_{s\text{-}var}$	A_{path}	S_{diff}	S_{equiv}	Time
1	HTTP-Server	184	14	0	98 → 92	18 → 12	41	4	0.214 → 0.177
2	png-parser	192	4	2	256 → 126	27 → 10	13	4	3.403 → 0.939
3	Html-Parser	215	11	1	84 → 81	13 → 10	16	6	0.198 → 0.169
4	filebench (adapted)	245	9	2	96 → 93	8 → 3	16	8	0.218 → 0.205
5	png_parser	397	7	0	78 → 78	2 → 1	18	6	0.210 → 0.095
6	jread	406	6	0	73 → 73	4 → 1	12	6	0.130 → 0.121
7	HTML-parser	538	9	0	2117 → 93	15475 → 22	20	6	TO → 2.084
8	W3eParser	716	17	0	6301 → 171	3527 → 14	9	3	TO → 3.803
9	C-JSON-Parser	830	12	0	187 → 107	114 → 41	28	7	5.360 → 2.224
10	CsvParserLib	872	16	1	176 → 171	31 → 10	29	4	0.857 → 0.618
11	JUnzip	1074	26	4	912 → 161	146 → 18	17	5	76.260 → 0.693
12	jsonic	1210	8	0	4968 → 434	5419 → 43	24	6	TO → 100.411
13	nxjson	1367	12	0	90 → 85	14 → 4	36	8	1.377 → 1.332
14	pdjson	1427	10	0	74 → 74	10 → 4	22	7	5.605 → 5.515
15	xjson	1501	25	2	103 → 90	122 → 34	33	7	5.169 → 4.335
16	SoftJson	1515	22	1	88 → 88	22 → 16	38	9	1.461 → 1.400
17	chtml	2072	22	1	121 → 77	788 → 22	26	6	36.503 → 0.968
18	LibHTML	2135	15	1	81 → 81	5 → 5	19	6	0.504 → 0.500
19	Smelt	2252	14	1	82 → 82	13 → 6	21	6	0.493 → 0.300
20	json	2270	38	7	74 → 72	1214 → 38	25	12	46.543 → 1.514
21	CSV_Parser	2280	15	0	193 → 77	496 → 6	40	8	15.813 → 2.000
22	zlib (adapted)	2826	27	3	737 → 226	486 → 32	61	15	106.489 → 18.592
23	parson	3761	29	1	94 → 86	73 → 22	22	10	8.007 → 7.324
24	minizip	6049	26	0	767 → 394	765 → 80	25	4	TO → 578.256
25	JPStream	11904	49	1	78 → 193	257 → 653	36	13	TO → 1408.112
26	lodepng	12621	48	3	203 → 275	4561 → 1554	45	12	TO → 604.556
27	cJSON	17249	26	2	253 → 150	1441 → 20	72	18	24.167 → 2.693
28	lua-gumbo	32785	8	3	130 → 123	54 → 20	18	6	60.052 → 59.406
29	flatcc	63532	57	5	1470 → 233	14116 → 83	65	17	TO → 5.724
30	exim	120797	213	16	2045 → 3908	3415 → 1985	115	17	TO → TO



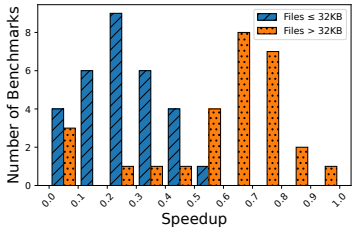
(a) Number of paths enumerated w/ and w/o pruning.



(b) Migration time w/ and w/o pruning.



(c) Scalability for large benchmarks.



(d) Performance gain.

Fig. 11. Experimental results.

checks on the value-flow graph as the complexity of the graph increases. The migration time does not always increase with the program size. When the value-flow graphs are less complex, even the large benchmark programs #27 (cJSON), #28 (lua-gumbo), and #29 (flatcc) can be migrated successfully within a minute. We attribute this efficiency to Sprout's ability to simplify loops and identify aliased functions, which facilitates migration in spite of sophisticated code structures.

6.2 Effectiveness of Path Pruning, Loop Simplification, and Function Merging

Recall that Sprout employs two pruning strategies—path pruning based on semantic variables and post-dominator blocks—to expedite the intra-procedural exploration of paths. To evaluate the effectiveness of pruning, we conducted an ablation study by disabling pruning in Sprout before applying it to the benchmark programs. Fig. 11a and Fig. 11b visualize the number of enumerated paths and the migration time, respectively, for Sprout with and without pruning. Blue bars represent the results with pruning enabled, while orange bars represent the results without pruning. The vertical axes use a logarithmic scale. For most of the programs, the numbers with pruning enabled are less than or equal to those without pruning. The only exceptions are benchmark programs #25 (JPStream), #26 (lodepng), and #30 (exim), where enabling pruning leads to more enumerated paths and/or semantic variables. We attribute these exceptions to pruning enabling Sprout to analyze more functions before timing out. In fact, for two of these programs, pruning is essential for completing the migration within the time limit. On average, pruning leads to 90.9% fewer enumerated paths within the time limit and 74.9% less migration time. These results indicate that the pruning strategies in Sprout are effective in improving migration performance.

Recall that Sprout simplifies loops and merges aliased functions to infer the I/O behavior of functions within a program. To evaluate the impact of these stages, we conducted an ablation study by disabling them in Sprout before applying it to the benchmark programs. The ablated version of Sprout fails to migrate 7 of the 30 programs (23%) since it can no longer identify any behaviorally equivalent I/O API invocations in these programs. The affected ones are #1 (HTTP-Server), #2 (png-parser), #4 (filebench), #20 (json), #28 (lua-gumbo), #29 (flatcc), and #30 (exim). Large programs are affected more heavily—the modified Sprout can no longer migrate 3 of the 6 large benchmark programs (50%) that exceed 10k LoC. These results indicate that loop simplification and aliased-function merging are essential for enabling migration in large programs.

6.3 Correctness and Performance of the Migrated Programs

For each benchmark, we developed a test suite that consists of 10 test cases that process files of varying sizes, ranging from 512 B to 64 MB. We categorize files smaller than 32 KB as small files and those exceeding 32 KB as large files. We then performed experiments by executing both the original programs and their migrated versions using the same test suites under identical environments. We used the Linux zero-copy APIs [40–42] provided by Ubuntu. For file systems that require NVM, we emulated it with DRAM, following prior work [38]. We evaluated the corresponding file systems [43, 96] on a virtual machine.

Fig. 11d presents the performance results of the migrated programs, averaged over 5 runs. The horizontal axis represents the speedup ratio relative to the original versions, while the vertical axis represents the number of benchmark programs. A rectangle drawn between horizontal markers l and r represents the programs for which the migrated versions achieve a speedup ratio within the interval $(l, r]$. All 30 migrated programs gain performance speedup for small files and 28 programs gain speedup for large files. The speedup for small files results mainly from removing file descriptors and reducing the context-switching overhead between user space and kernel space [96], and the speedup for large files results from zero-copy I/O [43]. The average speedup for small files is 24.9% (max 56.1%) and the average speedup for large files is 51.6% (max 90.1%). Two migrated

benchmark programs, #22 (zlib) and #24 (minizip), show no speedup when processing large files, so their results are omitted from the plot. We attribute it to some modern APIs, such as `get` and `put`, only improving performance when input files are below a certain size [88], since the data copy overhead dominates. These results highlight the potential to maximize performance benefits of modern storage systems through Sprout's automated migration.

6.4 Generality to Other Application Domains

To enable automated migration, Sprout leverages domain-specific insights such as semantic types, commonly aliased functions, and bounded loops that invoke APIs. To evaluate how well our approach generalizes to other domains, we conducted a case study that adapts Sprout to database access APIs (DB APIs) in the MySQL C Library [55]. Automated migration of DB APIs is important for purposes such as upgrading deprecated APIs to benefit from technological advancements [67, 97] and batching multiple queries to enable performance optimizations [8, 18, 71]. However, automated migration of DB APIs is nontrivial because it often requires stateful and many-to-many API mappings that are intermingled with the structured control flow in non-contiguous code. These challenges are similar to those encountered when migrating I/O APIs.

Effort Required for Adaptation. We made the following lightweight modifications to Sprout: (1) Customize the specification language and semantic types to characterize DB APIs. Furthermore, we incorporate the customized semantic types into the semantic pruning strategies to gain performance benefits, which is optional. (2) Incorporate the API state transitions and semantic guards into the FSM implementation. (3) Customize the sketches according to the new DB APIs.

These customizations were completed by our lead author in less than two weeks. For context, the full Sprout prototype took approximately 10 person-months to implement. These results indicate that customizing Sprout to work with a different application domain is relatively straightforward.

After these lightweight customizations, the remainder of Sprout remains unchanged. The unchanged core components include (1) the function alias detection algorithm, (2) the path enumeration algorithm, which includes the enhanced flow-sensitive pointer analysis for updating semantic variable contexts, (3) the typestate analysis that builds on FSM transitions accepting behaviors, and (4) the dependency-driven synthesis algorithm, as outlined in Fig. 1. Loop simplification is unnecessary for the DB APIs and therefore disabled. These core components address the key challenges in determining whether a set of multiple API invocations in non-contiguous code can be migrated and, if so, how to synthesize the migrated code. These algorithms are also applicable to other application domains beyond I/O and databases, provided that the aforementioned lightweight customizations are implemented.

$$
\begin{aligned}
\psi &::= \underline{\texttt{spec}}\ y(v_1 : \tau_1, \ldots, v_n : \tau_n) : \tau_{\text{return}}\ \{\chi\} \\
\chi &::= \varepsilon \mid \alpha \mid \alpha; \chi \\
\alpha &::= v_{\text{buffer}} \leftarrow \underline{\texttt{alloc}}(v_{\text{size}}) && \text{(Allocate)} \\
&\mid v_{\text{handle}} \leftarrow \underline{\texttt{init\_handle}}() && \text{(Init Handle)} \\
&\mid v_{\text{handle}} \leftarrow \underline{\texttt{connect}}(v_{\text{handle}}) && \text{(Connect Database)} \\
&\mid \underline{\texttt{disconnect}}(v_{\text{handle}}) && \text{(Disconnect)} \\
&\mid v_{\text{error\_flag}} \leftarrow \underline{\texttt{query}}(v_{\text{handle}}, v_{\text{stmt}}) && \text{(Query)} \\
&\mid v_{\text{result}} \leftarrow \underline{\texttt{get\_result}}(v_{\text{handle}}) && \text{(Get Result)} \\
&\mid \underline{\texttt{return}}\ \theta && \text{(Return)}
\end{aligned}
$$

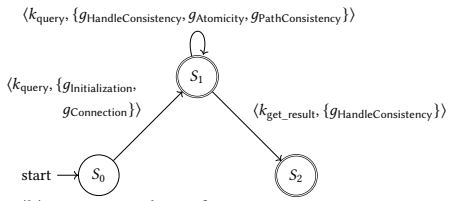
$$
\begin{aligned}
&\psi \in \text{BehavioralSpecifications} \quad y \in \text{FunctionNames} \\
&\tau_1, \ldots, \tau_n, \tau_{\text{return}} \in \text{SemanticTypes} \\
&\chi \in \text{OperationSequences} \quad \alpha \in \text{AtomicOperations} \\
&v, v_1, \ldots, v_n, v_{\text{buffer}}, v_{\text{handle}}, \\
&v_{\text{result}}, v_{\text{stmt}}, v_{\text{error\_flag}} \in \text{SpecVars}
\end{aligned}
$$

Fig. 12. Abstract syntax of behavioral specifications for database access API. Note that the semantic types in ψ have been adapted to the database domain.

Customized Specification Language and Semantic Types. We modified Sprout's specification language, originally defined for I/O APIs in Fig. 5, to incorporate new atomic operations and semantic types that work with DB APIs as in Fig. 12. The new atomic operations initialize a database

```
int mysql_real_query(MYSQL *mysql, const char
  *stmt_str, unsigned long length) {
ParamType:{arg0:Handle, arg1:QueryStmt, arg2:
  StmtLength}
Semantics:{query(arg0, arg1)}
Return: r0:ErrorFlag
}
Sketch: {
Sketch1: char queryStmt[MAX_QUERY_LENGTH];
...
Sketch2: sprintf(#0, "%s\n;", #1);
...
Sketch3: #2 = mysql_real_query(#3, #4, #5);
...
Sketch4: do {
MYSQL_RES* new_res = mysql_store_result(#6);
  if (#7) {
    mysql_free_result(#7);
  }
  #7 = new_res;
} while (mysql_next_result(#6) == 0);
}
```

(a) Specification and sketches for `mysql_real_query`.



(b) State machine for merging DB queries.

```
fd = open(filename, O_RDONLY);
fstat(fd, &st);
int file_size = st.st_size;
bool cond = func1();
if (cond) {
    file_size = func2();
}
...
read(fd, buf, file_size);
close(fd);
```

(c) Code not migrated.

Fig. 13. Example specification, state machine, and code snippet.

Table 2. Summary statistics for the results of applying Sprout to benchmark database programs.

Id	Name	P_{loc}	$P_{DB\text{-}f}$	$A_{a\text{-}f}$	$A_{s\text{-}var}$	A_{path}	S_{diff}	S_{equiv}	Time
1	dart-HOCH	130	5	0	10 → 7	8 → 5	12	3	0.629 → 0.638
2	EducationalMaterial	201	15	0	34 → 22	17 → 7	18	4	0.453 → 0.238
3	irix (db-module)	601	36	1	117 → 63	73 → 15	44	15	3.207 → 0.548
4	smlgr	818	19	0	67 → 57	6250 → 22	16	3	TO → 5.686
5	RoadApplePi	972	22	0	368 → 180	959 → 307	42	10	TO → 34.898
6	redhat (db-module)	1092	16	8	60 → 40	27 → 18	30	10	9.177 → 5.095
7	Forum-system	1182	18	0	13 → 12	11 → 9	34	10	13.417 → 13.373
8	openrail	2198	37	2	802 → 70	915 → 36	22	5	TO → 49.987
9	chatroom_lt	2507	40	0	492 → 256	164 → 34	96	42	23.159 → 12.317
10	C-Blog	2546	27	1	128 → 88	36 → 17	38	16	3.311 → 2.583

handle (Init Handle), establish a connection to a database instance (Connect Database), perform a query on a connected database instance (Query), and retrieve query results from the database instance (Get Result). New semantic types include Handle, Result, ErrorFlag, and QueryStmt. We used this formulation to specify the semantics of over 30 DB APIs provided by the MySQL C Library [55], including commonly used APIs `mysql_query`, `mysql_create_db`, and `mysql_select_db`.

Example Migration Problem. Consider the problem of upgrading programs that invoke the 5.x version of the MySQL C library to its 8.x version. Many APIs are deprecated and should be replaced, such as `mysql_connect`, `mysql_create_db`, and `mysql_drop_db` [54]. In addition, to take advantage of the new API `mysql_real_query` that can perform multiple queries at once to improve performance [8, 18, 71], we also aim to identify groups of multiple individual invocations to the old API `mysql_query` and merge them into a single invocation to `mysql_real_query`. Achieving this migration is nontrivial, since it requires replacing multiple related stateful API invocations with synthesized code that is behaviorally equivalent.

For instance, Fig. 13a presents the specification for `mysql_real_query`, and Fig. 13b presents the FSM for replacing multiple `mysql_query` invocations with a single merged `mysql_real_query` invocation. Each invocation to `mysql_query` emits a behavior node $\beta = \boxed{\alpha}$, where $\alpha = v_{\text{error\_flag}} \leftarrow$

$\overline{\texttt{query}}(v_{\text{handle}}, v_{\text{stmt}})$. To replace them, all guard conditions $g \in G_1 \cup G_2$ on the transitions $\langle S_0, k_{\text{query}}, \overline{G_1, S_1}\rangle$ and $\langle S_1, k_{\text{query}}, G_2, S_1\rangle$ in Fig. 13b must hold. The semantic guards in the transition includes:

- The database handle is initialized in the program ($g_{\text{Initialization}}$).
- The database handle is connected to a database instance ($g_{\text{Connection}}$).
- All relevant behavior nodes β_{query} are associated with the same database handle ($g_{\text{HandleConsistency}}$).
- Either all or none of such behavior tuples are emitted ($g_{\text{Atomicity}}$).
- All feasible program paths between the statements of previous and current behavior tuples do not emit any behavior ($g_{\text{PathConsistency}}$).

Experimental Results for Real-World Programs. We acquired 10 real-world benchmark programs from open-source C/C++ projects on GitHub that invoke the MySQL C Library APIs, with code sizes ranging from hundreds to thousands of lines. Tab. 2 presents statistics from applying Sprout to these benchmark programs, with a timeout limit of 30 minutes. Most columns are inherited from Tab. 1, with the exception of the fourth column ($\mathbf{P_{DB\text{-}f}}$), which presents the total number of database-related functions including atomic functions and composite functions in the program. Sprout successfully migrates all 10 benchmark programs (100%) in the time limit. On average, Sprout synthesizes programs that differ in 35.2 lines after migrating 11.8 function calls.

As in Sec. 6.3, we executed the original and migrated programs using the same test suites under identical environments. All migrated programs compile successfully, and when executed, produce outputs identical to the original. The migrated programs that perform merged queries exhibit an average speedup of 52.3%. These results highlight the effectiveness, efficiency, and generality of Sprout's approach in migrating real-world programs across important domains beyond I/O.

7 Limitations

The limitations of Sprout arise mainly from the requirement to ensure soundness of migration.

Missed Migration Opportunities. While Sprout ensures that all migrated programs are correct, it is conservative and does not guarantee that all invocations to the old APIs are migrated. This limitation arises from a fundamental challenge in static analysis. For example, to determine whether the code in Fig. 13c can be migrated, Sprout analyzes the potential values of `file_size` on Line 9. This analysis occurs when the FSM in Fig. 3b is in state S_1 and the semantic guard g_{Size} is being checked. This guard requires that `file_size` represents only the size of the file connected by `fd`. However, the over-approximated value-flow graph indicates that `file_size` can also hold the value generated on Line 6. This value is not known to be the size of the file connected by `fd`, which violates g_{Size}. Thus, Sprout does not migrate this case. If variable `cond` can be either true or false, then this case is indeed not migratable, and Sprout correctly refuses to migrate it. If `cond` is in fact always false, then this case represents a missed migration opportunity. In practice, however, this issue arises infrequently in our experiments. We attribute it to developers often following best practices to ensure variables keep their intended meanings.

Manual Specification Effort. Sprout requires experts to specify API semantics, which can sometimes be challenging. We anticipate that this cost is relatively small. Firstly, it is a one-time task performed by experts, for example experts who invented the storage systems that provide those APIs. Once specified, the expert API knowledge can be applied across a wide range of software applications that invoke such APIs or migrate towards those APIs, offering substantial automation benefits to users and developers at scale. Secondly, in our experiments, each API specification took our lead author less than 15 minutes on average to build from scratch, where most time was spent on understanding natural language documentation and seeking clarifications with operating system experts. The writing of the specification itself took under 5 minutes.

Manual Customization Effort. Although our implementation provides open interfaces in its framework, adapting Sprout to a different application domain requires developers to manually formalize API specifications and implement the migration rules for the new domain. Nevertheless, we anticipate that the cost of such customizations is small compared to the core components of our approach (see Sec. 6.4) and that the benefits of automated migration are substantial.

Compatibility with LLVM IR. Our current Sprout implementation performs static analysis on LLVM IR, so it works only with languages compatible with LLVM IR. However, our approach is adaptable to other languages with similar frameworks for pointer analysis and value-flow analysis.

8 Related Work

Rule-based Transformation Synthesis for Upgrading Dependencies. The most closely related line of research synthesizes semantic patches from user-provided examples for repetitive edits in the Linux kernel [46, 77]. Several other techniques synthesize rules for textual modifications in the source code [20, 23, 35, 39, 53, 59, 60, 90], often by summarizing transformation rules from user-provided example edits or mined patches in similar codebases. More broadly, there is a large body of research on updating downstream programs when library dependencies are upgraded [12, 14, 16, 22, 56, 59, 87] that mainly focus on editing one API invocation at a time and rely on a perfect match between the semantics of different versions of the same API. In contrast, Sprout (1) automatically synthesizes previously unknown transformations to reduce human effort, (2) does not require existing examples of similar and successfully migrated programs, (3) enables many-to-many, many-to-one, one-to-many, and one-to-one API migrations, and (4) guarantees behavioral equivalence between the original and migrated programs through static analysis.

Replacing Data Structures through Equivalence Checking. Some techniques synthesize alternative implementations for an existing program [33, 65, 76, 78, 80, 86, 91], among which the most closely related ones synthesize programs for replacement data structures [65, 76, 91]. Most of these techniques rely on verifying program equivalence [11, 13, 92–94], often limited to relatively simple data structures and programs. These techniques are not directly applicable to migrating I/O APIs that may perform sophisticated operations on system resources. In contrast, Sprout is capable of migrating such sophisticated APIs, and it does so by modeling the I/O API invocations with specifications and ensuring correctness through typestate analysis and pointer analysis.

Neural Approaches for Program Synthesis. Program synthesis using machine learning and LLM-based neural approaches is an active research area [2, 6, 19, 36, 37, 57, 58, 95]. These techniques are general and scalable but are often limited in correctness due to their predictive nature [5], especially with large codebases. Challenges include reasoning with long contexts [44, 47] and a tendency to hallucinate [34]. In contrast, Sprout ensures formal correctness through static analysis.

Other Applications of Static Analysis. Static analysis techniques, such as typestate analysis, are predominantly designed for bug detection [6, 7, 21, 48, 69, 79, 84]. ReBA [17] uses static analysis for software evolution, specifically, it uses flow-insensitive pointer analysis to accommodate different library versions by generating a compatibility layer. In contrast, Sprout directly modifies a program to generate an efficient migrated version, and it ensures soundness with a flow-, field-sensitive pointer analysis and Andersen's pointer analysis [66].

9 Conclusion

We propose a novel automated technique for migrating software to new I/O APIs, focusing on the preservation of program semantics through behavioral equivalence. Experimental results with real-world C programs highlight the effectiveness, efficiency, and generality of our approach.

Acknowledgments

We thank the anonymous reviewers for their insightful and helpful comments. Part of this work was conducted while the author Zhaoyang Zhang was at the University of Science and Technology of China. This work was supported in part by Hong Kong Research Grants Council (Project No. 26216025), Alibaba Group (through the Alibaba Innovative Research Program), and HKUST-WeBank Joint Lab (Project No. WEB24EG01-G). The views expressed in this work are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] ANTLR Project. 2024. ANTLR - Another Tool for Language Recognition. https://www.antlr.org/ [Online; accessed 8-November-2024].

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL] https://arxiv.org/abs/2108.07732

[3] Roberto Bez and Agostino Pirovano. 2004. Non-volatile memory technologies: emerging concepts and new materials. *Materials Science in Semiconductor Processing* 7, 4 (2004), 349–355. doi:10.1016/j.mssp.2004.09.127 Papers presented at the E-MRS 2004 Spring Meeting Symposium C: New Materials in Future Silicon Technology.

[4] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 38:1–38:30. doi:10.4230/LIPIcs.ECOOP.2023.38

[5] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2025. Verified code transpilation with LLMs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1310, 31 pages.

[6] Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gfOiAqYm

[7] Xiao Cheng, Jiawei Ren, and Yulei Sui. 2024. Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi-Point Slicing. *Proc. ACM Softw. Eng.* 1, FSE, Article 23 (July 2024), 23 pages. doi:10.1145/3643749

[8] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/2491956.2462180

[9] Jonathan Corbet. 2017. Zero-copy networking [LWN.net]. https://lwn.net/Articles/726917/ Accessed: 2025-07-15.

[10] Jonathan Corbet. 2018. Zero-copy TCP receive [LWN.net]. https://lwn.net/Articles/752188/ Accessed: 2025-07-15.

[11] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 183–190. doi:10.1007/978-3-319-96145-3_10

[12] Yaniv David, Xudong Sun, Raphael J. Sofaer, Aditya Senthilnathan, Junfeng Yang, Zhiqiang Zuo, Guoqing Harry Xu, Jason Nieh, and Ronghui Gu. 2022. UPGRADVISOR: Early Adopting Dependency Updates Using Hybrid Program Analysis and Hardware Tracing. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 751–767. https://www.usenix.org/conference/osdi22/presentation/david

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[14] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2187–2200. doi:10.1145/3133956.3134059

[15] DiffChecker. 2025. DiffChecker. https://www.diffchecker.com

[16] Danny Dig and Ralph Johnson. 2006. Automated upgrading of component-based applications. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 675–676. doi:10.1145/1176617.1176668

[17] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. 2008. ReBA: refactoring-aware binary adaptation of evolving libraries. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 441–450. doi:10.1145/1368088.1368148

[18] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 169–182. doi:10.14778/3282495.3282502

[19] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems* 36 (2023), 46701–46723.

[20] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 204–215. doi:10.1145/3293882.3330571

[21] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 576–587. doi:10.1145/2635868.2635869

[22] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 791–796. doi:10.1145/3236024.3275535

[23] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: output-oriented program synthesis for combating breaking changes in libraries. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 161 (Oct. 2021), 27 pages. doi:10.1145/3485538

[24] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 12 (Oct. 2014), 44 pages. doi:10.1145/2629609

[25] GOV.UK. 2021. Organising for digital delivery - Report from the Digital Economy Council, 9 September 2020. https://www.gov.uk/government/publications/organising-for-digital-delivery/organising-for-digital-delivery.

[26] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 135–148. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han

[27] Philip Hazel. 2003. *The Exim SMTP mail server: official guide for release 4.* UIT Cambridge, PO Box 145, Cambridge CB4 1GQ, England.

[28] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. 2018. PASTE: a network programming interface for non-volatile main memory. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 17–33. https://www.usenix.org/conference/nsdi18/presentation/honda

[29] IEEE and The Open Group. 2004. POSIX API open Description. https://pubs.opengroup.org/onlinepubs/007904875/functions/open.html. Accessed: 2025-07-15.

[30] IEEE and The Open Group. 2018. The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008). https://pubs.opengroup.org/onlinepubs/9699919799/

[31] Anna Irrera. 2017. Banks scramble to fix old systems as it 'Cowboys' ride into Sunset. https://www.reuters.com/article/us-usa-banks-cobol-idUSKBN17C0D8

[32] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC] https://arxiv.org/abs/1903.05714

[33] 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* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. doi:10.1145/1806799.1806833

[34] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12, Article 248 (March 2023), 38 pages. doi:10.1145/3571730

[35] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2020. Inferring Program Transformations from Singular Examples via Big Code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 255–266. doi:10.1109/ASE.2019.00033

[36] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2025. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:10.1145/3747588 Just Accepted.

[37] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] https://arxiv.org/abs/2310.06770

[38] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 494–508. doi:10.1145/3341301.3359631

[39] Stephen Kell. 2010. Component adaptation and assembly using interface relations. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 322–340. doi:10.1145/1869459.1869487

[40] Michael Kerrisk. 2025. Mmap [man7.org]. https://man7.org/linux/man-pages/man2/mmap.2.html Accessed: 2025-07-15.

[41] Michael Kerrisk. 2025. Sendfile [man7.org]. https://man7.org/linux/man-pages/man2/sendfile.2.html Accessed: 2025-07-15.

[42] Michael Kerrisk. 2025. Splice [man7.org]. https://man7.org/linux/man-pages/man2/splice.2.html Accessed: 2025-07-15.

[43] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. 2020. SubZero: zero-copy IO for persistent main memory file systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3409963.3410489

[44] Yury Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. 2024. Babilong: Testing the limits of llms with long context reasoning-in-a-haystack. *Advances in Neural Information Processing Systems* 37 (2024), 106519–106554.

[45] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. doi:10.1109/CGO.2004.1281665

[46] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 601–613. https://www.usenix.org/conference/atc18/presentation/lawall

[47] Shangyu Li, Juyong Jiang, Tiancheng Zhao, and Jiasi Shen. 2025. OSVBench: Benchmarking LLMs on Specification Generation Tasks for Operating System Verification. arXiv:2504.20964 [cs.CL] https://arxiv.org/abs/2504.20964

[48] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 859–872. doi:10.1145/3503222.3507770

[49] Jean loup Gailly and Mark Adler. 1995–2023. zlib. https://github.com/madler/zlib. Accessed: 2025-07-15.

[50] Dinesh Maheshwari. 2014. Memory and system architecture for 400Gb/s networking and beyond. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 116–117.

[51] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 175–186. doi:10.1145/2619239.2626311

[52] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413. doi:10.1145/3341301.3359657

[53] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic editing: generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 329–342. doi:10.1145/1993498.1993537

[54] MySQL. 2023. MySQL C API Function Deprecation Descriptions. https://dev.mysql.com/doc/c-api/8.0/en/c-api-basic-function-reference.html. Accessed: 2025-07-15.

[55] MySQL. 2023. MySQL C API Function Descriptions. https://dev.mysql.com/doc/c-api/8.0/en/c-api-function-descriptions.html. Accessed: 2025-07-15.

[56] Nacho Navarro, Salwa Alamir, Petr Babkin, and Sameena Shah. 2023. An Automated Code Update Tool For Python Packages. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 536–540. doi:10.1109/ICSME58846.2023.00068

[57] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 457–468. doi:10.1145/2642937.2643010

[58] Ansong Ni, Daniel Ramos, Aidan Z.H. Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2021. SOAR: A Synthesis Approach for Data Science API Refactoring. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 112–124. doi:10.1109/ICSE43902.2021.00023

[59] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 74–85. doi:10.1109/ICSE43902.2021.00020

[60] Marius Nita and David Notkin. 2010. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 205–214. doi:10.1145/1806799.1806832

[61] CT Department of Labor Communications Office Commissioner Kurt Westby. 2020. DOL offers guidance to employees to help speed up their claim approval. https://www.ctdol.state.ct.us/communic/newsrels/LABOR%20DEPARTMENT%20PROVIDES%20UPDATES%20TO%20EMPLOYEES,%20EMPLOYERS%20FOLLOWING%20RECORD%20NUMBER%20OF%20UNEMPLOYMENT%20BENEFIT%20APPLICATIONS.pdf

[62] State of New Jersey Governor Phil Murphy. 2020. TRANSCRIPT: April 4th, 2020 Coronavirus Briefing Media. https://nj.gov/governor/news/news/562020/approved/20200404b.shtml

[63] State of New Jersey Governor Phil Murphy. 2020. TRANSCRIPT: April 8th, 2020 Coronavirus Briefing Media. https://nj.gov/governor/news/news/562020/approved/20200408c.shtml

[64] U.S. Government Accountability Office. 2019. Information technology: Agencies need to develop modernization plans for Critical Legacy Systems. https://www.gao.gov/products/gao-19-471

[65] Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. 2024. Semantic Code Refactoring for Abstract Data Types. *Proc. ACM Program. Lang.* 8, POPL, Article 28 (Jan. 2024), 32 pages. doi:10.1145/3632870

[66] Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave Propagation and Deep Propagation for Pointer Analysis. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, USA, 126–135. doi:10.1109/CGO.2009.9

[67] Jeff H. Perkins. 2005. Automatically generating refactorings to support API evolution. *SIGSOFT Softw. Eng. Notes* 31, 1 (Sept. 2005), 111–114. doi:10.1145/1108768.1108818

[68] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: the operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[69] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Kaelbling. 2018. Selecting Representative Examples for Program Synthesis. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 4161–4170. https://proceedings.mlr.press/v80/pu18b.html

[70] Rick Rabinovich. 2013. 40Gb/s & 100Gb/s ethernet long-reach host board channel design. *IEEE Communications Magazine* 51, 11 (2013), 152–158.

[71] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.

[72] Digital Reality. 2018. All-Change in Financial Services as Incumbents Meet Startups. https://www.digitalrealty.com/resources/white-papers/all-change-in-financial-services-as-incumbents-meet-startups

[73] Dennis M. Ritchie and Ken Thompson. 1974. The UNIX time-sharing system. *Commun. ACM* 17, 7 (July 1974), 365–375. doi:10.1145/361011.361061

[74] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 9.

[75] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 404–415. doi:10.1109/ICSE.2017.44

[76] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. 2019. Synthesizing replacement classes. *Proc. ACM Program. Lang.* 4, POPL, Article 52 (Dec. 2019), 33 pages. doi:10.1145/3371120

[77] Lucas Serrano, Van-Anh Nguyen, Ferdian Thung, Lingxiao Jiang, David Lo, Julia Lawall, and Gilles Muller. 2020. SPINFER: inferring semantic patches for the Linux kernel. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 16, 14 pages. https://www.usenix.org/conference/atc20/presentation/serrano

[78] 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* (Phoenix, AZ,

USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 269–285. doi:10.1145/3314221.3314591

[79] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706. doi:10.1145/3192366.3192418

[80] Armando Solar-Lezama. 2008. *Program Synthesis By Sketching*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html

[81] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (Jan. 1986), 157–171. doi:10.1109/TSE.1986.6312929

[82] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 460–473. doi:10.1145/2950290.2950296

[83] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[84] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 254–264. doi:10.1145/2338965.2336784

[85] A.A. Terekhov and C. Verhoef. 2000. The realities of language conversions. *IEEE Software* 17, 6 (2000), 111–124. doi:10.1109/52.895180

[86] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C. Rinard. 2021. Supply-Chain Vulnerability Elimination via Active Learning and Regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1755–1770. doi:10.1145/3460120.3484736

[87] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A. Gerosa, and Igor Scaliante Wiese. 2023. I Depended on You and You Broke Me: An Empirical Study of Manifesting Breaking Changes in Client Packages. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 94 (May 2023), 26 pages. doi:10.1145/3576037

[88] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. doi:10.1145/2592798.2592810

[89] Bo Wang, Aashish Kolluri, Ivica Nikolić, Teodora Baluta, and Prateek Saxena. 2023. User-Customizable Transpilation of Scripting Languages. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 82 (April 2023), 29 pages. doi:10.1145/3586034

[90] Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. 2016. Transforming Programs between APIs with Many-to-Many Mappings. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:26. doi:10.4230/LIPIcs.ECOOP.2016.25

[91] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. 2022. Complexity-guided container replacement synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 68 (April 2022), 31 pages. doi:10.1145/3527312

[92] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2017. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.* 2, POPL, Article 56 (Dec. 2017), 29 pages. doi:10.1145/3158144

[93] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 286–300. doi:10.1145/3314221.3314588

[94] Tim Wood, Sophia Drossopolou, Shuvendu K. Lahiri, and Susan Eisenbach. 2017. Modular Verification of Procedure Equivalence in the Presence of Memory Allocation. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 937–963. doi:10.1007/978-3-662-54434-1_35

[95] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. arXiv:2212.09420 [cs.SE] https://arxiv.org/abs/2212.09420

[96] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. 2023. Enabling High-Performance and Secure Userspace NVM File Systems with the Trio Architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 150–165. doi:10.1145/3600006.3613171

[97] Jing Zhou and Robert J. Walker. 2016. API deprecation: a retrospective analysis and detection method for code examples on the web. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 266–277. doi:10.1145/2950290.2950298

Received 2025-03-26; accepted 2025-08-12