Association for Computing Machinery

Advancing Computing as a Science & Profession

June 25, 2024
Copenhagen, Denmark

SOAP '24

Proceedings of the 13th ACM SIGPLAN International Workshop on

the State Of the Art in Program Analysis

Edited by:
Raphaël Monat and Cindy Rubio-González

Sponsored by:
ACM SIGPLAN

Co-located with:
PLDI '24

Association for Computing Machinery, Inc.
1601 Broadway, 10th Floor
New York, NY 10019-7434
USA

ACM ISBN: 979-8-4007-0621-9

Cover photo:
Title: “Cirkelbroen”
Photographer: Thomas Rousing
Licensed by Copenhagen Media Center. All rights reserved.
Cropped from original:
https://platform.crowdriff.com/m/visitcopenhagen/folder/16190/album/76822?a=3251-wl-VFRTLKU9R1SIO8M7B9CRJIDBVS

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Welcome from the Chairs

The 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'24) is co-located with the 45th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'24). In line with past workshops, SOAP'24 aims to bring together members of the program analysis community to share new developments and shape innovations in program analysis.

This edition of SOAP had eleven submissions, each reviewed by three reviewers. Six were accepted (54% acceptance rate), and three others were conditionally accepted, of which two where successfully shepherded in the end.

Along with the accepted papers, SOAP'24 features three keynotes by leading members of the program analysis community: Eva Darulova (Uppsala University), Anders Møller (Aarhus Universyt) and Manu Sridharan (University of California Berkeley).

We would like to commend the efforts of the eleven members of the program committee, who donated their valuable time and effort to make the reviewing process possible. We also thank the PLDI chairs, the workshops chairs and the ACM staff for their continued support in making this workshop possible. We hope you enjoy SOAP24 and look forward to enlightening discussions.

June 2024

Cindy Rubio González
Raphaël Monat

Program Chairs

Raphaël Monat (Inria and University of Lille, France)
Cindy Rubio González (University of California at Davis, USA)

Program Committee

Alexandra Bugariu (Max Planck Institute for Software Systems, Germany)
Ben Greenman (University of Utah, USA)
Michaël Marcozzi (CEA, LIST, Université Paris Saclay, France)
Jorge A. Navas (Certora, USA)
Luca Negrini (Università Ca' Foscari, Italy)
Martin Schäf (Amazon Web Services, USA)
Natarajan Shankar (SRI International, USA)
Quentin Stiévenart (Université du Québec à Montréal, Canada)
Tucker Taft (AdaCore, USA)
Vesal Vojdani (University of Tartu, Estonia)
Wenwen Wang (University of Georgia, USA)

Steering Committee

Pietro Ferrara (Università Ca' Foscari, Italy)
Liana Hadarean (Amazon, USA)
Laure Gonnord (Grenoble-INP/LCIS, France)
Neville Grech (University of Malta, Malta)
Ben Hermann (Technische Universität Dortmund, Germany)
Padmanabhan Krishnan (Oracle Labs, Australia)
Thierry Lavoie (Synopsys, Canada)
Lisa Nguyen Quang Do (Google, Switzerland)
Christoph Reichenbach (Lund University, Sweden)
Laura Titolo (NIA/NASA LaRC, USA)
Omer Tripp (Amazon, USA)
Caterina Urban (INRIA and École Normale Supérieure, France)

Contents

Frontmatter

Papers

Dr Wenowdis: Specializing Dynamic Language C Extensions using Type Information

Maxwell Bernstein
acm@bernsteinbear.com
Northeastern University
Boston, Massachusetts, USA

Carl Friedrich Bolz-Tereick*
cfbolz@gmx.de
Heinrich-Heine-Universität
Düsseldorf, Germany

Abstract

C-based interpreters such as CPython make extensive use of C "extension" code, which is opaque to static analysis tools and faster runtimes with JIT compilers, such as PyPy. Not only are the extensions opaque, but the interface between the dynamic language types and the C types can introduce impedance. We hypothesise that frequent calls to C extension code introduce significant overhead that is often unnecessary.

We validate this hypothesis by introducing a simple technique, "typed methods", which allow selected C extension functions to have additional metadata attached to them in a backward-compatible way. This additional metadata makes it much easier for a JIT compiler (and as we show, even an interpreter!) to significantly reduce the call and return overhead.

Although we have prototyped typed methods in PyPy, we suspect that the same technique is applicable to a wider variety of language runtimes and that the information can also be consumed by static analysis tooling.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Runtime environments**.

Keywords: Just-in-time compilers, JIT, C extensions, PyPy, CPython, Python, type information

ACM Reference Format:
Maxwell Bernstein and Carl Friedrich Bolz-Tereick. 2024. Dr Wenowdis: Specializing Dynamic Language C Extensions using Type Information. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3652588.3663316

*Author's preferred name is CF, but publishing tools require previous name.

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663316

1 Introduction

One of the reasons for the success of dynamic languages such as Python and Ruby is the ease with which they can interface to existing C libraries through the use of C-implemented extension modules. Another common reason for writing C extensions is to improve the performance when the dynamic language runtime isn't fast enough in a hotspot. The downside of C extensions is that they cannot easily be analyzed by static analysis tools together with the Python or Ruby code that is calling into the library. The same problem plagues more advanced dynamic language implementations with a JIT compiler because a call into a C extension represents an optimization barrier [14]. For example, objects that may otherwise be unboxed by the JIT [5] now require boxing for consumption by the C extension, only to often be immediately unboxed again by the called C code.

As a motivating example, Listing 1 shows a complete minimal C extension module definition. The `inc` function takes and returns a Python `int` object. It unboxes its argument, increments it, and re-boxes the result. This type information is not available to the Python runtime; it's implicit in the C argument processing wrapper code. Calling this function from an optimizing Python implementation such as PyPy is thus very costly since it requires a generic call path and the allocation of C data structures that behave like the C extension expects them to.

In this paper we propose Dr Wenowdis [29], a very lightweight mechanism to expose some amount of type and effect knowledge about the functions a C extension module implements. In our prototype, we convert C function signatures to type annotations manually. We carry out this work specifically in the context of the CPython C API. We want to make it possible to incrementally add this knowledge to existing libraries without having to do an invasive rewrite or introduce a new dependency. We use the exposed information for improving the performance of Python→C calls using the PyPy JIT compiler [6]. The same type information can also be used for type checking (e.g. in MyPy [25]) or static analysis.

For the example, we can add the type information that the function takes and returns a C `long` by writing the code in Listing 2. This annotation is enough to speed up calling the `inc` function in PyPy by about 60 times, because it can call the `inc_impl` function directly, and optimize away the argument

checking and unboxing. The example will be discussed more thoroughly in Section 2.2 and Section 3.

We present an early prototype that requires manual annotations with very limited expressiveness but a more complete version could generate the annotations of most extension functions automatically using a binding generator such as Cython [4] or PyO3 [17].

2 Background

2.1 CPython

CPython is the reference implementation of Python. Older versions implement the Python language by compiling it into a simple stack-based bytecode and running that in a straightforward interpreter [3, 32]. More recent versions of CPython (from 3.11 onwards) use bytecode quickening [8–11] to speed up bytecode execution [19]. The upcoming 3.13 release is probably also going to contain a simple baseline JIT [12] based on the copy-and-patch approach [37]. CPython uses reference counting in combination with a cycle-finding garbage collector to manage its memory. CPython boxes all of its objects, including integers and floating point numbers. It does not use pointer tagging or similar techniques.

2.2 The CPython C API

While writing Python code is the normal way of interacting with the CPython runtime, it is also possible to interact with it using its C API. The C API is commonly used to create C extension modules, which expose new functions and data types to Python code, that are implemented in C.[1] The C API consists of a number of free functions and data types, some of which are opaque to the API client. It gives the tools to create Python objects, introspect them, call Python functions, and more from C.

As an example, a minimal C extension can be found in Listing 1. In this C extension, `PyInit_signature` sets up the module. It calls the C API function `PyModule_Create`, which takes a description of the module it wants to create: the `PyModuleDef`. In the struct, we only define the minimal features for this example: a name, documentation, and a method table.

`PyModule_Create` walks the method table (the array of `PyMethodDef`), creating `PyCFunctionObjects` from the descriptions. In this example, it creates a C function called `inc` that takes one argument (indicated by the flag `METH_O`). When called, the C extension wrapper code inside the Python runtime does argument count checking and then passes the C function `inc` the singular argument it needs.

[1]The C API also makes it possible to embed CPython into other projects, but this usecase is much less frequent and we won't discuss it in this paper.

```c
#include <Python.h>

long inc_impl(long arg) { return arg+1; }

PyObject* inc(PyObject* module, PyObject* obj)
    {
  long l = PyLong_AsLong(obj);
  if (l == -1 && PyErr_Occurred()) return NULL;
  return PyLong_FromLong(inc_impl(l));
}

static PyMethodDef signature_methods[] = {
  {"inc", inc, METH_O, "Add one to a long."},
  {NULL, NULL, 0, NULL},
};

static struct PyModuleDef def = {
  PyModuleDef_HEAD_INIT, "signature", "doc",
    -1, signature_methods, NULL, NULL, NULL,
    NULL };

PyMODINIT_FUNC PyInit_signature(void) {
  return PyModule_Create(&def);
}
```

Listing 1. A tiny C extension, `signature`, exposing one function callable from Python, `inc`. The function `PyInit_signature` is called on first import.

```c
SIG(inc, LIST(T_C_LONG), T_C_LONG)
static PyMethodDef signature_methods[] = {
  TYPED_SIG(inc, inc, METH_O, "doc"),
  {NULL, NULL, 0, NULL},
};
```

Listing 2. Adding typing information to the minimal C extension in Listing 1.

Extension authors are typically required to write their own argument processing code.[2] In this case, we convert the argument from a Python `int` object to a C `long` and raise an exception if that is not possible (this happens in `PyLong_AsLong`). Then we call the underlying C function, `inc_impl`, and box up the result for consumption in Python.

2.3 PyPy

PyPy is an alternative implementation of the Python language. PyPy is not implemented in C, but in RPython, a statically typed subset of Python 2 [2]. PyPy uses a moving generational garbage collector for managing its memory. PyPy contains a tracing just-in-time compiler to speed up the execution of Python code [6]. To help the JIT compiler generate better code, PyPy's object model is quite different

[2]There is a CPython-internal preprocessor called Argument Clinic that automates some of the work in writing argument processing code, but it is not meant for external projects. We discuss it in Section 6.2.

than that of CPython. In particular, Python instances are implemented using Self-style [13] maps/hidden classes [7].

2.4 Cpyext and its Problems

To allow PyPy to use the vast quantity of C extensions that exist for CPython, PyPy has a compatibility layer for the CPython C API, called *cpyext* [15]. It exposes (a subset of) the functions and structs of the C API.

Implementing this compatibility layer is quite challenging because CPython and PyPy function quite differently. The CPython C API exposes a number of internal implementation details of CPython, most noticeably CPython's choice of reference counting for memory management. Handling Python objects from C requires the correct usage of `Py_INCREF` and `Py_DECREF` everywhere in the C code.[3] PyPy objects don't have a reference count field as the first word in each object, and the PyPy GC would really like to be able to move objects as part of its minor collections. Therefore, PyPy creates tiny CPython-layout compatible structs for those of its objects that that are passed to C functions.

Maintaining the link and converting between PyPy objects and CPython-layout compatible PyObjects is expensive. Every time PyPy calls a C function, we need to convert all of the function arguments to PyObjects and then convert the result back to a PyPy object. This is particularly expensive for boxed primitive types, because the C code will very likely just unbox them (with API functions such `PyLong_AsLong`) to work with the primitive values. However, because all the argument parsing and unboxing is done in C code and is therefore a black box, PyPy has no way to circumvent it. This is the central problem that we want to address in this paper and is visualized in Figure 1.

The problem becomes even more pronounced when PyPy's JIT is involved. The JIT will often compile the Python code that calls a C-implemented function in a C extension module. The JIT infers the types of variables that are used in Python code at run-time. In the case of integers the JIT will unbox them and store their integer values in machine registers [5]. In order to now pass such an unboxed integer as an argument to a C function, the JIT first has to recreate (i.e. allocate) a PyPy object, which is then converted to a PyObject in order to pass it to the C code. The C code will then unbox the value to work with the integer value itself. This kind of ping-pong between various representations is incredibly costly.

3 Adding Type Information to C Extensions

We want to pass known type information from the C extension functions back to Python. Doing this in a backwards compatible manner is non-trivial. We describe some of the problems of doing so in this section.

[3] This includes the implicit runtime-owned wrapper code around C extension function calls.

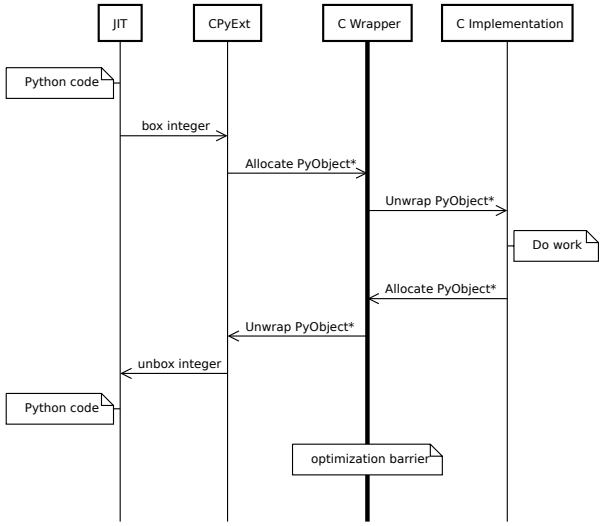


Figure 1. Example call to from Python code optimized by the JIT to a C function, passing one integer argument. The diagram shows all the needed conversions in the process.

3.1 Exposing Type Information

Our broad goal is to allow the runtime—the *caller*—to make decisions about argument type checking and unboxing instead of the C extension—the callee. To do that, the runtime needs to know some type information and other metadata about each C function.

In order to make this work, adding our type information must be backwards-compatible both with PyPy and CPython. By this we mean that any version of CPython or PyPy that does not understand the annotations should not be tripped up by them: the C extensions should compile, load, and run just fine. This is tricky because the Python C API is not very extensible and also because of some guarantees that Python makes about its C API.

3.2 The Stable ABI

The C API exposed by CPython is consumed not only be extension authors, but also by CPython developers. Over the years, API clients have come to rely on details that have been exposed by the CPython C API, such as struct sizes and structs fields that were originally intended for internal use only. Some source code has been completely lost to time, and all people have are shared objects that call into C API code. To support this CPython has defined a stable binary interface (ABI).

Under the stable ABI contract, functions are not removed, functions do not add or remove parameters, and data types do not change size. There are a few additional minor restrictions.

If we're going to try and make existing C API interactions faster under PyPy with minimal effort, we need to find a way to add lightweight annotations to methods. We can't change

types, we can't change functions, and we can't make people work too hard.

Because both the size of `PyMethodDef`[4] and the sizes and types of its fields cannot change, we must smuggle in a pointer to more information stored elsewhere.

3.3 Sneaking in Pointers

We can at least signal that there *is* additional information for a `PyMethodDef` by taking another bit in the `ml_flags` bitset. We propose the `METH_TYPED` bit. When this bit is set, the PyPy extension module loader knows to look for the extra type information.

```
struct PyMethodDef {
    const char  *ml_name;
    void        *ml_meth;
    int         ml_flags;
    const char  *ml_doc;
};
typedef struct PyMethodDef PyMethodDef;
```

Instead of the usual C string literal assigned to `ml_name`, we store the string in the `PyPyTypedMethodMetadata` struct and point `ml_name` to that buffer. The name size we chose is arbitrary. We then calculate an offset from that field to the beginning of the struct to use the added fields.

```
struct PyPyTypedMethodMetadata {
  int* arg_types;  // sentinel value of -1
  int ret_type;    // negative => can raise
  void* underlying_func;
  const char ml_name[100];
};
typedef struct PyPyTypedMethodMetadata
    PyPyTypedMethodMetadata;
```

A sample typed method looks like:

```
int inc_arg_types[] = {T_C_LONG, -1};
struct PyPyTypedMethodMetadata inc_sig = {
  inc_arg_types, T_C_LONG, inc_impl, "inc",
};
static PyMethodDef signature_methods[] = {
  {inc_sig.ml_name, inc, METH_O | METH_TYPED, "
    doc"},
  {NULL, NULL, 0, NULL},
};
```

To make this less irritating to write, we also provide macros to reach the form that we already saw in Listing 2. The macros also provide another feature: backwards compatibility. Instead of doing `#ifdef` yourself for type signature feature detection, the macros do it for you. On runtimes that support the `METH_TYPED` flag, they emit signatures. On runtimes that do not, they emit only standard C API method metadata. For now, we only support the C types `long`, `double`, and `PyObject*` (but extending the set of supported types is not conceptually hard).

[4] https://docs.python.org/3/c-api/structures.html#c.PyMethodDef

Once we know that the type information exists, we can use a trick from the Linux kernel [23] and read backwards from the `ml_name`:

```
PyPyTypedMethodMetadata*
GetTypedSignature(PyMethodDef* def)
{
  return (PyPyTypedMethodMetadata*)(def->
    ml_name - offsetof(PyPyTypedMethodMetadata,
     ml_name));
}
```

4 Using Type Information in PyPy

Once the argument and return type information is in place for a C extension, this information can be used in cpyext. When we load a C extension module into PyPy, we load the module's methods. We check if each method has a `METH_TYPED` flag set. If it does, we find the metadata, build the signature, and store it on the internal method object.

When the function is called from PyPy, we first check whether the called function has type information attached. If that is the case, cpyext can use a fast path for implementing the call. The arguments that are declared to be primitive types can be type-checked on the PyPy side, without reboxing and subsequent conversion to `PyObject*`. The call can then use the `underlying_func` function pointer and therefore skip the overhead of whatever Python calling convention the function uses.

Being able to do the type checks for primitive arguments on the PyPy side (as opposed to doing it in C) also meshes with PyPy's JIT type annotation, which means the type check may not be required at all.

Last, instead of doing the slow and generic exception check, PyPy knows if the function may never raise an exception—so it need not check—or what special sentinel value to look for if it does raise. Functions can return `NULL`, or `-1`, or something else depending on the return type to signal an error. This fast value check removes the need for the full `PyErr_Occurred()` call in the case where the function did not signal that it raised. It is similar to CPython's existing strategy for exception checking.

5 Evaluation

To evaluate our changes, we compare our modified PyPy against mainline PyPy, CPython[5] [6], and GraalPy. We also measure our modified PyPy *with the JIT disabled* against mainline PyPy with the JIT disabled.

At this early stage of our research we are only using some micro-benchmarks. Every micro-benchmark is calling a C function many times in a hot loop. The different benchmarks

[5] We also tested the alpha release of the upcoming CPython 3.13 and it gave similar, if slightly slower results than CPython 3.12.

[6] We would have also liked to benchmark against the Cinder JIT, but the open-source build for the JIT was broken at the time of writing.

exercise different kinds of calls from Python into native code. All the C function are themselves doing very little actual work. This means that the performance is dominated by the overheads of the C API and converting between the different representations. The results therefore represent the performance ceiling: the best possible improvements our approach can can make. They are not meant to represent real-world code. The four microbenchmarks are:

- `ffibench`, calling a `METH_O` function with C types `long→long`
- `objbench`, calling a `METH_FASTCALL` function with C types `PyObject* → long → long`
- `idbench`, calling a `METH_O` identity function with C types `PyObject* → PyObject*`.
- `idbench_exc`, also calling a `METH_O` identity function with C types `PyObject* → PyObject*`, but this variant is annotated with the information that it can raise an exception.

We run each benchmark 3 times for 1 billion iterations.[7] We then make a box and whisker plot of time taken for the three runs. There is not much variance per runtime per benchmark. The results can be seen in Figure 2.

The best case benchmark for optimization is `ffibench`. While the wrapper function deals in heap allocated Python `int`s, the underlying C function takes and returns a `long`. The type information lets PyPy's JIT skip the overhead of creating `PyObject*` for the argument and the return value completely. Our changes bring PyPy from slowest (about 164 seconds) to fastest (about 2.7 seconds).

In `objbench` one of the C-functions arguments is unboxed and the other one requires allocating a `PyObject` for each call. We do this to approximate a more realistic call. Not all calls to C extensions are going to be able to avoid all boxing. In this benchmark, even though PyPy is still allocating a `PyObject`, removing the overhead from building an array of `PyObject` for the `METH_FASTCALL` convention and from boxing the result makes a big difference. Our changes bring PyPy 3.10 from second slowest to fastest.

The third benchmark, `idbench`, benchmarks the identity function. It is a `METH_O` function (one parameter) but this time it cannot type-specialize its parameters or return value. This means that the runtime must box up the argument into a `PyObject` and unbox the return value the same way. Unlike with `METH_FASTCALL`, we are not eliminating any overhead for allocating an argument tuple/array. Despite this, we cut PyPy's execution time in half and are a close second place for time. GraalPy 22 is fastest, we suspect due to Sulong being able to optimize the call to the identity function.

[7]We specifically picked such high iteration counts to to give the Graal JIT enough time to warm up and have Sulong (Section 6.1) kick in for GraalPy 22. We omit GraalPy 23 because it took much longer than the other runtimes to finish and we killed the process. We hypothesize this is due to the removal of Sulong between versions 22 and 23. This makes GraalPy 23+ another good candidate for using type information from C extensions.

The last benchmark, `idbench_exc`, is similar to `idbench` except that it can raise an exception. This means that the runtime must check first for an agreed-upon sentinel value (in this case, `NULL`) and second check for an exception. Even though we are checking exceptions, we cut PyPy's execution time in half and are a close second place for time.

Last, we ran all four benchmarks in PyPy with the JIT turned off (Figure 3). Our changes still improve PyPy performance because the runtime need not allocate the arguments array of `PyObjects` for each call to a `METH_FASTCALL` function; it knows how many arguments to pass and can pass them in registers when calling the underlying function. Additionally, the runtime can skip slow thread-local storage (TLS) lookups in exception checking for C extension functions that cannot raise Python exceptions.

6 Prior Work

Monat et al. [31] built a multi-language static analysis platform called Mopsa. They analyze several open-source libraries and their unit tests and find that multiple Python↔C calls happen per unit test (ranging from 2.7 to 51.7). Their analysis could potentially be repurposed to generate type information for C extensions (they note that they would like to infer standard library type information using their techniques). They note that the analysis is limited by use of hard-to-analyze Python libraries and imprecision in the C analysis. This work is expanded in Monat [30].

Similarly, Hu et al. [22] describe PyCType, which automatically infers the argument types for functions exposed by Python C extension in order to find bugs. This information should be usable for runtime performance improvements.

Tsai et al. [34] use the LLVM JIT to speed up the performance of JNA callbacks in the Java Hotspot Server VM [34]. Their approach yields 8-16% performance improvements and does not apply to *calling* C functions from Java (only the other way around).

Li and Tan [26, 27] find bugs in Java Native Interface (JNI) modules related to exception-checking. Their tools, JET and TurboJet, implement a static analysis to find missing declarations for checked exceptions. Automatically finding and annotating C extensions that do not raise exceptions could help improve run-time performance with less manual work.

The HPy project [1] is a complete re-design of the CPython C API from the ground up. One of its main goals is to move away from reference counting being visible in the API. However, it still does nothing to solve the problems discussed in this paper.

6.1 Sulong

Sulong [33] is an self-optimizing interpreter based on the Truffle framework [36] for LLVM [24] bitcode. It can be used to speed up calling from Python into C extensions by JIT-compiling both the Python and the C code in the same

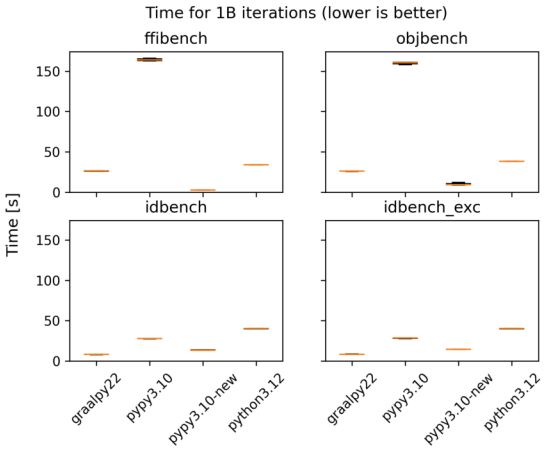


Figure 2. Benchmark results for PyPy with JIT on

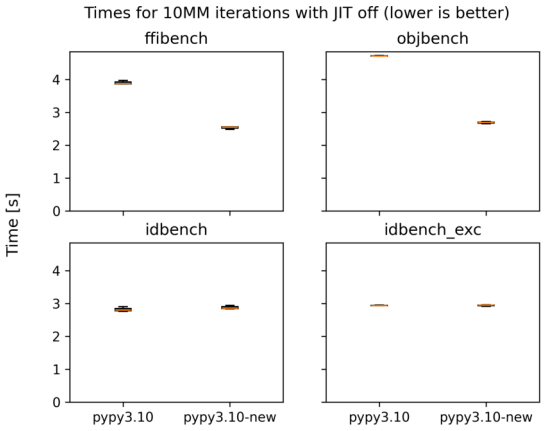


Figure 3. Benchmark results for PyPy with JIT off

```
from __static__ import native, int32, box

@native("libc.so.6")
def abs(i: int32) -> int32:
    pass

def abs_wrapper(i: int) -> int:
    j: int32 = int32(i)
    return box(abs(j))
```

Listing 3. Static Python example taken from the Cinder test suite. In this example, the `abs` function is declared as a stub to be loaded from `libc.so`. It takes an unboxed (C) `int32` and returns the same. The `test` function takes a boxed (Python) `int`, unboxes, calls `abs`, and re-boxes.

compilation unit using the Graal JIT compiler. This gets rid of most of the conversion overhead, but has the downside that called C function is also running on top of Sulong, which is slower than using a well-tuned static C compiler for the core algorithms of extensions.

6.2 Argument Clinic

The CPython runtime has an internal tool called the Argument Clinic [21] that takes as input descriptions of C functions and generates argument processing code and documentation strings for them. The clinic is used only to generate standard library code in CPython.

6.3 Static Python

The Cinder project[8] is perhaps most widely known for its JIT compiler, but it also includes a compiler and runtime for a statically-typed dialect of Python known as Static Python (SP) [28]. The SP compiler contains primitives for declaring, resolving, and calling C functions directly from Static Python code. The SP compiler produces typed bytecode, which allows the JIT to compile specialized, zero-overhead calls to these C functions. See Listing 3 for an example.

The JIT uses `dlsym` and the typed SP bytecode to build a `NativeTarget`: a function pointer, return type, and argument types.

Annotating declaration of existing C functions is a manual process. Also, the SP compiler does not allow passing `int`s into the unboxed `abs` function; callers must explicitly unbox.

7 Conclusion

We have shown that adding type information to C extensions can make them faster under the PyPy JIT. We have also shown that the techniques improve performance on PyPy even with the JIT compiler turned *off*.

Type information specialization is effective even in an interpreted context and potentially even without unboxed objects. We believe that this technique is not limited to PyPy and can be adopted by other dynamic language runtimes. For example, in runtimes such as Skybison and MicroPython with efficient representations for small objects (such as tagged integers in pointers), the runtime need not allocate a `PyObject` for each argument and return value; it can use the efficient representation directly [16, 18].

8 Future Work

Given how promising our early results are, we would like to build out support for more complex signatures, such as support for C strings and primitive types wider than 64 bits. We would also like to support a more expressive declaration language, such as the one used in the Argument Clinic.

In the future, we would like to see these type annotations automatically emitted by binding generators. Projects such as Cython and PyO3 already have machinery for generating wrapper code for C functions, and therefore have sufficient knowledge about the C function types.

[8] https://github.com/facebookincubator/cinder/

Such binding generators also have more insight into the effects that happen inside a native function. For example, Cython may be able to statically guarantee that a function does not raise an exception, need to acquire the GIL, or something else. Reducing the set of effects from "all effects possible" could aid the PyPy optimizer.

Supporting Cython would unlock performance in commonly-used numerical and scientific packages such as NumPy [20] and SciPy [35], which use Cython extensively.

Acknowledgments

To Sarah, for proposing the title. To Kate McKinnon, for ongoing comedic genius.

References

[1] [n. d.]. *HPy - A better C API for Python.* https://hpyproject.org/

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*. ACM, Montreal, Quebec, Canada. https://doi.org/10.1145/1297081.1297091

[3] Gergö Barany. 2014. Python interpreter performance deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications*. 1–9. https://dl.acm.org/doi/10.1145/2617548.2617552

[4] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science and Engg.* 13, 2 (mar 2011), 31–39. https://doi.org/10.1109/MCSE.2010.118

[5] Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Austin, Texas, USA) *(PEPM '11)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/1929501.1929508

[6] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*. ACM, Genova, Italy, 18–25. https://doi.org/10.1145/1565824.1565827

[7] Carl Friedrich Bolz and Laurence Tratt. 2015. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming* 98 (Feb. 2015), 408–421. https://doi.org/10.1016/j.scico.2013.02.001

[8] Stefan Brunthaler. 2010. Efficient inline caching without dynamic translation. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung (Eds.). ACM, 2155–2156. https://doi.org/10.1145/1774088.1774542

[9] Stefan Brunthaler. 2010. Efficient interpretation using quickening. In *Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, October 18, 2010, Reno, Nevada, USA*, William D. Clinger (Ed.). ACM, 1–14. https://doi.org/10.1145/1869631.1869633

[10] Stefan Brunthaler. 2010. Inline Caching Meets Quickening. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 429–451. https://doi.org/10.1007/978-3-642-14107-2_21

[11] Stefan Brunthaler. 2021. Multi-Level Quickening: Ten Years Later. *CoRR* abs/2109.02958 (2021). https://doi.org/10.48550/arXiv.2109.02958 arXiv:2109.02958

[12] Brandt Bucher. 2023. https://github.com/python/cpython/pull/113465.

[13] C. Chambers, D. Ungar, and E. Lee. 1989. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, Vol. 24. https://doi.org/10.1145/74878.74884

[14] Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. 2023. Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 20–33. https://doi.org/10.1145/3617651.3622982

[15] Antonio Cuni. 2018. Inside cpyext: Why emulating CPython C API is so Hard. https://www.pypy.org/posts/2018/09/inside-cpyext-why-emulating-cpython-c-8083064623681286567.html

[16] MicroPython Developers. 2024. smallint.h. https://github.com/micropython/micropython/blob/master/py/smallint.h.

[17] PyO3 Developers. 2024. The PyO3 user guide. https://pyo3.rs/v0.20.2/.

[18] Skybison Developers. 2024. objects.h. https://github.com/tekknolagi/skybison/blob/trunk/runtime/objects.h.

[19] Jake Edge. 2021. Making CPython faster. https://lwn.net/Articles/857754/

[20] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[21] Larry Hastings. 2024. Argument Clinic How-To. https://docs.python.org/3.10/howto/clinic.html.

[22] Mingzhe Hu, Yu Zhang, Wenchao Huang, and Yan Xiong. 2021. Static Type Inference for Foreign Functions of Python. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 423–433. https://doi.org/10.1109/ISSRE52982.2021.00051

[23] Greg Kroah-Hartman. 2005. container_of(). http://www.kroah.com/log/linux/container_of.html

[24] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[25] Jukka Lehtosalo. [n. d.]. mypy - About. https://mypy-lang.org/about.html.

[26] Siliang Li and Gang Tan. 2011. JET: exception checking in the Java native interface. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 345–358. https://doi.org/10.1145/2048066.2048095

[27] Siliang Li and Gang Tan. 2014. Exception analysis in the Java Native Interface. *Science of Computer Programming* 89 (2014), 273–297. https://doi.org/10.1016/j.scico.2014.01.018

[28] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2022. Gradual Soundness: Lessons from Static Python. *The Art, Science, and Engineering of Programming* 7, 1 (June 2022), 2:1–2:40. https://doi.org/10.22152/programming-journal.org/2023/7/2

[29] Kate McKinnon and Colin Jost. [n. d.]. *Weekend Update: Dr. Wenowdis on Trump's Televised Health Exam - SNL*. Saturday Night Live. https://www.youtube.com/watch?v=2kQxVwYwrME

[30] Raphaël Monat. 2021. *Static type and value analysis by abstract interpretation of Python programs with native C libraries*. PhD thesis. Sorbonne Université. https://theses.hal.science/tel-03533030/document

[31] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 323–345. https://doi.org/10.1007/978-3-030-88806-0_16

[32] Russell Power and Alex Rubinsteyn. 2013. How fast can we make interpreted Python? *CoRR* abs/1306.6047 (2013). https://doi.org/10.48550/arXiv.1306.6047 arXiv:1306.6047

[33] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing low-level languages to the JVM: efficient execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. Association for Computing Machinery, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[34] Yu-Hsin Tsai, I-Wei Wu, I-Chun Liu, and Jean Jyh-Jiun Shann. 2013. Improving performance of JNA by using LLVM JIT compiler. In *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*. 483–488. https://doi.org/10.1109/ICIS.2013.6607886

[35] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[36] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[37] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–30. https://doi.org/10.1145/3485513

Received 04-MAR-2024; accepted 2023-04-19

Interleaving Static Analysis and LLM Prompting

Patrick J. Chapman
University of California, Davis
Davis, USA
pchapman@ucdavis.edu

Cindy Rubio-González
University of California, Davis
Davis, USA
crubio@ucdavis.edu

Aditya V. Thakur
University of California, Davis
Davis, USA
avthakur@ucdavis.edu

Abstract

This paper presents a new approach for using Large Language Models (LLMs) to improve static program analysis. Specifically, during program analysis, we *interleave* calls to the static analyzer and queries to the LLM: the prompt used to query the LLM is constructed using intermediate results from the static analysis, and the result from the LLM query is used for subsequent analysis of the program. We apply this novel approach to the problem of error-specification inference of functions in systems code written in C; i.e., inferring the set of values returned by each function upon error, which can aid in program understanding as well as in finding error-handling bugs. We evaluate our approach on real-world C programs, such as MbedTLS and zlib, by incorporating LLMs into EESI, a state-of-the-art static analysis for error-specification inference. Compared to EESI, our approach achieves higher recall across all benchmarks (from average of 52.55% to 77.83%) and higher F1-score (from average of 0.612 to 0.804) while maintaining precision (from average of 86.67% to 85.12%).

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; *Error handling and recovery*; • **Computing methodologies** → **Natural language processing**.

Keywords: static analysis, large language model, error handling, error specifications

ACM Reference Format:
Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. 2024. Interleaving Static Analysis and LLM Prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3652588.3663317

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663317

1 Introduction

This paper presents a new approach for using *Large Language Models (LLMs)* to improve static program analysis. LLMs [18, 25] have been shown to demonstrate impressive reasoning abilities in natural and programming languages tasks via few-shot [3] and chain-of-thought [28] prompting. The approach presented in this paper utilizes this reasoning ability of LLMs when the static analysis is unable to make progress; the results of the query to the LLM are used for subsequent analysis. Furthermore, the query (or prompt) to the LLM incorporates the current results of the static analysis, which enables it to provide more accurate results. In this way, our approach *interleaves* calls to the static analyzer and the LLM, with each utilizing the results of the other.

We apply this novel approach to the problem of *error-specification inference* of functions in systems code written in C, i.e., inferring the set of values returned by each function upon error (Section 2). The C language does not have built-in exception or error handling; thus, a common idiomatic practice for error-handling is to check the return value of a function on error, i.e., the *return code idiom*. These return values indicate the functions' error specifications, which can aid in program understanding as well as in finding error-handling bugs. EESI [6] has shown higher effectiveness and performance at inferring error specifications compared to prior approaches [1, 7, 14]. Our approach interleaves calls to the EESI static analyzer and the LLM (Figure 1).

We evaluated our approach on six real-world C programs, such as MbedTLS and zlib (Section 5). Our approach improves recall and F1-score over EESI from 52.55% to 77.83% and 0.612 to 0.804, respectively, while maintaining a high precision of 85.12% compared to 86.67% in EESI. Our evaluation demonstrates that by interleaving static analysis and LLM prompting, we can significantly improve upon the error specification inference capabilities of just a static analyzer.

The contributions of this paper are as follows:

- We propose a technique for interleaving a static analysis with LLM prompting.
- We designed a tool for error specification inference of C programs using our approach of combining EESI static analyzer and LLM prompts.
- We evaluate our approach on 6 real-world C programs comparing it with prior state-of-the-art EESI. We provide an ablation study on the individual components of our approach.

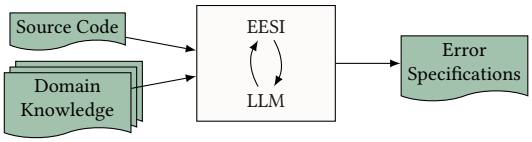


Figure 1. Our approach infers error specifications by interleaving calls to the EESI static analyzer and the LLM.

2 Background

Error Specification Inference. The C language does not feature programming constructs for exception handling. Instead, developers often use the *return code idiom* to indicate error. An *error specification* refers to the set of values returned by a function upon error. Because it is not possible to enforce compile-time rules regarding error code propagation and checking, the return code idiom often leads to bugs, e.g., developers may miss or incorrectly check the error return values of functions.

A few approaches have presented techniques for inferring error specifications [1, 6–8, 14, 30]. In this paper, we consider a state-of-the-art static program analysis using abstract interpretation for error specification inference named EESI [6]. As input, EESI takes in multiple forms of optional user-supplied *initial domain knowledge*: (1) initial error specifications, (2) error codes, (3) success codes, and (4) error-only functions (only called along error paths). With this initial domain knowledge, EESI uses static analysis to infer new error specifications.

While EESI has demonstrated success in error specification inference, it has two inherent limitations that affect its recall and precision: (1) *incomplete program facts*, and (2) *third–party* functions. As EESI is a static program analyzer using abstract interpretation to infer program semantics related to idiomatic practices, it provides approximations that may be insufficient in learning enough program facts for error specification inference. One important source of incomplete knowledge is *third-party functions*. Third-party functions are called within a program, but defined elsewhere. Because the analyzer does not have access to the source code, it cannot reason about their error specifications.

Large Language Models (LLMs). LLMs are language models trained on large amounts of data for tasks such as text generation and language understanding. These models have been developed for both natural language [25] and programming languages [23], while some models are trained for both [18, 22, 26]. One of the key components of LLMs are the *prompts*, i.e., the input to the LLM. There has been considerable research done in recent years related to the generation of prompts that improve the performance of LLMs in various tasks [2, 22, 27, 28]. These approaches include concepts such as *chain-of-thought* [28], where LLMs are given question and answer as examples with the associated chain-of-though

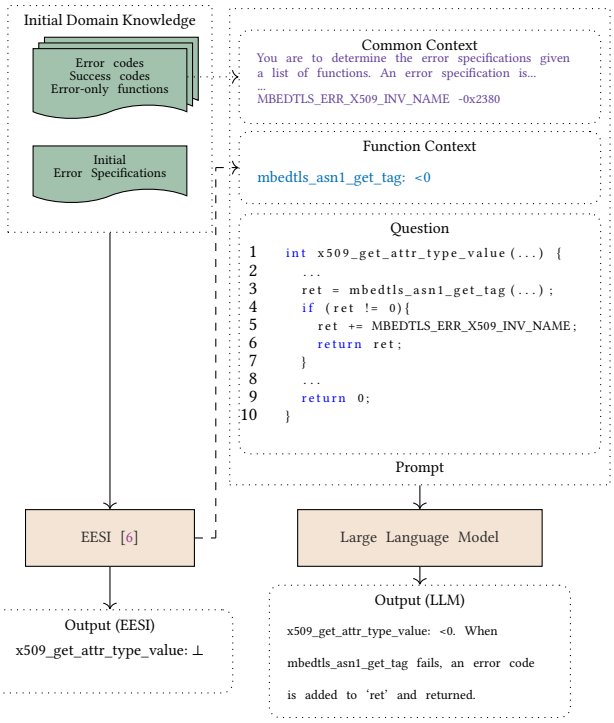


Figure 2. Using EESI and the LLM to infer error specifications

reasoning, and *self-consistency* [27] prompting, where LLMs are prompted with the same question multiple times, using the most consistent answer given.

3 Overview Example

This section illustrates how our approach of interleaving calls to the EESI static analyzer and the LLMs to infer error specifications. Consider the function `x509_get_attr_type_value` in MbedTLS. EESI alone is unable to infer its error specification; EESI infers $\bot$ as the function error-specification as shown in Figure 2.

The LLM alone is also unable to infer its error specification. We can construct a prompt to the LLM that includes the general description of the error specification inference problem (Common Context in Figure 2) as well as the source code of the function (Question in Figure 2). However, querying the LLM with just this information is not sufficient to give us the correct error specification. In particular, the LLM infers that the error condition for `mbedtls_asn1_get_tag` is $\neq \mathbf{0}$ from the conditional check. Even when the value of the error code `MBEDTLS_ERR_X509_INV_NAME` is included in the Common Context, the incorrect assumption about the called function leads the LLM to incorrectly infer that the return value on the error-path is the negative error code added with any non-zero value; that is, the LLM infers that the error value could be anything, and the error specification is $\top$, instead of $< \mathbf{0}$.

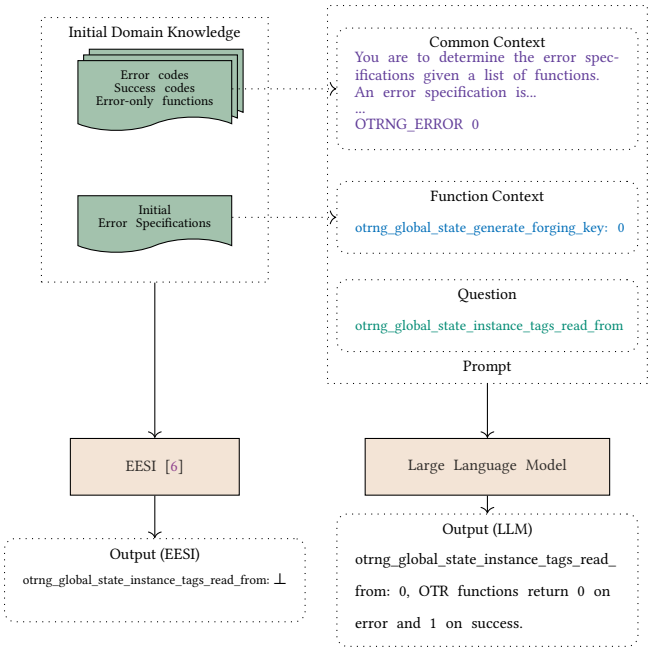


Figure 3. Using the LLM to infer error spefication of a third-party function

However, if we also include intermediate results from the EESI static analyzer in the LLM prompt, then the LLM is able to return the fact that `x509_get_attr_type_value` returns a value <0 on error. In particular, the LLM prompt includes the error specification of the function `mbedtls_asn1_tag` that is called from `x509_get_attr_type_value` (Function Context in Figure 2); this error specification is inferred by the EESI static analyzer.

This example illustrates how our approach provides benefits over purely static analysis or LLM approaches by interleaving calls to the static analyzer and the LLM: the LLM is used only when the static analyzer is unable to make progress, and the LLM prompt includes intermediate information gleaned by the static analyzer. Furthermore, the output of the LLM is fed back into the EESI static analyzer. For example, the LLM's specification for `x509_get_attr_type_value` would allow EESI to subsequently find the error specification (<0) for `mbedtls_x509_get_name` from analyzing its implementation:

```
if( ( ret = x509_get_attr_type_value(...) ) != 0 )
    return( ret )
```

The specifics about the LLM prompt construction; viz, Common Context, Function Context, and Question, are deferred to Section 4.1.

Figure 3 illustrates another scenario illustrating the benefits of incorporating calls to an LLM in the static analyzer. The function `otrng_global_state_instance_tags_read_from` is a third-party function called in Pidgin OTRv4. Because the source code of this function is not available, EESI is unable to infer its error specification, and consequently, it might not be able to infer the specifications of functions that call it. However, constructing an LLM prompt that includes information from the user-provided domain knowledge, the LLM is able to correctly infer the error specification for `otrng_global_state_instance_tags_read_from`.

4 Approach

We illustrate our approach for interleaving static analysis and LLMs in Figure 1. The input is the program source code and optional domain knowledge, and the output are the function error specifications inferred by the analysis.

4.1 Building Prompts

When interacting with the LLM, we construct a prompt that consists of the *Common Context*, *Function Context*, and *Question*, as mentioned in Section 3.

Common Context. The prompt *Common Context* used for error specification inference consists of a problem description and an explanation of the abstract domain used by the EESI static analyzer. We provide the explanation of the abstract domain, because we want the LLM to output its learned error specifications using this domain. Relating to the program under analysis, the *Common Context* also contains any error codes, success codes, and error-only functions from the domain knowledge input. We include additional observed idiomatic practices related to the return code idiom:

1. Error specification values must be a subset of the returned values of a function.
2. Unknown error specifications are $\perp$.
3. Success values are not part of the error specification.
4. The NULL return value is equal to 0.
5. Error codes from standard library functions are positive integers.
6. Macros may check return values and return if failing.

We also provide multiple, basic *chain-of-thought examples* that consist of a function definition and its associated error specification, with a chain-of-thought explanation. We do so to demonstrate the task of error specification inference and so that the LLM generates parse-able output. We do this, in addition to providing the explanation of the abstract domain, in order to limit the LLM from generating output that is unexpected. However, if the LLM output does not follow the expected format, then the related error specification will consist of the $\perp$ element, i.e., unknown. For example, the expected output for `malloc` would be *malloc: 0*.

Function Context. The *Function Context* of the prompt relates to any relevant function error specifications for the function that is being queried by the LLM. The *Function Context* that is generated depends on the selected LLM query function that will be explained further when introducing our algorithm, Algorithm 1. In all cases, these error specifications are provided as *few-shot examples* to the LLM, with the aim to

generate parse-able output, as well as providing demonstrative examples to the LLM. These error specifications provide additional context that can assist the LLM when it comes to understanding returned error values. This is especially true when there are functions that exist in the same library as demonstrated with Figure 2.

Question. The *Question* in all constructed prompts asks for the LLM to return any error specification that it is confident in using the abstract domain used by EESI.

4.2 Error Specification Inference

For the task of error specification inference, we present Algorithm 1 to demonstrate how the static analyzer and LLM are used. Our algorithm takes in the domain knowledge as a map of program facts P and the set of functions from the source code F. The algorithm returns the updated facts P after performing analysis.

The analysis begins by iterating over the functions $f \in F$ bottom-up in the Call Graph (CG) as demonstrated on Line 2. This ensures that that called functions are inferred before their caller, because called functions provide additional context for error specification inference. Note, for brevity, we do not include in the algorithm that we perform a fixpoint on the Strongly Connected Components (SCC) in CG, as recursion may exist in the call chains. The algorithm algorithm will attempt to infer an error specification in one of three cases: (1) queryLLMThirdParty (Line 4), (2) runAnalysis (Line 6), or (3) queryLLMAnalysis (Line 8).

Algorithm 1: InferErrorSpecification(P, F)

INPUT: Map of program facts P, Set of functions F.
OUTPUT: Updated P with new error specifications.

```
1:  CG ←CallGraph(F)
2:  for all f ∈ reverseTopologicalSort(CG) do
3:      if isThirdParty(f) then
4:          spec ←queryLLMThirdParty(P, f)
5:      else
6:          spec ←runAnalysis(P, f,EESI )
7:          if spec = ⊥ then
8:              spec ←queryLLMAnalysis(P, f)
9:          end if
10:     end if
        P ←updateFacts(P, f, spec)
11: end for
12: return P
13: Function queryLLMAnalysis(P, f)
14:     question ←getSourceCode(f)
15:     functionContext ← getCalledErrorSpecifications(P, f)
16:     prompt ←buildPrompt(functionContext, question)
17:     spec ←parseOutput(queryLLM(prompt))
18:     return spec
19: EndFunction
20: Function queryLLMThirdParty(P, f)
21:     question ← getName(f)
22:     functionContext ←getErrorSpecifications(P)
23:     prompt ←buildPrompt(functionContext, question)
24:     spec ← parseOutput(queryLLM(prompt))
25:     return spec
26: EndFunction
```

4.2.1 Third-Party Function Error Specifications.

For each function, we first check if it is a third-party function (Line 3), and if it is, we perform queryLLMThirdParty as demonstrated on Line 4. Because the source code definition is not available for third-party functions, we cannot statically analyze it. As *Function Context* for the prompt, we provide the entire set of error specifications that are in P on Line 22. The *Question* in this case just simply lists the name of the function-of-interest (Line 21). The LLM is then queried, where the output is then parsed (Line 24) and if any error specification is learned, the program facts are updated (Line 10).

4.2.2 Error Specification Analysis.

If the function is not third-party, then the EESI static analyzer will perform its own analysis. EESI will determine if the error specification of the function is infallible ($\emptyset$), unknown ($\bot$), or any other value (e.g., $< \mathbf{0}$) from runAnalysis on Line 6. If this result is $\bot$ (Line 7), then we query the LLM once for the function under analysis with queryLLMAnalysis on Line 8.

Unlike the *Function Context* provided in queryLLMThirdParty, we only provide the known error specifications of called functions contained in the function definition (Line 15). We demonstrate an example of this in Figure 2, where error specification `mbedtls_asn1_get_tag` is learned from EESI and provided as *Function Context* to the LLM, correctly inferring `x509_get_attr_type_value`.

The constructed *Question* as part of the prompt consists of the source code of the function being analyzed (Line 14).

The resulting output from the LLM is then parsed (Line 17) and any newly learned error specification is updated in the program facts on Line 10.

4.2.3 Validating the LLM Response.

We re-query the LLM for every generated prompt to limit the side effects of *hallucination*. Hallucination refers to when LLMs make up information to satisfy a prompt, even if the provided *chain-of-thought* reasoning is contradictory. We specifically ask the LLM to ensure that the error specifications provided match the given chain-of-thought description from itself. Additionally, we also limit some of the imprecision by identifying two inconsistencies with formal reasoning. First, we do not infer error specifications if the resulting error value from the LLM includes a known success value. Second, we do not infer an error specification if the LLM states that the error specification is an improper superset of the return range of the function. As both of these program semantics are obtained via an approximation during the analysis of EESI, we cannot

Table 1. Selected benchmarks with their LOC and selected domain knowledge — initial error specifications, error-only (EO) functions, error codes, and success codes.

			Domain Knowledge			
					Codes	
Benchmark	KLOC	Ver.	Init. Specs	EO	Error	Success
Apache HTTPD	288	2.4.46	14	0	44	1
LittleFS	2	1.7.0	11	0	14	1
MbedTLS	255	2.21.0	21	1	221	1
Netdata	51	1.11.0	43	0	0	0
Pidgin OTRv4	15	4.0.2	34	0	0	0
zlib	18	1.2.11	7	0	6	1

guarantee that these inconsistencies are removed entirely, but we can utilize these rules to limit low-hanging fruit.

5 Experimental Evaluation

For our experimental evaluation, we perform an ablation study. We propose three research questions with one baseline to target components of our approach:

RQ0 How well does the static analysis of EESI perform? This is our baseline.

RQ1 What is the impact using the LLM to infer third-party error specifications, i.e., queryLLMThirdParty?

RQ2 What is the impact of using the direct LLM analysis, i.e., queryLLMAnalysis?

RQ3 What is the impact of interleaving EESI and the LLM?

Our code and data are publicly available at https://github.com/ucd-plse/eesi-llm.

5.1 Experimental Setup

Benchmarks. We consider a data set of six benchmark programs that represent a variety of error-handling patterns and system types, as listed in Table 1.

Domain Knowledge. For all approaches, we supply the same initial domain knowledge as input. *Initial error specifications* are identified via one of two strategies. The first is that we select applicable error specifications from a list of common and well-known standard library functions. The second is that we manually inspect a small subset of functions based on the program's call graph, supplying functions that appear lower in the call graph as initial domain knowledge. *Success* and *error codes* are mined automatically through pattern matching header files for patterns such as `ERR`, `err`, and `SUCCESS`. *Error-only functions* (only called on error paths) are selected via manual inspection. The manual effort involved in finding the above domain knowledge for all benchmarks took a total of one hour.

Evaluation metrics and ground truth. We measure precision, recall, and F1 (F1-score) — where we only consider a true positive (TP) to be a learned error specification that matches the ground truth exactly; for example, $\leq \mathbf{0}$ and $< \mathbf{0}$ are not equivalent and would be considered a false positive (FP). If the analysis determines an error specification is unknown $\perp$, then that is considered a false negative (FN). As every function-under-analysis will have an error specification, even infallible $\emptyset$ functions, we do not have true negatives (TN). For all metrics, we calculate based on a manually inspected ground-truth $\mathcal{G}$ as depicted in Table 2. For smaller benchmarks, we inspected all functions, but for larger benchmarks we randomly sampled a subset. We did so, as manual inspection over all functions is not feasible due to time constraints, as some functions may consist of hundreds or thousands of lines. Note, numbers represented in Table 2 do not count initial error specifications from the domain knowledge.

Table 2. Total number of functions, functions in $\mathcal{G}$, and third-party functions in $\mathcal{G}$.

Benchmark	Total	$\mathcal{G}$	Third Party $\in \mathcal{G}$
Apache HTTPD	1210	600 (49.6%)	135 (22.5%)
Little FS	60	60 (100.0%)	9 (15.0%)
MbedTLS	1211	598 (49.4%)	15 (2.5%)
Netdata	720	338 (47.6%)	74 (21.9%)
Pidgin OTRv4	277	277 (100.0%)	200 (72.2%)
zlib	126	126 (100.0%)	10 (7.9%)

Precision, *recall*, and *F1* are defined as:

$$Precision = \frac{TP_{\mathcal{G}}}{TP_{\mathcal{G}} + FP_{\mathcal{G}}} \qquad Recall = \frac{TP_{\mathcal{G}}}{TP_{\mathcal{G}} + FN_{\mathcal{G}}}$$

$$F1 = \frac{2 * Precision * Recall}{100 * (Precision + Recall)}$$

Implementation Details.

EESI is implemented using the LLVM infrastructure [15] to analyze bitcode and our LLM error specification inference uses GPT-4 [18] as the LLM. Our experiments were run on a 2.10 GHz Xeon Silver 4216 CPU with 384 GB of RAM.

5.2 Experimental Results

RQ0: How well does EESI perform in error specification inference?

For this task, we simply supply the initial domain knowledge and source code to the static analyzer of EESI and receive its inferred error specifications. The number of inferred error specifications are represented in Table 3. From these, we can see that the most common error specification inferred across all benchmarks is $< \mathbf{0}$. Many standard library functions indicate that they return a negative error code on failure, which has been adopted by many other software programs. However, this cannot be assumed for all functions, as indicated with benchmarks such as Apache HTTPD, which

Table 3. Specification counts, precision, recall, and F1-score for EESI

Benchmark	<0	>0	0	≤ 0	≥ 0	$\neq 0$	$\emptyset$	Total	Precision	Recall	F1
Apache HTTPD	16	42	16	0	1	27	183	285	94.16%	37.56%	0.537
Little FS	40	0	7	0	0	0	10	57	91.30%	75.00%	0.824
MbedTLS	723	10	48	3	0	1	246	1031	90.64%	84.55%	0.875
Netdata	17	35	108	0	1	1	116	278	64.60%	24.50%	0.355
Pidgin OTRv4	11	4	24	0	0	0	29	68	82.35%	10.33%	0.184
zlib	68	1	14	0	0	0	29	112	97.14%	83.33%	0.895

often can return <0, >0, and $\neq 0$ on error. Additionally, some programs may have a considerable number of infallible ($\emptyset$) functions, e.g., MbedTLS.

EESI achieves a precision ranging from 64.60% to 97.14% as seen in Table 3, averaging at 86.67% per benchmark. However, the recall varies even more depending on the benchmark, ranging from 10.33% to 83.33%, averaging 52.55%. The benchmark with the lowest recall, Pidgin OTRv4 (10.33%) is also notably the benchmark with the highest percentage of third-party functions at 72.2% as listed in Table 2.

RQ1: What is the impact of queryLLMThirdParty?

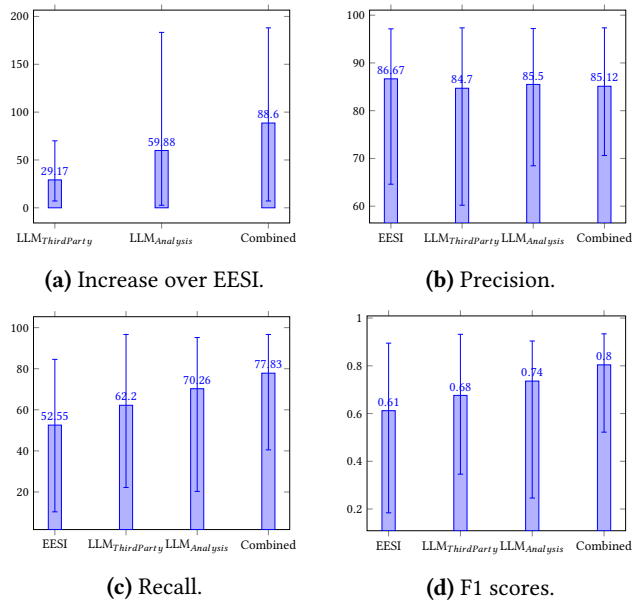
We measure the impact of queryLLMThirdParty by running it in the first step of our interleaved error specification inference. We then run the static analysis of EESI through runAnalysis, however, we do not call queryLLMAnalysis when EESI infers $\perp$.

As we can see demonstrated in Figure 4, we notice an average recall of 62.20% (Figure 4c) and average increase of 29.17% (Figure 4a) for inferred error specifications over EESI. Our precision remained similar to EESI (Figure 4b). We notice the largest impact for the benchmark Netdata, which increased the most by 70.50%. This benchmark was impacted significantly, as it refers to many well-known libraries such as pthread. We do not see as much of an increase in the Pidgin OTRv4, as many of the third-party libraries are for niche purposes, e.g., the GTK library. However, this is not the case for all library functions; for example, the error specification inference demonstrated in Figure 3 occurs through queryLLMThirdParty.

RQ2: What is the impact of using queryLLMAnalysis?

To isolate the contributions of queryLLMAnalysis, we skip queryLLMThirdParty in the workflow. Instead, we proceed to running the static analysis of EESI, followed by querying the LLM if the result is $\perp$.

The results depicted in Figure 4 demonstrate an average increase of 59.88% (Figure 4a) across all benchmarks, with an average recall of 70.26% (Figure 4c). Our benchmark that saw the largest percentage increase was Apache HTTPD at 183.33%, which contains the second highest percentage of third-party functions (Table 2). In Figure 2, we can see that the direct LLM analysis allows the LLM to reason about



(a) Increase over EESI. **(b)** Precision.

(c) Recall. **(d)** F1 scores.

Figure 4. Average increase, precision, recall, and F1-score for approaches. The minimum and maximum benchmark results are represented as error bars for their respective metric.

function bodies, even while the static analysis of EESI is insufficient.

RQ3: What is the impact of interleaving EESI and LLM?

For our combined approach, we utilize the entire workflow, calling both queryLLMThirdParty and queryLLMAnalysis. We see in Table 4, that our combination of prompting strategies is extremely beneficial in applications such as Pidgin OTRv4, Netdata, and Apache HTTPD. Significantly improving the recall and F1 over EESI in Table 3. In fact, we see an increase over the average F1 of EESI by +0.192 (Figure 4d). We also see the precision Δ on newly learned error specifications that were not inferred strictly via static analysis. With Netdata, we saw 144 new <0 error specifications inferred, with our overall precision going up for the benchmark. We make note that even when we do lose some precision, as seen with Apache HTTPD, we have an increase of 188.07% and still significantly improve our F1-score to 0.752.

Table 4. Specification counts, precision, recall, and F1-score for our framework interleaving static analysis and LLMs.

Benchmark	<0	>0	0	≤ 0	≥ 0	$\neq 0$	$\emptyset$	Total	Increase	Precision	Precision Δ	Recall	F1
Apache HTTPD	46	98	42	2	6	93	534	821	188.07 %	85.92%	75.20%	66.85%	0.752
Little FS	50	0	7	0	0	0	10	67	17.54%	92.86%	100.0%	92.86%	0.929
MbedTLS	818	15	64	4	0	1	272	1174	15.55%	90.34%	79.49%	96.68%	0.934
Netdata	161	72	222	2	4	1	234	696	150.36%	70.59%	75.40%	80.63%	0.753
Pidgin OTRv4	16	4	95	0	4	0	53	172	152.94%	73.68%	70.71%	40.50%	0.522
zlib	76	1	14	0	0	0	29	120	7.14%	97.35%	100.0%	89.43%	0.932

In Figure 4, our combination of prompting strategies to the LLM only improved upon the total number of inferred error specifications (Figure 4a), obtaining the highest recall (Figure 4c), and F1 (Figure 4d), while maintaining a similar precision (Figure 4b) to the analysis of EESI. We specifically highlight the advantages that each component has demonstrated, where queryLLMThirdParty demonstrated great success in assisting analyze benchmarks with a significant majority of third-party functions such as Pidgin OTRv4; where queryLLMAnalysis has demonstrated great success in analyzing function bodies directly, inferring error specifications in scenarios such as their called context.

6 Related Work

Error Specification Inference. Acharya and Xie [1] introduce techniques for mining error specifications for APIs using static traces. APEx [14] uses path-sensitive symbolic execution to find error-paths on the assumption that error paths are shorter than normal paths. Several other works [10, 20, 21, 24] find function error specifications via fault injection. MLPEx [30] is a machine-learning based approach that uses path-features to learn whether or not a program path is an error path. EESI [6] is a static analysis of C programs for error specification inference that allows the use of domain knowledge to bootstrap the analysis. Our task improves EESI by interleaving it with LLM prompting.

Program Analysis and LLMs. Ahmed and Devanbu [2] demonstrate that when a LLM is provided semantic information produced by static analysis, then tasks such as code summarization can be significantly improved. Li et al. [16] demonstrate that by carefully crafting questions using function-level behavior and summaries, LLMs can assist in removing false positives from a bug finding tool. Li et al. [17] also introduce a technique for combining static analysis using symbolic execution with LLMs to find *Use Before Initialization* (UBI) bugs, demonstrating that the LLM can be used to extract some program semantics and filter out false positives caused by the imprecision of the static analysis. Wen et al. [29] also demonstrate success in removing false positive warnings by using customized questions with domain knowledge from the Juliet [12] benchmark. LLMs have also been recently used to generate program invariants [19], including generating loop invariants [13] and subsequently ranking them using zero-shot prompting [4]. In contrast to all of the above, our work *interleaves* facts provided by both a static analysis and a LLM to improve the precision of an existing static analysis for error specification inference.

Program Analysis and Machine Learning. Seldon [5] is a tool using semi-supervised learning through building and solving a constraint system from information flow constraints for taint specification inference. InspectJS [9] is an approach for taint specification inference that uses manual modeling from CodeQL [11], inferred specifications using an adaptation of Seldon, a ranking strategy using embeddings, and manual user feedback. As discussed previously in relation to error specification inference, MLPEx [30] uses machine learning for error specification inference. While these approaches combined machine learning and traditional program analysis techniques to improve analysis results, our technique differs in that we are using LLMs and that both the static analysis and LLM-based inference results are interleaved throughout the entire analysis.

7 Conclusion

We have presented an approach for interleaving static program analysis and LLMs for the task of error specification inference. We have demonstrated that by providing program facts from the analysis of EESI to the LLM that it can infer error specifications correctly and in-turn can assist EESI to further learn new error specifications. We show this in our evaluation (Section 5), where our average recall grows from 52.55% to 77.83% and our F1-score improves from 0.612 to 0.804. Our evaluation also demonstrates a similar precision to the original static analysis, where the average only decreases from 86.67% to 85.12%.

Acknowledgments

This work was supported by the National Science Foundation under awards CCF-1750983, CCF-2119348 and CCF-2107592.

References

[1] Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Fundamental Approaches to Software*

Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5503), Marsha Chechik and Martin Wirsing (Eds.). Springer, 370–384. https://doi.org/10.1007/978-3-642-00593-0_25

[2] Toufique Ahmed and Premkumar T. Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 177:1–177:5. https://doi.org/10.1145/3551349.3559555

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and Amanda Askell et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 9164–9175. https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.614

[5] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 760–774. https://doi.org/10.1145/3314221.3314648

[6] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. 2019. Effective error-specification inference via domain-knowledge expansion. In *ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 466–476. https://doi.org/10.1145/3338906.3338960

[7] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 423–433. https://doi.org/10.1145/3236024.3236059

[8] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embeddings. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 430–431. https://doi.org/10.1145/3183440.3195042

[9] Saikat Dutta, Diego Garbervetsky, Shuvendu K. Lahiri, and Max Schäfer. 2022. InspectJS: Leveraging Code Similarity and User-Feedback for Effective Taint Specification Inference for JavaScript. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 165–174. https://doi.org/10.1109/ICSE-SEIP55303.2022.9794015

[10] Christof Fetzer, Karin Högstedt, and Pascal Felber. 2003. Automatic Detection and Masking of Non-Atomic Exception Handling. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*. IEEE Computer Society, 445–454. https://doi.org/10.1109/DSN.2003.1209955

[11] GitHub. 2021. *CodeQL*. https://codeql.github.com

[12] Frederick Boland Jr. and Paul Black. 2012. The Juliet 1.1 C/C++ and Java Test Suite. 45 (October 2012). https://doi.org/10.1109/MC.2012.345

[13] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. *CoRR* abs/2311.07948 (2023). https://doi.org/10.48550/ARXIV.2311.07948 arXiv:2311.07948

[14] Yuan Jochen Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 472–482. https://doi.org/10.1145/2970276.2970354

[15] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[16] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2107–2111. https://doi.org/10.1145/3611643.3613078

[17] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. (2024).

[18] OpenAI. 2023. GPT-4 Technical Report. https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[19] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520. https://proceedings.mlr.press/v202/pei23a.html

[20] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Model-Based Failure Analysis of Journaling File Systems. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*. IEEE Computer Society, 802–811. https://doi.org/10.1109/DSN.2005.65

[21] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, Andrew Herbert and Kenneth P. Birman (Eds.). ACM, 206–220. https://doi.org/10.1145/1095810.1095830

[22] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training. (2018). https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf

[23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[24] Martin Süßkraut and Christof Fetzer. 2006. Automatically Finding and Patching Bad Error Handling. In *Sixth European Dependable Computing Conference, EDCC 2006, Coimbra, Portugal, 18-20 October 2006*. IEEE Computer Society, 13–22. https://doi.org/10.1109/EDCC.2006.3

[25] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal

Bhargava, and Shruti Bhosale et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[27] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=1PL1NIMMrw

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/arXiv.2201.11903 arXiv:2201.11903 [cs.CL]

[29] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* (mar 2024). https://doi.org/10.1145/3653718

[30] Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. 2019. Generating precise error specifications for C: a zero shot learning approach. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 160:1–160:30. https://doi.org/10.1145/3360586

Received 2024-03-07; accepted 2024-04-19

A Better Approximation for Interleaved Dyck Reachability

Giovanna Kobus Conrado
Hong Kong University of Science and Technology
Hong Kong, Hong Kong
gkc@connect.ust.hk

Andreas Pavlogiannis
Aarhus University
Aarhus, Denmark
pavlogiannis@cs.au.dk

Abstract

Interleaved Dyck reachability is a standard, graph-based formulation of a plethora of static analyses that seek to be *context*- and *field*- sensitive, where each type of sensitivity is expressed via a CFL/Dyck language. Unfortunately, the problem is well-known to be undecidable in general, and as such, existing approaches resort to clever overapproximations. Recently, a mutual-refinement algorithm, that iteratively considers each of the two sensitivities in isolation until a fixpoint is reached, was shown to achieve high precision.

In this work we present a more precise approximation of interleaved Dyck reachability, by extending the mutual-refinement algorithm in two directions. First, we develop refined CFLs to express each type of sensitivity precisely, while simultaneously also lightly overapproximating the opposite type. Second, we apply the resulting algorithm on an on-demand basis, which effectively masks out imprecision incurred by parts of the graph that are irrelevant for the query at hand. Our experiments show that the new approach offers significantly higher precision than the vanilla mutual-refinement algorithm and other common baselines; for a particularly challenging benchmark, we report, on average, 51% of the reachable pairs compared to the most recent alternative.

CCS Concepts: • **Software and its engineering** → *Software verification and validation*; • **Theory of computation** → *Theory and algorithms for application domains.*

Keywords: CFL Reachability, Static Analysis, Graphs

ACM Reference Format:
Giovanna Kobus Conrado and Andreas Pavlogiannis. 2024. A Better Approximation for Interleaved Dyck Reachability. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3652588.3663318

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663318

1 Introduction

Static analyses are common techniques for inferring properties of a program prior to its execution, and are used to detect vulnerabilities and security issues, as well as for optimization purposes. One common approach is to create a graph $G = (V, E)$ that serves as a program model [10]. Nodes in G represent distinct, indivisible entities of the source code, such as variables, pointers, or code blocks, and edges capture direct relationships between such entities, such as control-flow between instructions or data-dependence between variables. Static analyses then reduce to the algorithmic problem of finding paths in G that satisfy certain properties.

Dyck Reachability. In order to increase the precision of the analysis, normally, not all paths in G are treated as valid approximations of program behavior. CFL/Dyck reachability is a popular, generic graph-based formulation of a plethora of static analyses following this scheme. Here, the edges of G are labeled with opening and closing parenthesis symbols, and reachability between nodes in G is only considered when witnessed by paths whose labels along the edges produce a properly-balanced parenthesis string [7, 20].

Parentheses have been commonly used to increase the sensitivity, and thus the precision, of the analysis, modeling two distinct features. In *context sensitivity*, parentheses capture sensitivity in calling contexts during whole-program analysis, by filtering out interprocedurally-invalid paths (see Fig. 1 for an example). This approach has found applications in a wide range of interprocedural static analyses, including data-flow and shape analysis [9, 12], type-based flow analysis [8], taint analysis [3], and data-dependence analysis [18].

Similarly, in *field sensitivity*, parentheses capture the flow of information through the fields of composite objects (see Fig. 2 for an example). This approach is frequently used in points-to analysis [5, 10].

Interleaved Dyck reachability. Naturally, for a high-precision analysis, we desire both context- and field-sensitivity. This leads to an *interleaved* reachability problem, where the label of paths witnessing reachability is well-balanced with respect to both parentheses and brackets, but the two types of symbols are *freely interleaved* (see Fig. 3).

```
fun next(x)
    return x+1

fun next2()
    int a ← 2
    int b ← next(a)

fun next3()
    int c ← 3
    int d ← next(c)
```

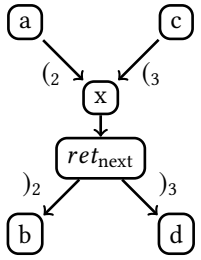


Figure 1. A data-dependence graph for whole-program analysis. Although a reaches d, there is no dependence of d on a, as the path $P_1 : a \rightsquigarrow d$ does not correspond to any interprocedurally-valid execution. The label of P_1 is $(_2)_3$ and thus not properly balanced, so Dyck reachability deems d not reachable from a. On the other hand, b is reachable from a via a path $P_2 : a \rightsquigarrow b$ with properly balanced labels $(_2)_2$, which witnesses a true dependence of b on a.

```
int a ← 2
int b ← 3
point p
p.x ← a
p.y ← b
int c ← p.x
int d ← p.y
```

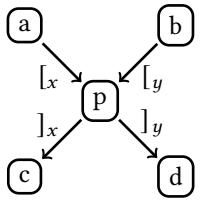


Figure 2. A data-dependence graph with composite objects and fields. We have that c depends on a, witnessed via a path labeled $[_x]_x$, capturing this dependence via the field x of p. On the other hand, d does not depend on a as the corresponding path has non-matching labels $[_x]_y$.

This approach has been used in many analysis applications, such as alias analysis [19], points-to analysis [13, 16], type-based flow analysis [6], taint analysis [3], and value-flow analysis [17], among others.

Unfortunately, interleaved Dyck reachability is known to be undecidable [4, 11], thus applications like the above resort to various overapproximations, such as approximating one of the languages with a regular language [15], using overapproximate algorithms based on linear conjunctive languages [21], and intersecting the reachability information obtained for each type of sensitivity independently via Synchronized Pushdown Systems [14]. The most recent development in this direction is a mutual-refinement algorithm developed in [2], which roughly performs an iterative application of the intersection method until a fixpoint, and was shown to be more precise than other baselines. Despite this progress, none of the existing approximations is very precise, and hence we consider this challenge in this work.

Our contribution. We develop a new approach for approximating interleaved Dyck reachability, by means of two extensions to the mutual-refinement algorithm of [2].

```
fun getX(p)
    int v ← p.x
    return v

fun f()
    int a ← 2
    int b ← 3
    point q
    q.x ← a
    q.y ← b
    int c ← getX(q)

fun g()
    int d ← 1
    point r
    r.x ← d
    int e ← getX(r)
```

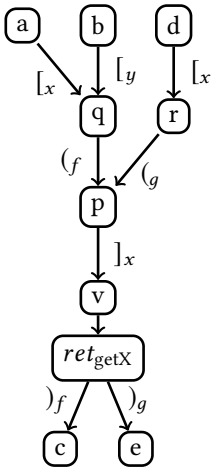


Figure 3. A data-dependence graph for whole-program analysis with composite objects and fields. Notice that b reaches c through a interprocedurally valid path labeled $[_y(_f]_x)_f$, but c does not depend on b since these variables do not flow correctly through their fields. Similarly, a reaches e through a path labeled $[_x(_f]_x)_g$. In this case the labels corresponding to the fields match, but the path is not interprocedurally valid. The path $P : a \rightsquigarrow c$ with label $[_x(_f]_x)_f$, on the other hand, is well-balanced both in terms of context and field labels, thus the value of c is truly dependent on a.

(1) We develop simple but stronger CFLs to express each type of sensitivity precisely, while simultaneously also overapproximating the opposite type using two efficient conditions. In effect, each iteration of the mutual refinement becomes more precise as it reasons simultaneously about both types of sensitivity (i.e., both parentheses and brackets), albeit being precise on only one of them.
(2) We apply the resulting algorithm on-demand, by considering each pair of nodes independently. This effectively filters out imprecision that is due to other nodes in the graph, thereby leading to more precise results.

An experimental evaluation of our approach on standard benchmarks shows that it is considerably more precise than the vanilla mutual-refinement and other common baselines.

2 Preliminaries

General notation. Given some $k \in \mathbb{N}$, we let $[k] = \{1, \cdots, k\}$. For a set S, $S(x)$ is a shorthand for $x \in S$. We let ϵ be the empty string. Given an alphabet Σ, a subset $A \subseteq \Sigma$ and a string $s \in \Sigma^*$, we denote by $s \restriction A$ the projection of s to A, i.e., the subsequence of s consisting of the letters in A.

Dyck language. Given a natural number $n_\alpha \in \mathbb{N}$, the Dyck language $D(\Sigma_\alpha)$ is the language over the alphabet $\Sigma_\alpha = \{\alpha_i, \overline{\alpha}_i\}_{i \in [n_\alpha]}$, consisting of strings contained in S, where S

is the smallest set satisfying the following three conditions.

$$S(\epsilon) \tag{1}$$

$$\forall i \in [n_\alpha],\ S(\alpha_i x \overline{\alpha}_i) \leftarrow S(x) \tag{2}$$

$$S(x_1 x_2) \leftarrow S(x_1), S(x_2) \tag{3}$$

In words, the Dyck language consists of well-nested labeled parenthesis; α_i and $\overline{\alpha}_i$ correspond, respectively, to labeled opening and closing parentheses. In our examples, we use $(_i$ for α_i and $)_i$ for $\overline{\alpha}_i$, to benefit readability.

Interleaved Dyck languages. Given two natural numbers $n_\alpha, n_\beta \in \mathbb{N}$, two disjoint alphabets $\Sigma_\alpha = \{\alpha_i, \overline{\alpha}_i\}_{i\in[n_\alpha]}$ and $\Sigma_\beta = \{\beta_i, \overline{\beta}_i\}_{i\in[n_\beta]}$ and the two corresponding Dyck languages $D(\Sigma_\alpha)$ and $D(\Sigma_\beta)$, a string s belongs to the interleaved language $D(\Sigma_\alpha) \odot D(\Sigma_\beta)$ if $s \downharpoonleft \Sigma_\alpha \in D(\Sigma_\alpha)$ and $s \downharpoonleft \Sigma_\beta \in D(\Sigma_\beta)$. In words, the substring of s formed by its characters in Σ_α (resp., Σ_β) is in $D(\Sigma_\alpha)$ (resp., $D(\Sigma_\beta)$).

Similarly to the interpretation of α_i and $\overline{\alpha}_i$ as parentheses, we interpret β_i and $\overline{\beta}_i$ as brackets $[_i$ and $]_i$. Then, the interleaved Dyck language $D(\Sigma_\alpha) \odot D(\Sigma_\beta)$ consists of strings that may contain both labeled parenthesis and labeled brackets, e.g. $[_2[_1(_5)_5(_2]_1)_2]_2$, and whose projection to parenthesis $((_5)_5(_2)_2)$ and brackets $([_2[_1]_1]_2)$ are properly balanced.

Language reachability. Consider a labeled directed graph $G = (V, E, \Sigma)$, where V is a set of nodes, Σ is an alphabet, and $E \subseteq V \times V \times (\Sigma \cup \{\epsilon\})$ is a set of partially labeled edges. A path $P\colon a \rightsquigarrow b$ in G is a sequence of edges $(u_1, v_1, c_1), \cdots, (u_k, v_k, c_k)$ such that $u_1 = a$, $v_k = b$, $\forall i \in [k], (u_i, v_i, c_i) \in E$ and $\forall i \in [k-1], v_i = u_{i+1}$. We let $\lambda(P) = c_1 c_2 \dots c_k$ be the label of P.

Given a language $L \subseteq \Sigma^*$ and a graph G, the L-reachability problem seeks to identify the set $\mathcal{R} = \{(a, b) \mid \exists P\colon a \rightsquigarrow b \in G \mid \lambda(P) \in L\}$. In static analysis, two prominent instances of this problem are *CFL/Dyck reachability*, where L is a CFL/Dyck language, and *interleaved Dyck reachability*, where $L = D(\Sigma_\alpha) \odot D(\Sigma_\beta)$ is an interleaved Dyck language. Although CFL/Dyck reachability is solved in cubic time [1, 20], interleaved Dyck reachability is undecidable [11]. As a way to circumvent this undecidability, various techniques focus on tractable (over)approximations of the reachability set $\mathcal{R}$, i.e., they compute a set $\mathcal{R}_{\text{over}} \supseteq \mathcal{R}$. Naturally, the goal of each approximation $\mathcal{R}_{\text{over}}$ is to minimize the number of false positives, i.e., the quantity $|\mathcal{R}_{\text{over}}| - |\mathcal{R}|$.

3 Baseline Approximations

Here we review the standard approximation approaches.

3.1 Regularization

A natural approximation to the problem is *regularization*. It relies on the fact that the intersection of a context-free language and a regular language is a context-free language. Thus, we can approximate, say, $D(\Sigma_\beta)$, through a regular language $R(\Sigma_\beta) \supseteq D(\Sigma_\beta)$, and perform standard reachability with the CFL $D_{\text{reg}}(\Sigma_\alpha, \Sigma_\beta) = D(\Sigma_\alpha) \cap R(\Sigma_\beta)$.

A standard approach for this is overapproximation via k-contexts. The idea is that we keep track of a bounded stack for $D(\Sigma_\beta)$ of $\leq k$ open brackets, making sure that each is properly matched when we encounter a close bracket. $R(\Sigma_\beta)$ also accepts all strings when the stack exceeds the bound k.

Fig. 4 shows a finite state automaton for $R(\Sigma_\beta)$ for $k = 1$, adapted from [21]. Observe that with n_β types of brackets, the size of the automaton is $\Theta({n_\beta}^k)$, i.e., the automaton grows exponentially quickly in k. For this reason, practical applications rely on $k = 1$ or $k = 2$, to keep the complexity blowup modest (though it is still considerable).

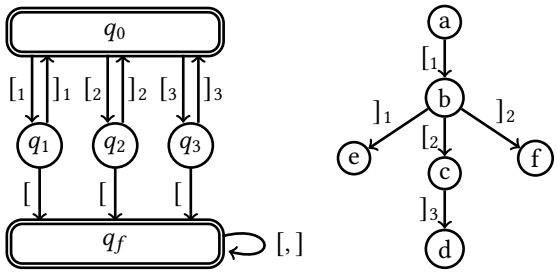


Figure 4. *(Left):* An automaton for $R(\Sigma_\beta)$ with $n_\beta = 3$ and $k = 1$. Unlabeled brackets represent brackets of any label. *(Right):* A graph with three paths $P_1\colon a \rightsquigarrow c$, $P_2\colon a \rightsquigarrow d$ and $P_3\colon a \rightsquigarrow e$, whose label is in $R(\Sigma_\beta)$.

3.2 Intersection

Another simple and common approach is the *intersection method*, which was recently popularized under the term Synchronized Pushdown Systems [14].

Consider a string $s \in D(\Sigma_\alpha) \odot D(\Sigma_\beta)$, thus $s \downharpoonleft \Sigma_\alpha \in D(\Sigma_\alpha)$. We will construct a new grammar $D'(\Sigma_\alpha, \Sigma_\beta)$ that "ignores" characters in Σ_β. $D'(\Sigma_\alpha, \Sigma_\beta)$ will be defined similarly to $D(\Sigma_\alpha)$, with the addition of the following rule.

$$\forall i \in [n_\beta], S(\beta_i) \text{ and } S(\overline{\beta}_i) \tag{4}$$

We can construct $D'(\Sigma_\beta, \Sigma_\alpha)$ analogously, with the addition of the following rule to $D(\Sigma_\beta)$.

$$\forall i \in [n_\alpha], S(\alpha_i) \text{ and } S(\overline{\alpha}_i) \tag{5}$$

Observe that $D(\Sigma_\alpha) \odot D(\Sigma_\beta) \subseteq D'(\Sigma_\alpha, \Sigma_\beta) \cap D'(\Sigma_\beta, \Sigma_\alpha)$. Moreover, each of $D'(\Sigma_\alpha, \Sigma_\beta)$ and $D'(\Sigma_\beta, \Sigma_\alpha)$ is a CFL, thus the corresponding reachability problem is solvable in cubic time. Thus, we can compute $\mathcal{R}_\alpha$ (resp., $\mathcal{R}_\beta$) as the result of $D'(\Sigma_\alpha, \Sigma_\beta)$-reachability (resp., $D'(\Sigma_\beta, \Sigma_\alpha)$-reachability) on G, and we have that $\mathcal{R}_{\text{over}} = \mathcal{R}_\alpha \cap \mathcal{R}_\beta$ is an overapproximation of $\mathcal{R}$, as desired. See Fig. 5 for an illustration.

3.3 Mutual Refinement

The mutual refinement algorithm introduced in [2] builds upon intersection, and is based on the following observation. Let again $\mathcal{R}_\alpha$ (resp., $\mathcal{R}_\beta$) be the result of $D'(\Sigma_\alpha, \Sigma_\beta)$-reachability (resp., $D'(\Sigma_\beta, \Sigma_\alpha)$-reachability), and assume that

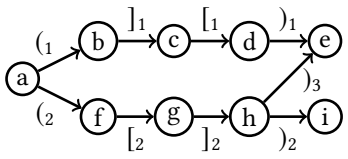


Figure 5. There are two paths $P_1, P_2 \colon a \rightsquigarrow e$, with $\lambda(P_1) = (_1[_1)_1 \in D'(\Sigma_\alpha, \Sigma_\beta)$ and $\lambda(P_2) = (_2[_2]_2)_3 \in D'(\Sigma_\beta, \Sigma_\alpha)$, thus the intersection algorithm incorrectly returns that e is reachable from a. On the other hand, there is a path $P_3 \colon a \rightsquigarrow i$ with $\lambda(P_3) = (_2[_2]_2)_2 \in D'(\Sigma_\alpha, \Sigma_\beta) \cap D'(\Sigma_\beta, \Sigma_\alpha)$, thus the algorithm correctly identifies that i is reachable from a.

for two nodes a, b, we have $(a, b) \in \mathcal{R}_\alpha \cap \mathcal{R}_\beta$. If the set of paths witnessing $(a, b) \in \mathcal{R}_\alpha$ is disjoint from the set of paths witnessing $(a, b) \in \mathcal{R}_\beta$, then we can conclude that $(a, b) \notin \mathcal{R}$. The mutual-refinement algorithm exploits this insight in a recursive way, as follows (see Fig. 6).

(1) Compute $\mathcal{R}_\alpha$ and identify the set of edges $E' \subseteq E$ that are contained in some path P such that $L(P) \in D'(\Sigma_\alpha, \Sigma_\beta)$. This can be done through a simple modification of the standard CFL-reachability algorithm.
(2) Analogously, compute $\mathcal{R}_\beta$ and identify the set of edges $E'' \subseteq E$, formed by the edges that participate in some path P such that $L(P) \in D'(\Sigma_\beta, \Sigma_\alpha)$.
(3) If $E' \cap E'' = E$, halt and return $\mathcal{R}_\alpha \cap \mathcal{R}_\beta$. Otherwise, let $G = (V, E' \cap E'', \Sigma_\alpha \cup \Sigma_\beta)$ and return to Item (1).

Since an edge must be removed at any repetition of Item (1), the algorithm converges. In fact, in real world instances this algorithm is shown to converge quickly.

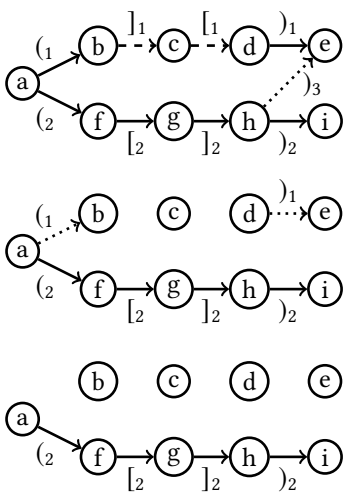


Figure 6. Executing the mutual refinement algorithm on the graph of Fig. 5. Dashed (resp., dotted) lines represent the edges in some path exclusively in $D'(\Sigma_\alpha, \Sigma_\beta)$ (resp., $D'(\Sigma_\beta, \Sigma_\alpha)$), while solid lines represent edges in both. Node e is no longer incorrectly identified as reachable from a, as edges from paths P_1 and P_2 have been removed. Node i is still identified as reachable from a, as P_3 has been kept intact.

4 Our Approach

Our approach extends the mutual-refinement algorithm in two distinct directions. First, we introduce new grammars for each refinement step: instead of working with the language $D'(\Sigma_\alpha, \Sigma_\beta)$ (i.e., the one that faithfully captures the parentheses in Σ_α but completely ignores the brackets in Σ_β), we introduce a new language that simultaneously performs a simple regular approximation of the brackets. Second, we run the resulting algorithm in an on-demand way (i.e., for computing reachability between a given pair of nodes), which effectively reduces imprecision by removing parts of the graph that are provably not relevant for the query at hand.

4.1 Stronger Grammars

We focus on extensions of $D'(\Sigma_\alpha, \Sigma_\beta)$, as the ones for $D'(\Sigma_\beta, \Sigma_\alpha)$ are analogous. The focus is on adding a only a constant amount of rules to $D'(\Sigma_\alpha, \Sigma_\beta)$, and trying to discard cases of unreachable pairs of nodes that the usual mutual refinement algorithm does not recognize as unreachable.

Parity conditions. Observe that for any $s \in D(\Sigma_\beta)$, we have that $|s|$ is even, as every opening bracket is matched to a unique closing bracket. We can thus extend $D'(\Sigma_\alpha, \Sigma_\beta)$ to $D_{\text{par}}(\Sigma_\alpha, \Sigma_\beta)$, the latter being identical to the former but excluding strings for which the number of brackets is odd. This is achieved by the following CFG, where S_0 is the start symbol (see Fig. 7 for an illustration).

$$S_0(\epsilon) \tag{6}$$

$$\forall i \in [n_\alpha], p \in \{0, 1\}, S_p(\alpha_i x \overline{\alpha}_i) \leftarrow S_p(x) \tag{7}$$

$$\forall i \in [n_\beta], S_1(\beta_i) \text{ and } S_1(\overline{\beta}_i) \tag{8}$$

$$\forall p_1, p_2 \in \{0, 1\}, S_{p_1 \oplus p_2}(x_1 x_2) \leftarrow S_{p_1}(x_1), S_{p_2}(x_2) \tag{9}$$

Where $\oplus$ represents "exclusive or".

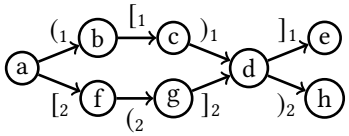


Figure 7. There are paths $P_1, P_2 \colon a \rightsquigarrow d$ with $\lambda(P_1) = (_1[_1)_1 \in D'(\Sigma_\alpha, \Sigma_\beta)$ and $\lambda(P_2) = [_2(_2]_2 \in D'(\Sigma_\beta, \Sigma_\alpha)$. Both paths are retained after mutual refinement, as they contribute to the reachability $a \rightsquigarrow e$ and $a \rightsquigarrow h$. Hence, the algorithm incorrectly concludes that a reaches d. However, neither of $\lambda(P_1)$, $\lambda(P_2)$ is in $D_{\text{par}}(\Sigma_\alpha, \Sigma_\beta)$ or $D_{\text{par}}(\Sigma_\beta, \Sigma_\alpha)$.

Extended parity conditions. The above parity condition extends naturally: for any string $s \in D(\Sigma_\beta)$, not only is $|s|$ even, but the number of characters of any particular label type is also even. Unfortunately, keeping track of the parity of every single label type increases the grammar size exponentially. To circumvent this, we choose a small number k, and arbitrarily partition the bracket labels into k groups. Now, we require that the number of characters in each group is even.

We thus arrive at the language $D_{\text{par}^k}(\Sigma_\alpha, \Sigma_\beta)$, parameterized by k. The corresponding grammars are generalizations of the grammar above for the parity condition —we omit their formal description here for brevity.

Valid endpoints. Recall that the regularization method (Section 3.1) makes an explicit representation of the stack of size up to k. Unfortunately, this blows up the grammar significantly: even for $k = 1$ the grammar increases by a factor n_β. Here we introduce a simpler but effective condition. We use a regular language $R'(\Sigma_\beta)$ that recognizes strings that start with β_i and end with $\overline{\beta}_j$, for some $i, j \in [n_\beta]$, expressed by the automaton in Fig. 8.

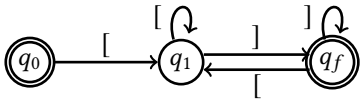


Figure 8. A finite-state automaton for $R'(\Sigma_\beta)$. The unlabeled brackets are used as shorthand for a bracket of any label.

The choice of this language is twofold: this condition eliminates many paths that none of the previous approaches do, as shown in Fig. 9, while it is succinctly expressible. Combining these with the extended parity conditions, and generating $D_{\text{par}^k}(\Sigma_\beta, \Sigma_\alpha)$ and $R'(\Sigma_\alpha)$ analogously, our new languages for the mutual refinement algorithm are as follows.

$$D^+_{\text{par}^k}(\Sigma_\alpha, \Sigma_\beta) = D_{\text{par}^k}(\Sigma_\alpha, \Sigma_\beta) \cap R'(\Sigma_\beta) \tag{10}$$

$$D^+_{\text{par}^k}(\Sigma_\beta, \Sigma_\alpha) = D_{\text{par}^k}(\Sigma_\beta, \Sigma_\alpha) \cap R'(\Sigma_\alpha) \tag{11}$$

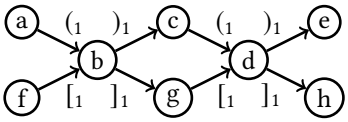


Figure 9. There are two paths $P_1, P_2 \colon b \rightsquigarrow d$ with $\lambda(P_1) =)_1(_1$ and $\lambda(P_2) =]_1[_1$. Mutual refinement does not remove any of their edges, as they are contained in valid paths $a \rightsquigarrow e$ and $f \rightsquigarrow h$, respectively. We have $\lambda(P_1) \in D_{\text{par}}(\Sigma_\beta, \Sigma_\alpha)$ and $\lambda(P_2) \in D_{\text{par}}(\Sigma_\alpha, \Sigma_\beta)$, thus d is incorrectly deemed reachable from b. Notice, though, that $)_1(_1 \notin R'(\Sigma_\alpha)$ and $]_1[_1 \notin R'(\Sigma_\beta)$, thus $\lambda(P_1) \notin D^+_{\text{par}^k}(\Sigma_\alpha, \Sigma_\beta)$ and $\lambda(P_2) \notin D^+_{\text{par}^k}(\Sigma_\beta, \Sigma_\alpha)$, correctly concluding that d is not reachable from b.

4.2 On-Demand Analysis

Up until now we have sought to approximate the whole set of reachable node pairs $\mathcal{R}$. A natural question is whether we can increase the precision by focusing on the on-demand question, i.e., we ask whether some fixed node v is reachable from some fixed node u.

We tackle this problem by a small modification to the mutual refinement algorithm. In Item (1), $E' \subseteq E$ is defined as the set of edges that are contained in some path P such that $L(P) \in D'(\Sigma_\alpha, \Sigma_\beta)$. We now, instead, consider only the edges that are contained in some path $P \colon u \rightsquigarrow v$ for which $L(P) \in D'(\Sigma_\alpha, \Sigma_\beta)$. Similarly for Item (2). This avoids many cases where the original algorithm fails to discard reachability, like the reachability $a \rightsquigarrow d$ in Fig. 7 and from $b \rightsquigarrow d$ in Fig. 9. To solve the reachability problem for all pairs of nodes, we repeatedly solve the on-demand problem. Though this is generally slower, it leads to increased precision.

5 Experiments

Here we report on an implementation and experimental evaluation of our approach for approximating interleaved Dyck reachability [1] (Section 4). We use two sets of benchmarks from the literature: a taint analysis for android [3] and a value-flow analysis for LLVM [17].

Methods. We compare the following methods:

- Regularization: as described in Section 3.1.
- Intersection: as described in Section 3.2.
- Mutual Refinement: as described in Section 3.3.
- Stronger Grammar: mutual refinement method using the grammar defined in Section 4.1, with $k = 2$.
- On-Demand: mutual refinement method using the grammar defined in Section 4.1, with $k = 2$, executed on-demand for all pairs of vertices in the graph, as described in Section 4.2.

Note that for the four last methods, each is strictly more precise than the previous.

Underapproximation. Since we cannot compare our results to the unknown value $|\mathcal{R}|$, we use a simple, standard *underapproximation* of $\mathcal{R}$ instead: the paths identified by the CFL-reachability algorithm applied to the language $D(\Sigma_\alpha \cup \Sigma_\beta)$. This identifies every pair of nodes of which there is a path whose label is properly-balanced in parenthesis and brackets, without any interleaving. For example, $(_1[_1]_1)_1 \in D(\Sigma_\alpha \cup \Sigma_\beta)$ but $(_1[_1)_1]_1 \notin D(\Sigma_\alpha \cup \Sigma_\beta)$. We call this set of reachable pairs $\mathcal{R}_{\text{under}}$. In our experimental results, we report $|\mathcal{R}_{\text{over}}| - |\mathcal{R}_{\text{under}}|$, where $\mathcal{R}_{\text{over}}$ is the overapproximation computed by the corresponding method. This corresponds to an overapproximation of the false positives.

Setup and implementation. Taint analysis is modeled through a data-dependence graph with parentheses modelling context sensitivity and brackets modelling field sensitivity. The task is to identify all pairs of variables $(u, v), u \neq v$ such that the value of u flows into v. See [3] for details.

Value-flow analysis is modeled through a data-dependence graph with parenthesis modelling context sensitivity and brackets modelling memory stores/loads (in this case $n_\beta = 1$, i.e., there is only one type of bracket). We seek to identify all pairs of variables $(u, v), u \neq v$ such that the value of u flows into v. Furthermore, we only consider paths of which the first edge is of type $[_1$ and the last edge is of type $]_1$, problem [3]. This is done in regularization by incorporating

[1] github.com/kobusgiovanna/interleaved-dyck-approximation

this condition in the automaton and in the other methods by taking into account an additional grammar that identifies strings $s = [_1 s']_1$ for some string $s' \in (\Sigma_\alpha \cup \Sigma_\beta)^*$. See [17] for details.

Our benchmark graphs were taken from [2]. For the taint analysis benchmark we selected all graphs with $\leq$ 1800 nodes, for a total of 11 benchmarks. For the value-flow analysis, we selected all graphs with $\leq 10^5$ nodes, for a total of 8 benchmarks. These were selected so none of the algorithms would timeout. We implemented all the methods in Go, and executed them on a MacBook Air with an Apple M1 chip and 8GB of RAM. We set a timeout of one hour for each algorithm and each benchmark.

Results. The results for the taint analysis benchmark are shown in Fig. 10 and Table 1. We observe that Mutual Refinement is always more precise than both Regularization and Intersection, which aligns with the conclusions made in [2]. However, we see that our methods are always more precise than Mutual Refinement, and often by a large amount. Overall, the geometric average of $|\mathcal{R}_{\text{over}}|/|\mathcal{R}_{\text{under}}|$ for Mutual Refinement and our method is 2.22 and 1.14, respectively. This number is 1.24 if we only consider our method's grammar improvement and do not execute the algorithm on-demand. This means that, on average, Mutual Refinement has 55% potential false positive rate, while our method has 12% potential false positive rate. We note that since our method is theoretically guaranteed to be more precise than Mutual Refinement, all pairs reported by Mutual Refinement but not our method are indeed false positives. The geometric mean of the overapproximations $|\mathcal{R}_{\text{over}}^{\text{Our method}}|/|\mathcal{R}_{\text{over}}^{\text{Mutual Refinement}}|$ is 0.51.

Finally, for the value-flow benchmark, both Mutual Refinement and our method obtained a tight result, as we had $\mathcal{R}_{\text{over}} = \mathcal{R}_{\text{under}}$, for all but one benchmark (povray), in which Mutual Refinement over-reported one path, which our method discarded. This means that this benchmark set is not particularly challenging (though, to our knowledge, this tightness had not been observed before).

Performance Considerations. Since the increase in grammar size is constant, the mutual refinement method using our stronger grammar is only slower than the original mutual refinement algorithm by some constant factor. In our grammar as defined in Section 4.1, incorporating the valid endpoints condition, the number of predicates increases by a factor of 4×2^k. The number of rules will be proportional to the square of the number of predicates.

We ran one instance of CFL reachability for each one of the 11 benchmarks shown in Figure 10 for both the original Dyck grammar, as used in Section 3.2 and Section 3.3, and for our grammar, as described in Section 4.1, with $k = 2$. Our grammar ran, on average, 36 times slower.

It is worth noting that this work mainly seeks to simply minimize $|\mathcal{R}_{\text{over}}| - |\mathcal{R}_{\text{under}}|$. More practical applications may

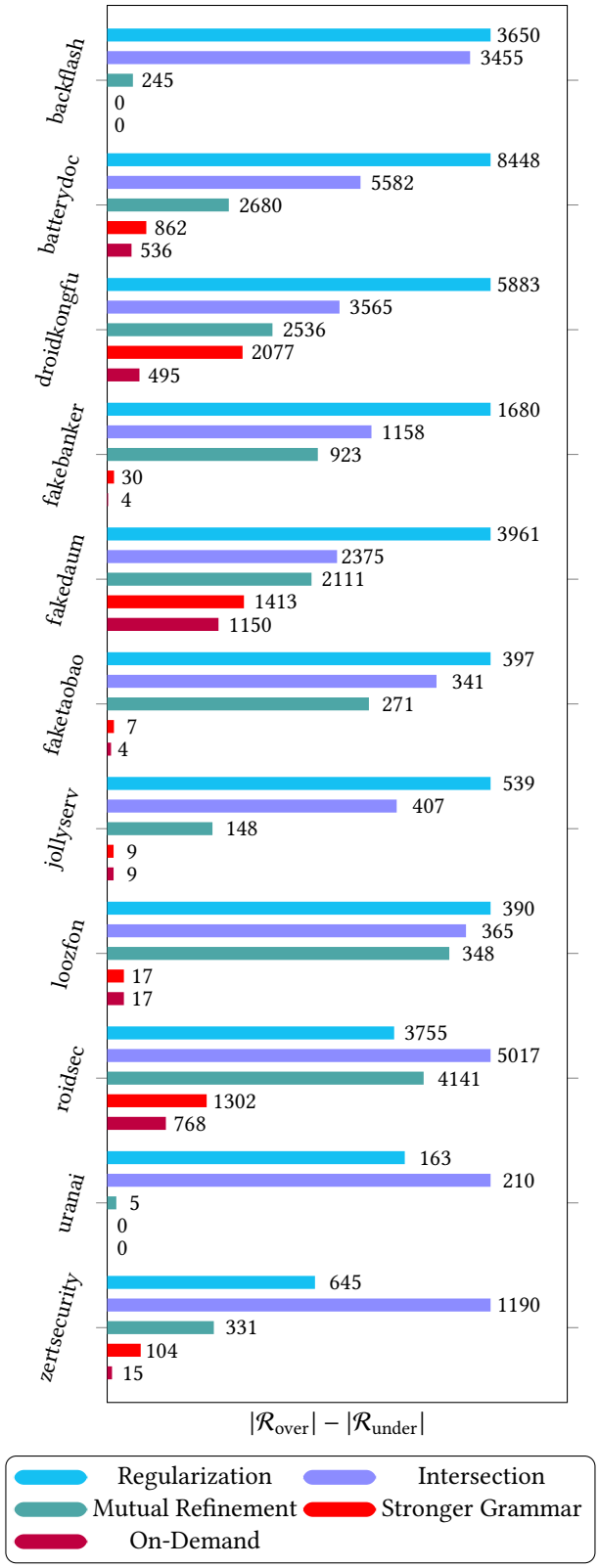


Figure 10. Experimental results for the taint analysis. Column numbers indicate $|\mathcal{R}_{\text{over}}| - |\mathcal{R}_{\text{under}}|$. Bar lengths are normalized by the worst-performing method. *Stronger Grammar* and *On-Demand* correspond to our methods.

Table 1. Experimental results for taint analysis. We compare $|\mathcal{R}_{under}|$ with the three best-performing methods, $|\mathcal{R}^{\text{MR}}_{\text{over}}|$, $|\mathcal{R}^{\text{SG}}_{\text{over}}|$ and $|\mathcal{R}^{\text{OD}}_{\text{over}}|$, which correspond to *Mutual Refinement*, *Stronger Grammar* and *On-demand*, respectively.

Benchmark	$\lvert\mathcal{R}_{under}\rvert$	$\lvert\mathcal{R}^{\text{MR}}_{\text{over}}\rvert$	$\lvert\mathcal{R}^{\text{SG}}_{\text{over}}\rvert$	$\lvert\mathcal{R}^{\text{OD}}_{\text{over}}\rvert$
backflash	2625	2870	2625	2625
batterydoc	2804	5484	3666	3340
droidkongfu	2906	5442	4983	3401
fakebanker	249	1172	279	253
fakedaum	1132	3243	2545	2282
faketaobao	57	328	64	61
jollyserv	155	303	164	164
loozfon	76	424	93	93
roidsec	12284	16425	13586	13052
uranai	143	148	143	143
zertsecurity	779	1110	883	794

consider trade-offs between runtime and false positive rate. For example, running CFL reachability on the grammar with $k = 1$ instead of $k = 2$ for the benchmarks from Figure 10 takes, on average, only 28% of the time, while only increasing the potential false positive rate from 12.39% to 12.74%.

The on-demand version runs much faster than the general one for a particular pair of nodes, in particular because when querying for some pair (u, v), removing vertices that are not reachable from u or do not reach v often greatly reduces the graph size. This algorithm, on the other hand, runs significantly slower than the alternatives for all-pairs reachability, as it must execute the mutual refinement algorithm many times. This can be optimized in ways by, for example, only running the algorithm in the pairs of which the reachability is not known, i.e., $\mathcal{R}_{\text{over}} \setminus \mathcal{R}_{\text{under}}$ for some under and overapproximation.

6 Conclusion

Interleaved Dyck reachability is a framework used in a plethora of static analyses, but undecidable in general, thus existing work focuses on approximations. In this work, we expanded upon a recent, mutual-refinement approximation in two distinct ways. Our experimentation with standard benchmarks shows that our new method is often significantly more precise than the best alternative.

Acknowledgments

This work was partially supported by a research grant (VIL42117) from VILLUM FONDEN. G.K. Conrado was partially supported by the Hong Kong PhD Fellowship Scheme (HKPFS).

References

[1] Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. *SIGPLAN Not.* 43, 1 (jan 2008), 159–169. https://doi.org/10.1145/1328897.1328460

[2] Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *Static Analysis: 30th International Symposium, SAS 2023, Cascais, Portugal, October 22–24, 2023, Proceedings* (Lisbon, Portugal). Springer-Verlag, Berlin, Heidelberg, 231–258. https://doi.org/10.1007/978-3-031-44245-2_12

[3] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/2771783.2771803

[4] Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The decidability and complexity of interleaved bidirected Dyck reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 12 (jan 2022), 26 pages. https://doi.org/10.1145/3498673

[5] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive points-to analysis: is it worth it?. In *Proceedings of the 15th International Conference on Compiler Construction* (Vienna, Austria) *(CC'06)*. Springer-Verlag, Berlin, Heidelberg, 47–64. https://doi.org/10.1007/11688839_5

[6] Ana Milanova. 2020. FlowCFL: Generalized Type-Based Reachability Analysis: Graph Reduction and Equivalence of CFL-based and Type-Based Reachability. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–29. https://doi.org/10.1145/3428246

[7] Andreas Pavlogiannis. 2022. CFL/Dyck Reachability: An Algorithmic Perspective. *ACM SIGLOG News* 9, 4 (Oct. 2022), 5–25. https://doi.org/10.1145/3583660.3583664

[8] Jakob Rehof and Manuel Fähndrich. 2001. Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 54–66.

[9] Thomas Reps. 1995. Shape Analysis As a Generalized Path Problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '95)*. ACM, 1–11.

[10] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11 (1998), 701–726. https://doi.org/10.1016/S0950-5849(98)00093-7

[11] Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (jan 2000), 162–186. https://doi.org/10.1145/345099.345137

[12] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. ACM, New York, NY, USA.

[13] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 264–274. https://doi.org/10.1145/2259016.2259050

[14] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (jan 2019), 29 pages. https://doi.org/10.1145/3290361

[15] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *SIGPLAN Not.* 41, 6 (jun 2006), 387–400. https://doi.org/10.1145/1133255.1134027

[16] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

[17] 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 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[18] Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. 2017. Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 880–908.

[19] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-Driven Context-Sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 155–165. https://doi.org/10.1145/2001420.2001440

[20] Mihalis Yannakakis. 1990. Graph-theoretic Methods in Database Theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 230–242.

[21] Qirun Zhang and Zhendong Su. 2017. Context-Sensitive Data-Dependence Analysis via Linear Conjunctive Language Reachability. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 344–358. https://doi.org/10.1145/3093333.3009848

Received 05-MAR-2024; accepted 2023-04-19

Interactive Source-to-Source Optimizations Validated using Static Resource Analysis

Guillaume Bertholon
Arthur Charguéraud
Thomas Kœhler
Begatim Bytyqi
Damien Rouhling
Inria & Université de Strasbourg, CNRS, ICube
Strasbourg, France

Abstract

Developments in hardware have delivered formidable computing power. Yet, the increased hardware complexity has made it a real challenge to develop software that exploits the hardware to its full potential. Numerous approaches have been explored to help programmers turn naive code into high-performance code, finely tuned for the targeted hardware. However, these approaches have inherent limitations, and it remains common practice for programmers seeking maximal performance to follow the tedious and error-prone route of writing optimized code by hand.

This paper presents OptiTrust, an interactive source-to-source optimization framework that operates on general-purpose C code. The programmer develops a script describing a series of code transformations. The framework provides continuous feedback in the form of human-readable *diff*s over conventional C code. OptiTrust supports advanced code transformations, including transformations exploited by the state-of-the-art DSL tools Halide and TVM, and transformations beyond the reach of existing tools. OptiTrust also supports user-defined transformations, as well as defining complex transformations by composition of simpler transformations. Crucially, to check the validity of code transformations, OptiTrust leverages a *static resource analysis* in a simplified form of Separation Logic. Starting from user-provided annotations on functions and loops, our analysis deduces precise resource usage throughout the code.

CCS Concepts: • **Software and its engineering** → **Software performance**; **Development frameworks and environments**; • **Theory of computation** → *Separation logic.*

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663320

Keywords: High performance code, Source-to-source optimization, Separation logic

ACM Reference Format:
Guillaume Bertholon, Arthur Charguéraud, Thomas Kœhler, Begatim Bytyqi, and Damien Rouhling. 2024. Interactive Source-to-Source Optimizations Validated using Static Resource Analysis. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3652588.3663320

1 Introduction

1.1 Motivation

Performance matters in numerous fields of computer science, and in particular in applications from machine learning, computer graphics, and numerical simulation. Massive speedups can be achieved by fine-tuning the code to best exploit the available hardware [15]. Between a naive implementation and an optimized implementation, it is common to see a speedup of the order of 50×—on a single core. For many applications, the code can then be accelerated further by one or two orders of magnitude by refining the code to exploit multicore parallelism or GPUs.

Yet, producing high performance code is hard. Over the past decades, nontrivial mechanisms with subtle interactions were integrated into hardware architectures. Reasoning about performance requires reasoning about the effects of multiple levels of caches, the limitations of memory bandwidth, the intricate rules of atomic operations, and the diversity of vector instructions (SIMD). These aspects and their interactions make it challenging to build cost models. For example, the cost of a memory access can range from one CPU cycle to hundreds of CPU cycles, depending on whether the corresponding data is already in cache. In the general case, accurately modeling cache behavior requires a deep understanding of the algorithm and hardware at play.

Accurately predicting runtime behavior is challenging for expert programmers, and appears beyond the capabilities of automated tools. Therefore, compilers struggle to navigate the exponentially large search space of all possible code

candidates [24], resorting to best effort heuristics, and often failing to produce competitive code [3].

Today, it remains common practice in industry for programmers to write optimized code *by hand* [1, 9]. However, manual code optimization is unsatisfactory for at least three reasons. First, manually implementing optimized code is time-consuming. Second, the optimized code is hard to maintain through hardware and software evolutions. Third, the rewriting process is error-prone: not only every manual code edition might introduce a bug, but the code complexity also increases, especially when introducing parallelism. These three factors are exacerbated by the fact that optimizations typically make code size grow by an order of magnitude (for example, the optimized code for our following matrix multiplication case study is 7× bigger).

In summary, neither fully automatic nor fully manual approaches are satisfying for generating high performance code. Both machine automation and human insight are needed in the optimization process.

1.2 Contribution

This paper introduces OptiTrust, the first interactive optimization framework that operates on general-purpose C code and that supports and validates state-of-the-art optimizations. OptiTrust is open-source and available at the URL: https://github.com/charguer/optitrust.

In OptiTrust, the user starts from an unoptimized C code, and develops a *transformation script* describing a series of optimization steps. Each step consists of an invocation of a specific transformation at specified *targets*. OptiTrust provides an expressive target mechanism for describing, in a concise and robust manner, one or several code locations. On any step of the transformation script, the user can press a key shortcut to view the *diff* associated with that step, in the form of a comparison between two human-readable C programs. Concretely, a transformation script consists of an OCaml program linked against the OptiTrust library.

To ensure that the user applies only semantic-preserving transformations, OptiTrust performs validity checks that leverage our *static resource analysis*, which concretely takes the form of a type checking algorithm, in a type system featuring linear resources. This type system may be thought of as a variant of the Rust type system, or as a scaled down version of Separation Logic [20]. Our resource-based system aims to be similar in spirit to RefinedC [22], a Separation Logic-based type system for C code, even though we have not implemented all the features of RefinedC yet.

For type-checking resources, functions and loops need to be equipped with *contracts* describing their resource usage. These contracts may be inserted either directly as no-op annotations in the C source code, or they may be inserted by dedicated commands as part of the transformation script. OptiTrust is able to automatically infer simple loop contracts, thus not all loops need to be annotated manually. Every OptiTrust transformation takes care of updating contracts in order to reflect changes in the code. In other words, a well-typed program remains well-typed after a transformation.

Currently, OptiTrust only automates the application of transformations and the checking of their validity, but we also plan to explore future work to guide the user towards useful optimizations.

Next, we illustrate how OptiTrust works through an example optimization script.

```
void mm(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ~> Matrix2(m, p), B ~> Matrix2(p, n)");
  __modifies("C ~> Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n -> &C[i][j] ~> Cell");
    for (int j = 0; j < n; j++) {
      __xmodifies("&C[i][j] ~> Cell");
      float sum = 0.0f;
      for (int k = 0; k < p; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
    }
  }
}
void mm1024(float* C, float* A, float* B) {
  __reads("A ~> Matrix2(1024, 1024), "
          "B ~> Matrix2(1024, 1024)");
  __modifies("C ~> Matrix2(1024, 1024)");
  mm(C, A, B, 1024, 1024, 1024);
}
```

Listing 1. Unoptimized matrix multiplication. The function `mm` multiplies the matrices `A` and `B` and stores the result in `C`. The function `mm1024` specializes input sizes to 1024. We write `A[i][k]` instead of `A[MINDEX2(m, p, i, k)]`, for conciseness. In the future, we plan to leverage a mechanism for automatically propagating size information.

2 OptiTrust by Example

In this section we present the features of OptiTrust through an example: optimizing matrix multiplication. The aim is to produce similar code as a reference TVM *schedule* that was written by an expert targeting Intel CPUs.[1] TVM is an industrial-strength domain-specific compiler for machine learning.

Annotated Code. We start from the C code presented in Listing 1: a naive, unoptimized implementation of matrix multiplication. To use OptiTrust, we annotate the code with resource contracts, which follow a double-underscore prefix.

The `mm` function reads a matrix `A` of size $m \times p$, reads a matrix `B` of size $p \times n$, and modifies a matrix `C` of size $m \times n$. This is explicitly described by the function contract and its `__reads` /`__modifies` clauses. Each clause mentions a set of resources

[1] https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html

```
!! Function.inline_def [cFunDef "mm"];
let tile (id, tile_size) = Loop.tile (int tile_size)
   ~index:("b" ^ id) ~bound:TileDivides [cFor id] in
!! List.iter tile [("i", 32); ("j", 32); ("k", 4)];
!! Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
   [cPlusEq ()];
!! Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
   ~indep:["bi"; "i"] [cArrayRead "B"];
!! Matrix.stack_copy ~var:"sum" ~copy_var:"s" ~copy_dims:1
   [cFor ~body:[cPlusEq ()] "k"];
!! Omp.simd [cFor ~body:[cPlusEq ()] "j"];
!! Omp.parallel_for [cFunBody ""; cStrict; cFor ""];
!! Loop.unroll [cFor ~body:[cPlusEq ()] "k"];
```

Listing 2. OptiTrust script for optimizing `mm1024`.

separated by ",". For example, the resource `A ⇝ Matrix2(m, p)` specifies that the matrix at address `A` in memory has size $m \times p$ *and* gives permission to access this matrix: effectively the permission over every individual cell of the matrix, that is: `for i in 0..m → for j in 0..p → &A[i][j] ⇝ Cell`.

Each iteration of the loops with index `i` and `j` modifies a separate group of cells from matrix `C`. Loop contracts contain two kinds of clauses: clauses prefixed by "x" are used to describe resources *exclusive* to one iteration, and clauses prefixed by "s" to describe resources *shared* by all iterations. For example, in the loop with index `j`, the clause `__xmodifies("&C[i][j] ⇝ Cell")` indicates that only iteration `j` can modify the `j`-th cell of the `i`-th row of the matrix `C`. By contrast, the clause `__smodifies("for j in 0..n → &C[i][j] ⇝ Cell")` would have indicated that all loop iterations can modify the `n` cells of the `i`-th row of the matrix `C`. Similarly, `__xreads("&A[i][j] ⇝ Cell")` indicates that the `j`-th iteration is the only one that reads the `j`-th cell of `i`-th row of the matrix `A`, and `__sreads("A ⇝ Matrix2(m, p)")` indicates that every iteration may read any of the `m` × `p` cells of the matrix `A`. By default, resources are shared by all iterations, so in this code the following clause is inferred for the three loops: `__sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)")`.

Transformation Script. To apply optimizations, we write an OptiTrust script in OCaml, as shown in Listing 2. For the reader not familiar with OCaml, `f x y` denote the call of `f` on the arguments `x` and `y`; the symbol `~` is used to provide optional (or named) arguments; `[x; y; z]` denotes a list; `(x, y, z)` denotes a tuple; `s1 ^ s2` denotes a string concatenation; and `let f x = e1 in e2` introduces a local function f. The transformation script calls a series of transformations that are functions taking various arguments, including a *target* as last argument.

The optimizations applied by this script improve data locality, parallelism, and specialize the matrix sizes. The script consists of 8 transformation steps, developed interactively: with the cursor on a line starting with `!!`, we can press (e.g.) "F6" in the VSCode editor to visualize the *diff* associated with the transformation on that line. All intermediate versions of the code consist of human-readable, executable C code. The `!!` operator is used purely to enable interactivity and early termination. Additionally, a complete transformation report can be generated and explored.[2]

Targets. As mentioned earlier, transformations take targets as parameters, that describe code locations. A target consists of a list of constraints (prefixed by "c") that is satisfied by code paths that go through nodes satisfying each constraint, in the given order. For example, `cFunDef "mm"` requires visiting a function definition with the name `"mm"`, and `cFor id` requires visiting a for loop over an index with the name `id`. Targets may also include special modifiers (the ones that make a target relative are prefixed by "t"). For example, `tBefore` allows targeting the interstice before an instruction. As another example, `cStrict` controls the depth: `[cFunBody ""; cStrict; cFor ""]` targets for loops over any index name that appear immediately within a function body with any name, as opposed to being nested within other constructs. Targets may also be given as arguments to constraints, for example, `cFor ~body:[cPlusEq ()] "k"` requires visiting a for loop over an index with the name `"k"`, whose body also contains a `+=` operation.

Transformations. The script from Listing 2 calls functions from the OptiTrust library called transformations. We use `Function.inline_def` to inline the definition of `mm` into the `mm1024` function that specializes $m = n = p = 1024$. We use `Loop.tile`, `Loop.reorder_at` and `Loop.hoist_expr` to apply loop transformations improving data locality and exposing new dimensions for parallelization. We are free to use any OCaml feature, here defining the local `tile` function to iteratively apply it over a list: tiling the loops over `i` and `j` by 32, and the loop over `k` by 4 to create outer loops with indices `bi`, `bj`, and `bk`. `Loop.hoist_expr` creates a new temporary matrix with name `pB` to store values of matrix `B` using a better layout, something that actually requires manually changing the reference code (the *algorithm*) in a tool like TVM. While the last target argument locates which expression to hoist, the `~dest` target argument additionally describes where to hoist it. We locally promote an array to the stack using `Matrix.stack_copy`, introducing efficient `memcpy` operations, and allowing for the use of SIMD vector registers. We use `simd` and `parallel_for` to add OpenMP pragmas for multithreading and vectorizing a number of the newly created loops. Finally, we unroll the loop over `k` to help the downstream C compiler recognize instruction-level parallelism.

Combined Transformations. As witnessed by the detailed report generated by OptiTrust[2], our relatively concise optimization script for matrix multiplication actually involves a fair number of *basic* transformation steps happening under the hood. For example, `Loop.reorder_at` is a

[2] https://files.inria.fr/optitrust/soap24/matmul.html

$$\overline{\begin{array}{l}\textbf{for}_{\ \gamma}\ i\ \textbf{in}\ r_i\ \{\\ \quad t_1; \Delta_1\\ \quad \Gamma_2\ t_2; \Delta_2\\ \}\end{array}} \longmapsto \overline{\begin{array}{l}\textbf{for}_{\ \gamma_1}\ i\ \textbf{in}\ r_i\ \{\\ \quad t_1;\\ \}\\ \textbf{for}_{\ \gamma_2}\ i\ \textbf{in}\ r_i\ \{\\ \quad t_2;\\ \}\end{array}}$$

$$\textbf{correct if:} \begin{cases} i \text{ fresh in } \gamma.\text{shrd} \\ \gamma.\text{shrd.modifies} \cap \text{notRO}(\Delta_1) \cap \Delta_2 = \emptyset \\ \gamma.\text{shrd.modifies} \cap \text{notRO}(\Delta_2) \cap \Delta_1 = \emptyset \end{cases}$$

annotations produced on the right-hand-side:

$$\gamma_1 \equiv \begin{cases} \text{loop.vars} \equiv \gamma.\text{loop.vars} \\ \text{shrd} \equiv \gamma.\text{shrd} \cap \Delta_1 \\ \text{excl.pre} \equiv \gamma.\text{excl.pre} \\ \text{excl.post} \equiv R \end{cases} \qquad \gamma_2 \equiv \begin{cases} \text{loop.vars} \equiv \gamma.\text{loop.vars} \\ \text{shrd} \equiv \gamma.\text{shrd} \cap \Delta_2 \\ \text{excl.pre} \equiv R \\ \text{excl.post} \equiv \gamma.\text{excl.post} \end{cases}$$

$$R \equiv \text{cleanup}(\Gamma_2 - \gamma.\text{shrd})$$

Figure 1. `Loop.fission` transformation and its validation.

combined transformation that takes as argument a specific instruction, and takes a description of how we would like to order the loops around it. The transformation is defined recursively, "bringing down" the desired loops, from innermost to outermost. The call to `reorder_at` in our script involves 4 loop swaps, 6 loop fissions, and 2 hoist operations. In particular, the hoist operations result in turning the `sum` variable local to the inner loop into a 2D-array of values declared in an outer loop. The loop fissions isolate the initialization and the reads into this 2D-array into separate loop nests.

Typing Algorithm. While running the transformation script, every intermediate code is typechecked with our resource type system. Internally, function contract clauses such as `__reads` and `__modifies` that appear in the initial code are desugared into lower-level pre- and post- conditions. For linear resources, preconditions consume resources whereas postconditions produce resources. In this low-level representation, following standard separation logic, read-only permissions are encoded using *fractions* [6, 14]. Loop contracts are also desugared, into lower-level loop contracts (written γ). Low-level loop contracts separate resources in groups: γ.shrd.modifies and γ.shrd.reads describe resources modified and read by all iterations; γ.excl.pre and γ.excl.post describe resources that are exclusively consumed and produced by one iteration. Low-level loop contracts also bind logical variables in γ.loop.vars that the loop abstracts over (typically, the fraction variables of the read-only permissions).

The code is then typed according to the provided contracts by proceeding top to bottom, in a syntax-directed way that does not require difficult inferences. The typing context consists of the resources available at a given program point, where resources may be fully available, available in read-only mode (i.e., only a fraction is available) or available in "uninit" mode. Resources available in *uninit* mode cannot be read from before writing to them, which is useful to model when memory values for a resource are irrelevant.

To type a function body, the typing context is initialized with the precondition, and the final typing context is checked to imply the postcondition. Intuitively, the body of a function is given access to its consumed resources, and must return access to its produced resources. For every program point, our typing algorithm not only computes the resources available as typing contexts (written Γ), but also the local resource usage (written Δ).

Validity Checks. Leveraging the resource typing information, OptiTrust checks that each transformation applied by the script (Listing 2) preserves the semantics. The validity of a *combined* transformation is derived from the validity of all the *basic* transformations that it leverages. The validity of a *basic* transformation is verified by the OCaml implementation of that transformation, which is responsible for checking sufficient conditions under which it preserves semantics. To ensure that every intermediate code type-checks, each transformation must also maintain annotations such as those provided in the initial code (Listing 1). Simple examples are the `Loop.simd` and `Loop.parallel_for` transformations, that are correct if the annotations on the targeted loop captures the absence of interference: the γ.shrd.modifies resource set of the loop contract γ is empty.

A more complex example is the validity of `Loop.fission`, depicted in Figure 1. The transformation is described on the internal imperative λ-calculus representation of OptiTrust, where `for` loops are simplified to iterate over ranges. Annotations on the left (in green) represent resource information consumed by the transformation, and annotations on the right (in orange) represent resource information produced by the transformation.

Intuitively, loop fission is correct if the resources modified by t_1 and t_2 do not interfere across iterations. For γ.excl resources, there is no interference because each iteration is independent. For γ.shrd resources, we check for interference using Δ_1 and Δ_2: if t_1 modifies one resource from γ.shrd, then t_2 must not use this same resource; symmetrically, if t_2 modifies a resource, then t_1 must not use it. Note, however, that t_1 and t_2 may both read the same resource.

On top of checking for the correctness condition, the fission transformation must also synthesize loop contracts for the new loops, so that the resulting code still type-checks. For shrd resources, we simply project the subsets of γ.shrd resources used by t_1 and t_2. For excl resources, we preserve the previous pre- and post-conditions, but need to synthesize a new middle-point (R) corresponding to the iteration-exclusive resources available between t_1 and t_2. R is computed by

```
float* pB = (float*)malloc(sizeof(float[32][256][4][32]));
#pragma omp parallel for
for (int bj = 0; bj < 32; bj++) {
  for (int bk = 0; bk < 256; bk++) {
    for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
        pB[32768 * bj + 128 * bk + 32 * k + j] =
          B[1024 * (4 * bk + k) + 32 * bj + j]; }}}}
#pragma omp parallel for
for (int bi = 0; bi < 32; bi++) {
  for (int bj = 0; bj < 32; bj++) {
    float* sum = (float*)malloc(sizeof(float[32][32]));
    for (int i = 0; i < 32; i++) {
      for (int j = 0; j < 32; j++) {
        sum[32 * i + j] = 0.; }}
    for (int bk = 0; bk < 256; bk++) {
      for (int i = 0; i < 32; i++) {
        float s[32];
        memcpy(s, &sum[32 * i], sizeof(float[32]));
#pragma omp simd
        for (int j = 0; j < 32; j++) { // this loop is for k = 0
          s[j] += A[1024 * (32 * bi + i) + 4 * bk + 0] *
                  pB[32768 * bj + 128 * bk + 32 * 0 + j]; }
        // [...] similar unrolling, not shown, for k = 1, 2, 3
        memcpy(&sum[32 * i], s, sizeof(float[32])); }}
    for (int i = 0; i < 32; i++) {
      for (int j = 0; j < 32; j++) {
        C[1024 * (32*bi + i) + 32*bj + j] = sum[32*i + j]; }}
// [...] free instructions, not shown
```

Listing 3. Optimized C code produced by the OptiTrust script for `mm1024`. This code has similar structure and achieves similar performance as the reference output of TVM.

subtracting the shared resources (γ.shrd) from the resources available between t_1 and t_2 (Γ_2), and for technical scoping reasons, performing a final "cleanup".

Final Optimized Code. Listing 3 shows the optimized C code produced by our script. First, we checked that this output code matches the structural optimizations from the reference TVM case study. Note that TVM directly targets LLVM IR, and does not produce easily readable C code.

Second, we checked the performance. We benchmarked our code against TVM's code on a 4-core Intel i7-8665U CPU with AVX2 support. Both codes have similar runtime, corresponding to a speedup of 150× over the naive code.[3]

3 Comparison to Related Work

Now that we have seen how OptiTrust works by example, let us introduce a number of qualitative properties, before reviewing related tools for semi-automatic code optimization and finally explaining why OptiTrust achieves a unique combination of features.

- **Generality**: How large is the domain of applicability of the tool? In particular, is it restricted to a domain-specific language?
- **Expressiveness**: How advanced are the code transformations supported by the tool? Is it possible to express state-of-the-art code optimizations?
- **Control**: How much control over the final code is given to the user by the tool? In particular, is there a monolithic code generation stage?
- **Feedback**: Does the tool provide easily readable intermediate code after each transformation?
- **Composability**: Is it possible to define transformations as the composition of existing transformations? Can transformations be higher-order, i.e., parameterized by other transformations?
- **Extensibility** of transformations: Does the tool facilitate defining custom transformations that are not expressible as the composition of built-in ones?
- **Trustworthiness**: Does the tool ensure that user-requested transformations preserve the semantics of the code? Can it moreover provide mechanized proofs?

3.1 Related Work

Halide [19] is an industrial-strength domain-specific compiler for image processing. Halide popularized the idea of separating an *algorithm* describing what to compute from a *schedule* describing how to optimize the computation. This separation makes it easy to try different schedules. TVM [8] is a tool directly inspired by Halide, but tuned for applications to machine learning. Halide and TVM are inherently limited to their domain-specific languages. They do not support higher-order composition of transformations, and are not extensible [3, 18]. Moreover, understanding their output is difficult as the applied transformations are not detailed to the user. Interactive scheduling systems have been proposed to mitigate this difficulty [13].

Elevate [11] is a functional language for describing *optimization strategies* as composition of simple *rewrite rules*. Advanced optimizations from TVM and Halide can be reproduced using Elevate. One key benefit is extensibility: adding rewrite rules is much easier than changing complex and monolithic compilation passes [18]. Elevate strategies are applied on programs expressed in a functional array language named Rise, followed by compilation to imperative code. The use of a functional array language greatly simplifies rewriting, however it restricts applicability and makes controlling imperative aspects difficult (e.g. memory reuse).

Exo [12] is an imperative DSL embedded in Python, geared towards the development of high-performance libraries for specialized hardware. It is restricted to static control programs with linear integer arithmetic. Exo programs can be optimized by applying a series of source-to-source transformations. These transformations are described using a Python script, with simple string-based patterns for targeting code points. The user can add custom transformations, possibly defined by composition; higher-order composition seems possible but has not yet been demonstrated.

[3] We obtain a 90th percentile runtime of 9.4ms over 200 benchmark runs, and compare it to the 90th percentile of the naive code. Besides, the OptiTrust median runtime is slightly faster than the TVM median runtime.

Table 1. Overview of user-guided tools for high-performance code generation.

	Halide/TVM	Elevate+Rise	Exo	Clay/LoopOpt	ATL	Alpinist	Clava+LARA
Generality	◔	◐	◐	◐	◐	◔	●
Expressiveness	●	●	●	◐	◐	◔	◔
Control	◐	◐	◔	◐	◐	●	●
Feedback	◐	◐	●	●	◐	●	◐
Composability	○	●	◐	◐	●	○	●
Extensibility	○	●	●	○	●	●	●
Trustworthiness	◐	◐	◐	◐	●	●	○

Clay [2] is a framework to assist in the optimization of loop nests that can be described in the *polyhedral model* [10]. The polyhedral model only covers a specific class of loop transformations, with restriction over the code contained in the loop bodies, however it has proved extremely powerful for optimizing code falling in that fragment. Clay provides a decomposition of polyhedral optimizations as a sequence of basic transformations with integer arguments. The corresponding transformation script can then be customized by the programmer. Clint [26] adds visual manipulation of polyhedral schedules through interactive 2D diagrams. LoopOpt [7] provides an interactive interface that helps users design optimization sequences (featuring unrolling, tiling, interchange, and reverse of iteration order) that can be bound in a declarative fashion to loop nests satisfying specific patterns.

ATL [16] is a purely functional array language for expressing Halide-style programs. Its particularity is to be embedded into the Coq proof assistant. ATL programs can be transformed through the application of rewrite rules expressed as Coq theorems. With this approach, transformations are inherently accompanied by machine-checked proofs of correctness. The set of rules includes expressive transformations beyond the scope of Halide, and can be extended by the user. Once optimized, ATL programs are then compiled into imperative C code. Like Rise, generality and control are restricted by the functional array language nature of ATL.

Alpinist [21] is a *pragma*-based tool for optimizing GPU-level, array-based code, able to apply basic transformations such as loop tiling, loop unrolling, data prefetching, matrix linearization, and kernel fusion. The key characteristic of Alpinist is that it operates over code formally verified using the VerCors framework [5]. Concretely, Alpinist transforms not only the code but also its formal annotations. If Alpinist were to leverage transformation scripts instead of pragmas, it might be possible to chain and compose transformations; yet, this possibility remains to be demonstrated.

Clava [4] is a general-purpose C++ source-to-source analysis and transformation framework implemented in Java. The framework has been instantiated mainly for code instrumentation purpose and auto-tuning of parameters. Clava can also be used in conjunction with a DSL called LARA [23] for optimizing specific programs. LARA allows expressing user-guided transformations by combining declarative queries over the AST and imperative invocations of transformations, with the option to embed JavaScript code. The application paper on the Pegasus tool [17] illustrates this approach on loop tiling and interchange operations.

Table 1 summarizes the properties of the existing approaches, highlighting their diversity. The table is sorted by increasing generality. For the tools considered, this generality is negatively correlated with expressiveness, i.e., with how advanced the supported transformations are. Regarding generality, only Clava supports operating on general C code, yet provides absolutely no guarantees on semantics preservation. For each property considered, at least two tools show strengths on that property (above half score). However, even if we leave out the ambition of achieving mechanized proofs, each tool considered shows weaknesses on at least two properties (half score or less).

3.2 The Unique Features of OptiTrust

When considering the aforementioned criteria and tools, OptiTrust achieves a unique combination of features.

Generality. OptiTrust is generally applicable to optimizing C code. The code must parse using Clang, the parser of LLVM. The fragments of code that the user wishes to alter must moreover type-check in our resource type system. At the time of writing, we support only core features of the C language: sequences, loops, conditionals, functions, local and global variables, arrays, and structs. There is, however, no inherent limitation: OptiTrust could presumably be extended to support nearly all the C language (we do not plan to handle general goto's). Our resource type system currently only allows describing simple *shapes* of data structures, and does not yet allow specifying the stored values. That said, we have been planing to extend our implementation to a full-featured Separation Logic similar to RefinedC [22]. In summary, OptiTrust in its current form does not yet demonstrate full generality, however it has been designed towards that goal.

Expressiveness. The combination of three ingredients allows OptiTrust's users to generate their desired optimized code: (1) the use of a transformation script for describing a sequence of transformations; (2) the use of a *target* mechanism, allowing to precisely pinpoint where transformations should

be applied; (3) the availability of a catalog of general-purpose transformations, whose composition enables altering the code with a lot of flexibility.

Let us summarize the transformations currently supported in OptiTrust. For instruction-level transformations, we support: function inlining, constant propagation, instruction reordering, switching between stack and heap allocation, and basic arithmetic simplifications. For control-flow transformations, we support: loop interchange, loop tiling, loop fission, loop fusion, loop-invariant code motion, loop unrolling, loop deletion and loop splitting. For data layout transformations, we support: interchange of dimensions of an array, and array tiling. There are many more useful transformations for which we are working out sufficient correctness conditions.

Certain transformations may require nontrivial checks. For example, array tiling requires the tile size to divide the array size, and loop splitting requires arithmetic inequalities to hold. OptiTrust currently only validates simple conditions; in the future, more complex conditions could be handled using either SMT solvers or interactive theorem provers.

Control. Transformation scripts in OptiTrust empower the user with very fine-grained control over how the code should be transformed. A challenge is to allow for concise scripts. To that end, OptiTrust provides high-level *combined* transformations, effectively recipes for combining the *basic* transformations provided by OptiTrust. Section 2 presented the example of `Loop.reorder_at`, which attempts, using a combination of fission, hoist, and swap operations, to create a reordered loop nest around a specified instruction. Overall, the use of *combined* transformations allows for reasonably concise transformation scripts, with the user's intention being described at a relatively high level of abstraction. The user stays in control and can freely mix the use of concise abstractions and precise fine-tuning transformations.

Feedback. For each step in the transformation script, OptiTrust delivers feedback in the form of human-readable C code. The user usually only needs to read the *diff* against the previous code. Interestingly, OptiTrust also records a trace that allows investigating all the substeps triggered by a *combined* transformation. This information is critically useful when the result of a high-level transformation does not match the user's intention. Besides, a key feature of OptiTrust is its fast feedback loop. The production of fast, human-readable feedback in a system with significant control is reminiscent of interactive proof assistants, and of the aforementioned ATL tool [16].

Composability. OptiTrust transformation scripts are expressed as OCaml programs, and each transformation from our library consists of an OCaml function. Because OCaml is a full-featured programming language, OptiTrust users may define additional transformations at will by combining existing transformations. User-defined transformations may query the abstract syntax tree (AST) that describes the C code, allowing to perform analyses before deciding what transformations to apply. Furthermore, because OCaml is a higher-order programming language, transformation can take other transformations as argument. We use this programming pattern for example to customize the arithmetic simplifications to be performed after certain transformations.

Extensibility. If the user needs a transformation that is not expressible as a combination of transformations from the OptiTrust library, a custom transformation can be devised. Because OptiTrust does not rely on heuristics, adding a new transformation to OptiTrust does not impact in any way the behavior of existing scripts. To define relatively simple custom transformations, OptiTrust provides a term-rewriting facility based on pattern matching. For more complicated transformations, one can follow the patterns employed in the OptiTrust's library. For all custom transformations, it is the programmer's responsibility to work out the criteria under which applying the transformation preserves the semantics of the code, and to adapt contracts if necessary in order to produce well-typed code.

Trustworthiness. Compilers are well-known to be incredibly hard to get 100% correct [25]. Like compilers, optimization tools are highly subject to bugs. OptiTrust mitigates the risks of producing incorrect code in two ways.

Firstly, we instrumented OptiTrust to generate *reports* when processing transformation scripts. A report takes the form of a standalone HTML page, which contains the diff for every transformation step (and sub-steps). Such a report can be thoroughly scrutinized by a third-party reviewer.

Secondly, we have organized the OptiTrust code base so as to isolate the implementation of the *basic* transformations, which consists of transformations that directly modify the AST. Only basic transformations need to be trusted. We have been careful to systematically minimize the complexity of the interface and of the implementation of our basic transformations. All other transformations—the *combined* transformations—are *not* part of the trusted computing base.

4 Conclusion

To conclude, OptiTrust is general-purpose, takes as input C code, supports interactive development of transformation scripts, and produces at every step readable C code semantically equivalent to the original code. Our case study demonstrates that a reasonably concise OptiTrust script achieves the same performance as an expert-written TVM schedule.

In future work, we plan to integrate support for arbitrary logical assertions. Following the approach of Alpinist [21], OptiTrust will transform not only code, but also the expressive logical annotations that decorate the code. Leveraging on support for logical assertions, we look forward to completing more challenging case studies.

References

[1] Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, Helen Karatza, Christoph Kessler, Peter Kilpatrick, Hugo Martiniano, Ilias Mavridis, Sabri Pllana, Ana Respício, José Simão, Luís Veiga, and Ari Visa. 2020. Programming languages for data-Intensive HPC applications: A systematic mapping study. *Parallel Comput.* 91 (2020), 102584. https://doi.org/10.1016/j.parco.2019.102584

[2] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

[3] Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 177–183. https://doi.org/10.1145/3317550.3321441

[4] João Bispo and João M. P. Cardoso. 2020. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12 (2020), 100565. https://doi.org/10.1016/J.SOFTX.2020.100565

[5] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10510)*, Nadia Polikarpova and Steve A. Schneider (Eds.). Springer, 102–110. https://doi.org/10.1007/978-3-319-66845-1_7

[6] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[7] Lorenzo Chelini, Martin Kong, Tobias Grosser, and Henk Corporaal. 2021. LoopOpt: Declarative Transformations Made Easy. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems* (Eindhoven, Netherlands) *(SCOPES '21)*. Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/3493229.3493301

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[9] Thomas M Evans, Andrew Siegel, Erik W Draeger, Jack Deslippe, Marianne M Francois, Timothy C Germann, William E Hart, and Daniel F Martin. 2022. A survey of software implementations used by application codes in the Exascale Computing Project. *The International Journal of High Performance Computing Applications* 36, 1 (2022), 5–12. https://doi.org/10.1177/10943420211028940

[10] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem: one dimensional time. *Intl. Journal of Parallel Programming* 21, 5 (october 1992), 313–348. https://doi.org/10.1007/BF01407835

[11] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (aug 2020), 29 pages. https://doi.org/10.1145/3408974

[12] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

[13] Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2021. Guided Optimization for Image Processing Pipelines. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576341

[14] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018). https://doi.org/10.1017/S0956796818000151

[15] Vasilios Kelefouras and Georgios Keramidas. 2022. Design and Implementation of 2D Convolution on x86/x64 Processors. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3800–3815. https://doi.org/10.1109/TPDS.2022.3171471

[16] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

[17] Pedro Pinto, João Bispo, João M. P. Cardoso, Jorge G. Barbosa, Davide Gadioli, Gianluca Palermo, Jan Martinovič, Martin Golasowski, Kateřina Slaninová, Radim Cmar, and Cristina Silvano. 2022. Pegasus: Performance Engineering for Software Applications Targeting HPC Systems. *IEEE Transactions on Software Engineering* 48, 3 (2022), 732–754. https://doi.org/10.1109/TSE.2020.3001257

[18] Jonathan Ragan-Kelley. 2023. Technical Perspective: Reconsidering the Design of User-Schedulable Languages. *Commun. ACM* 66, 3 (feb 2023), 88. https://doi.org/10.1145/3580370

[19] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Conference on Programming Language Design and Implementation*. 12 pages. https://doi.org/10.1145/2491956.2462176

[20] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[21] Ömer Sakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. 2022. Alpinist: An Annotation-Aware GPU Program Optimizer. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13244)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 332–352. https://doi.org/10.1007/978-3-030-99527-0_18

[22] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. https://doi.org/10.1145/3453483.3454036

[23] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M.P. Cardoso, Carlo Cavazzoni, Daniele Cesarini, Stefano Cherubin, Federico Ficarelli, Davide Gadioli, Martin Golasowski, Antonio Libri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Kateřina Slaninová, and Emanuele Vitali. 2019. The ANTAREX domain specific language for high performance computing. *Microprocessors and Microsystems* 68 (2019), 58–73. https://doi.org/10.1016/j.micpro.2019.05.005

[24] Manish Vachharajani, Neil Vachharajani, David I. August, and Spyridon Triantafyllis. 2003. Compiler Optimization-Space Exploration. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA, USA, 204. https://doi.org/10.1109/CGO.2003.1191546

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Conference on Programming Language Design and Implementation* (San Jose, California, USA). Association for Computing Machinery, 12 pages. https://doi.org/10.1145/1993498.1993532

[26] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2018. Visual Program Manipulation in the Polyhedral Model. *ACM Trans. Archit. Code Optim.* 15, 1, Article 16 (mar 2018), 25 pages. https://doi.org/10.1145/3177961

Received 01-MAR-2024; accepted 2023-04-19

When to Stop Going Down the Rabbit Hole: Taming Context-Sensitivity on the Fly

Julian Erhard
LMU München and Technische Universität München
Garching, Germany
julian.erhard@tum.de

Johanna Franziska Schinabeck
Technische Universität München
Garching, Germany
johanna.schinabeck@tum.de

Michael Schwarz
Technische Universität München
Garching, Germany
m.schwarz@tum.de

Helmut Seidl
Technische Universität München
Garching, Germany
helmut.seidl@tum.de

Abstract

Context-sensitive analysis of programs containing recursive procedures may be expensive, in particular, when using expressive domains, rendering the set of possible contexts large or even infinite. Here, we present a general framework for context-sensitivity that allows formalizing not only known approaches such as full contexts or call strings but also combinations of these. We propose three generic lifters in this framework to bound the number of encountered contexts on the fly. These lifters are implemented within the abstract interpreter Goblint and compared to existing approaches to context-sensitivity on the SV-COMP benchmark suite. On a subset of recursive benchmarks, all proposed lifters manage to reduce the number of stack overflows and timeouts compared to a full context approach, with one of them improving the number of correct verdicts by 31% and showing promising results on the considered SV-COMP categories.

CCS Concepts: • **Theory of computation → Program analysis**.

Keywords: context-sensitive analysis, static program analysis, software verification, abstract interpretation

ACM Reference Format:
Julian Erhard, Johanna Franziska Schinabeck, Michael Schwarz, and Helmut Seidl. 2024. When to Stop Going Down the Rabbit Hole: Taming Context-Sensitivity on the Fly. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652588.3663321

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663321

1 Introduction

Context-sensitive static analysis, while often considered a key ingredient for precise results, is known to suffer from scalability issues, in particular when the set of contexts is large or infinite. This issue is even more pressing for recursive programs, where analysis may even fail to terminate, namely when infinitely many contexts are *encountered* and the analysis descends into an infinite rabbit hole of contexts. To remedy this without entirely giving up on expressive contexts, approaches to limit the amount of context-sensitivity inside call-cycles have been proposed [57]. These approaches require a pre-analysis computing a call graph to identify (mutually) recursive procedures. Similarly, selective context-sensitive approaches prioritize a set of program procedures and analyze others context-insensitively [10, 14, 16, 24–27, 35, 36, 49, 50, 56]. Again, these approaches rely on pre-analyses. For programs with dynamic procedure calls, such pre-analyses may either be imprecise, unsound or expensive. Thus, we are interested in approaches that limit the amount of encountered contexts *on the fly*, i.e., at the time the analysis is performed. At the same time, the implementation of the approach should not require changes to existing analyses. We thus present lifters that allow taming the context-sensitivity of existing analyses without sacrificing precision on most non-recursive programs.

This paper makes the following contributions:

- We present a unifying framework for context-sensitivity using side-effecting constraint systems (Section 2) and call string- and abstract-state-based contexts as instances.
- We propose three approaches to tame context-sensitivity *on the fly* (Sections 4 to 6), in particular for recursive programs. Section 3 details how to generically apply these approaches to existing analyses.
- We implement these techniques in the Goblint abstract interpreter and evaluate them on the SV-COMP benchmark suite (Section 7) and some larger programs.

Section 8 discusses related work and Section 9 concludes.

2 Preliminaries

Consider a given interprocedural abstract interpretation-based analysis. For a given program, each procedure is described by a control-flow graph with nodes from the finite set $\mathcal{N}$, distinct from the nodes of other procedures, and edges $\mathcal{E} = \mathcal{N} \times \mathcal{A} \times \mathcal{N}$, where $\mathcal{A}$ are actions from the programming language. Program execution starts from a procedure `main`. The analysis relates program nodes, enhanced with additional information from some non-empty set of contexts $\mathbb{C}$, to abstract program states $\mathbb{D}$. The abstract domain $\mathbb{D}$ is a complete lattice, equipped with a partial order $\sqsubseteq$ and operations $\sqcup$ and $\sqcap$, as well as $\top$ and $\bot$ elements. For domains with infinite ascending chains, we require a widening operator ∇ [8]. We call pairs $[v, c] \in \mathcal{N} \times \mathbb{C}$ *location unknowns*, for which we want to compute invariants. Together with a disjoint set of other *auxiliary unknowns*, e.g., for collecting flow- and context-insensitive information, they form the set of *unknowns* $\mathcal{X}$. The analysis then gives rise to constraints whose solutions are mappings from unknowns to abstract states. In our formulation, the constraints for call sites cause contributions both to the unknown corresponding to the intraprocedural control-flow successor and to an unknown corresponding to the start node of the called function, where the latter is called a *side-effect* of the constraint. Such *side-effecting* constraint systems [2] have been used, e.g., to combine flow- and context-sensitive analyses of locals with flow-insensitive analyses of globals in multithreaded programs [43, 44, 55], as well as to handle intricate non-local control-flow [42]. Our constraint systems collect restrictions of mappings $\eta : \mathcal{X} \to \mathbb{D}$, and are of the form:

$$\eta\ [\mathsf{st}_{\mathtt{main}}, c_0] \sqsupseteq d_0$$
$$\eta, \eta\ [v, c] \sqsupseteq [\![(u, a, v), c]\!]^\sharp_A\ \eta, \quad \text{for } (u, a, v) \in \mathcal{E}, c \in \mathbb{C}$$

where the ordering relation $\sqsubseteq$ is extended pointwise from $\mathbb{D}$ to maps and tuples. The first constraint provides some initial abstract state d_0 to the start node of function `main`, $\mathsf{st}_{\mathtt{main}}$, in some initial context c_0. The second type of constraint provides the abstract semantics for the program edges (u, a, v) together with contexts c. The function $[\![\cdot]\!]^\sharp_A : (\mathcal{E} \times \mathbb{C}) \to (\mathcal{X} \to \mathbb{D}) \to ((\mathcal{X} \to \mathbb{D}) \times \mathbb{D})$ defines how the abstract state is transformed from one program node to the next in the same context. It returns a pair, where the second element is the contribution to the unknown corresponding to the sink node of the edge, and the first element collects the encountered side-effects, i.e., all contributions to other unknowns. A mapping η satisfying all constraints is called *solution*.

For calls to a procedure f at some edge $e = (u, a, v)$, the right-hand side has the following form:

$$\begin{aligned}
&[\![e, c]\!]^\sharp_A\ \eta = \textbf{let } d = \eta\ [u, c] \textbf{ in}\\
&\quad \textbf{let } c_f, d_f = \mathsf{enter}^\sharp_{f,e}\ c\ d \textbf{ in}\\
&\quad \textbf{let } d' = \mathsf{combine}^\sharp\ d\ (\eta\ [\mathsf{ret}_f, c_f]) \textbf{ in}\\
&\quad (\{[\mathsf{st}_f, c_f] \mapsto d_f\}, d')
\end{aligned}$$

First, the abstract state $d \in \mathbb{D}$ for the preceding program node u in context c is retrieved. Then, $\mathsf{enter}^\sharp_{f,e}$ handles, e.g., the assignments of actuals to formals and the removal of caller locals from the abstract state, and derives the context c_f and abstract start state d_f for the callee. Afterwards, d_f is side-effected to the unknown for the start node st_f of f in context c_f. The abstract state d is combined with the abstract state obtained for the unknown for the return node ret_f of function f in context c_f, using $\mathsf{combine}^\sharp$, which, e.g., removes callee locals, restores caller locals, and possibly assigns a return value. The resulting abstract state is the contribution to the unknown corresponding to the sink v of the edge for the given context c.

The right-hand sides for other actions use the context c only to look up the abstract state at the appropriate predecessor. They may, however, also consult auxiliary unknowns or cause side-effects to those. By accumulating flow-insensitive invariants at such unknowns, our implementation can seamlessly support the analysis of multithreaded code [55].

As a warmup, we first cast some well-known approaches to context-sensitivity into our framework.

Example 2.1 (Full Contexts)**.** In this approach, the context is equal to the start state of the called function and $\mathbb{C} = \mathbb{D}$. Given a function $\mathsf{enter}^\sharp_{0,f,e}$ that computes the start state of the callee, one obtains:

$$\begin{aligned}
\textbf{let } \mathsf{enter}^\sharp_{f,e}\ c\ d \quad = \quad & \textbf{let } d' = \mathsf{enter}^\sharp_{0,f,e}\ c\ d \textbf{ in}\\
& (d', d')
\end{aligned}$$

Full contexts are often used in practice and are perhaps the most natural instance of the more general scheme of deriving contexts from abstract values.

Example 2.2 (Partial Contexts)**.** Consider contexts derived from start states via a projection function $\pi : \mathbb{D} \to \mathbb{C}$.

$$\begin{aligned}
\textbf{let } \mathsf{enter}^\sharp_{f,e}\ c\ d \quad = \quad & \textbf{let } d' = \mathsf{enter}^\sharp_{0,f,e}\ c\ d \textbf{ in}\\
& (\pi\, d', d')
\end{aligned}$$

π may, e.g., discard parts of abstract states such as values originating from infinite domains. We call these approaches *partial contexts* [2].

Example 2.3 (Call String)**.** This approach is obtained by

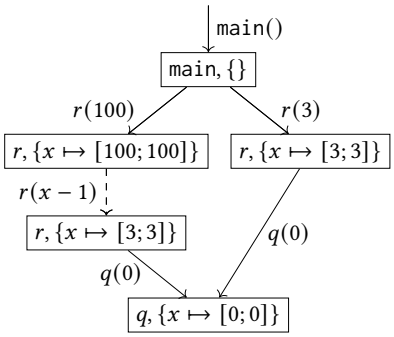
$$\begin{aligned}
\textbf{let } \mathsf{enter}^\sharp_{f,e}\ c\ d \quad = \quad & \textbf{let } d' = \mathsf{enter}^\sharp_{0,f,e}\ c\ d \textbf{ in}\\
& (c \cdot g, d')
\end{aligned}$$

where g is the procedure the edge e is contained in, and $\cdot$ denotes list concatenation. Again, $\mathsf{enter}^\sharp_{0,f,e}$ computes the start state of the callee. To obtain the k-call string approach, replace $c \cdot g$ by $\langle c \cdot g \rangle_k$ where $\langle\,\rangle_k$ denotes taking the k-suffix.

One may combine different styles of contexts using a product construction, which naturally fits into the framework. This, in fact, arises in practice when individual analyses are combined to exchange information and mutually improve precision, as done in many state-of-the-art analyzers [1, 7].

```
1  int q(int x) {
2   return x;
3  }
4  int r(int x) {
5   if (x ≤ 3){
6    return q(0);
7   }
8   return r(x-1);
9  }
10 int main() {
11  r(100);
12  r(3);
13  return 0;
14 }
```

(a) Code



(b) Call graph for full context analysis. Each node is given by a procedure and a context it is analyzed in. The dashed edge indicates call nodes omitted in the figure.

Figure 1. Recursive program and its fully context-sensitive call graph.

3 Context Lifting

When recursive procedures need to be analyzed in many different contexts, analyses using large or infinite sets of contexts may scale poorly.

Example 3.1. Consider the program from Figure 1. Function q returns its parameter x, whereas r recursively calls itself with a decremented parameter until it reaches the condition $x \leq 3$ and then calls $q(0)$. The `main` function calls $r(100)$, which leads to a chain of recursive calls with decreased parameter values until the condition holds. The second call of $r(3)$ in `main` does not cause recursive calls. An interval analysis with full contexts effectively *unrolls* the entire recursion, as shown in Figure 1b. The proposed approaches will avoid going too deep down this rabbit hole.

For other programs, even *infinitely many* contexts may be encountered during analysis – potentially even for terminating programs due to overapproximation. To tame the number of encountered contexts without changing existing analyses, we propose generic *analysis lifters*. Given a base analysis, a lifter extends or substitutes the original set of contexts $\mathbb{C}$ with some set $\hat{\mathbb{C}}$, and the abstract domain $\mathbb{D}$ with an adapted domain $\hat{\mathbb{D}}$. Accordingly, $\text{enter}^\sharp_{f,e} : \mathbb{C} \to \mathbb{D} \to (\mathbb{C} \times \mathbb{D})$ in the constraints is replaced with some $\hat{\text{enter}}^\sharp_{f,e} : \hat{\mathbb{C}} \to \hat{\mathbb{D}} \to (\hat{\mathbb{C}} \times \hat{\mathbb{D}})$ that adapts on the fly, i.e., during the lifted analysis, which contexts are generated. The new initial context is denoted by $c'_0 \in \hat{\mathbb{C}}$ and the new abstract start state by $d'_0 \in \hat{\mathbb{D}}$. The new constraint system thus has the form:

$$\eta\ [\mathsf{st}_{\texttt{main}}, \mathbf{c'_0}] \sqsupseteq \mathbf{d'_0}$$
$$\eta, \eta\ [v, c] \sqsupseteq [\![\, (\mathbf{u}, \mathbf{a}, \mathbf{v}), \mathbf{c} \,]\!]^\sharp_{\mathbf{B}}\ \eta, \quad \text{for } (u, a, v) \in \mathcal{E}, c \in \hat{\mathbb{C}}$$

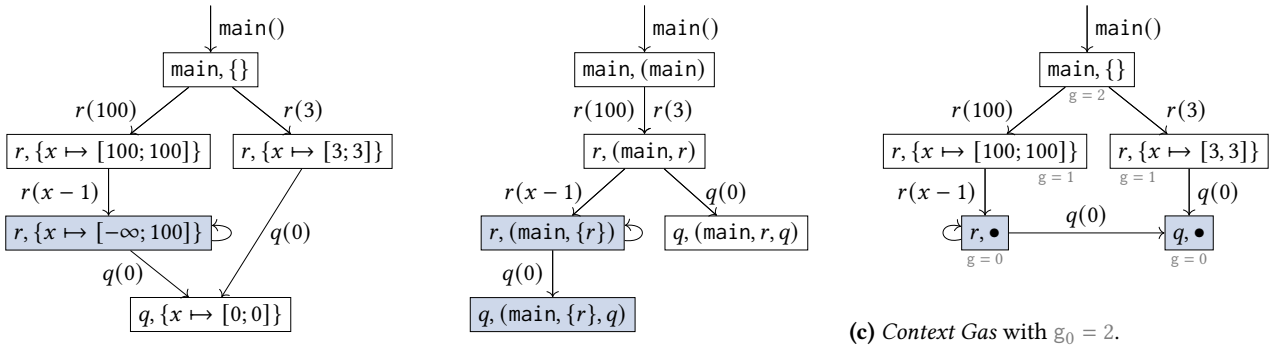
where changes are marked in bold and blue and $[\![\cdot]\!]^\sharp_B$ represents adapted right-hand sides. For function calls, the right-hand side is adapted by replacing $\text{enter}^\sharp_{f,e}$ and $\text{combine}^\sharp_{f,e}$ of the base analysis with $\hat{\text{enter}}^\sharp_{f,e}$ and $\hat{\text{combine}}^\sharp_{f,e}$, respectively. We will detail $\hat{\text{enter}}^\sharp_{f,e}$ for the individual lifters. In case $\hat{\mathbb{D}} = \mathbb{D} \times \mathbb{D}'$, the $\hat{\text{combine}}^\sharp_{f,e} : \hat{\mathbb{D}} \to \hat{\mathbb{D}}$ yields, for our instances, the $\mathbb{D}'$ component from its first argument, corresponding to the caller state. The right-hand sides for non-call edges remain unchanged, or, if the abstract domain was extended, additionally pass on parts added to the domain.

4 Context Widening

One instance where the number of encountered contexts may explode is when contexts are derived from corresponding start states, as described in Example 2.2, and $\mathbb{C}$ is large or infinite, which may, e.g., be the case when contexts incorporate elements from domains with infinite ascending chains. This echoes a problem already encountered in an intraprocedural analysis setting. For the abstract interpretation of programs with loops, abstract values may fail to stabilize. As a remedy, widening operators have been introduced, which are used to enforce termination of fixpoint algorithms even on domains with infinite ascending chains [8]. Here, we propose *Context Widening (CW)* – an approach to leverage such *widening* operators to limit the number of contexts to be considered.

To make our approach generic for the case where $\mathbb{C}$ is not a lattice, we do not perform a widening on the contexts directly, but instead reuse the widening operator already provided by the abstract domain $\mathbb{D}$. Recall that there is a function $\pi : \mathbb{C} \to \mathbb{D}$ to compute contexts from start states. When using Context Widening, for a call to a procedure f and some start state d, d is combined with the start states of calls to f further up the call stack using a widening operator ∇, *before* applying π. To realize Context Widening, the abstract state additionally stores a mapping $\mu \in \mathcal{M} = \mathbb{F} \to \mathbb{D}$, where $\mathbb{F}$ is the set of procedures. For every procedure, this mapping tracks an abstract state, abstracting all start states of the procedure further up the call stack. The new abstract domain $\hat{\mathbb{D}}$ is given by $\mathbb{D} \times \mathcal{M}$, where the ordering on $\mathcal{M}$ is induced by extending the ordering of $\mathbb{D}$ point-wise. The initial abstract state d'_0 is defined as (d_0, μ_0) where μ_0 is the mapping $\{\texttt{main} \mapsto d_0\}$ that yields d_0 for `main` and $\bot \in \mathbb{D}$ for every other procedure. The set of contexts $\hat{\mathbb{C}}$ is the same as for the base analysis and $c'_0 = c_0$. The function $\hat{\text{enter}}^\sharp_{f,e}$ receives the caller context $c \in \hat{\mathbb{C}}$ as well as state $(d, \mu) \in \hat{\mathbb{D}}$ to perform the widening and update the mapping:

$$\begin{aligned} \textbf{let } \hat{\text{enter}}^\sharp_{f,e}\ c\ (d, \mu) \quad = \quad & \textbf{let } \_, d' = \text{enter}^\sharp_{f,e}\ c\ d \textbf{ in} \\ & \textbf{let } d'' = \mu\ f\ \nabla\ d' \textbf{ in} \\ & (\pi\ d'', (d'', \mu \oplus \{f \mapsto d''\})) \end{aligned}$$



(a) *Context Widening* **(b)** *Loopfree Call String*

(c) *Context Gas* with $g_0 = 2$.
The gray text indicates the gas of the node.

Figure 2. The call graphs of the (a) *Context Widening*, (b) *Loopfree Call String*, and (c) *Context Gas* approaches for Example 3.1. The call label $r(x-1)$ is omitted on reflexive edges. Blue nodes highlight where lifters deviate from their base approaches.

The context and local state d' for the base analysis is computed with $\text{enter}^\sharp_{f,e}$. The tracked state for procedure f and d' are widened. The resulting state d'' is used to compute the context $\pi\, d''$ and is recorded in the callee's copy of μ.

Example 4.1. Assume $\mathbb{C}$ and $\mathbb{D}$ are instantiated with maps from parameters to intervals, and $\pi = \text{id}$. Figure 2a shows the corresponding call graph. The procedure `main` is analyzed with the initial values $c'_0 = \{\}$ and $d'_0 = (\{\}, (\texttt{main} \mapsto \{\}))$. When `main` calls r in line 11, those values are passed as parameters to $\widehat{\text{enter}}^\sharp_{r,e}$, which first computes $d' = \{x \mapsto [100; 100]\}$. As $\mu\, r = \bot$, we assume that widening returns its second parameter d'. The lifter inserts $\{r \mapsto d''\}$ into μ and returns $d'' = \{x \mapsto [100; 100]\}$. For the recursive function call $r(x-1)$ in line 8, those values are passed to $\widehat{\text{enter}}^\sharp_{r,e}$. Here, $\mu\, r$ evaluates to $\{x \mapsto [100; 100]\}$, and d' to $\{x \mapsto [99; 99]\}$ which results in $d'' = \{x \mapsto [-\infty; 100]\}$. The entry $\mu\, r$ is updated with this new value and returned together with d'' as context and abstract state. The recursive calls are then subsumed by the abstract start state $\{x \mapsto [-\infty; 100]\}$. For the call $q(0)$, $\mu\, q = \bot$ holds, so the widening does not affect the used abstract start state and context. For the sequence of calls starting with the invocation $r(3)$ in line 12 in `main`, no function is called twice, so it is not affected by the Context Widening.

5 Loopfree Call String

Context Widening does not immediately apply to call strings as considered by Sharir and Pnueli [46]. We thus propose the *Loopfree Call String (LFC)* approach that tracks an abstraction of call strings to limit the number of contexts. While the k-call string (k-CFA) approach uses k-suffixes (see Example 2.3), the *Loopfree Call String* represents cycles in the call string by the set of involved procedures. Thus, when a procedure f already appearing in the call string is to be appended again, all entries of the call string past f are merged into a set. Otherwise, the procedure is appended to the call string.

Therefore, the *Loopfree Call String* tracks a stack containing individual procedures and sets of procedures, defined as $\hat{\mathbb{C}} = \{(a_1, \cdots, a_n) \mid a_i \in \mathbb{F} \cup 2^{\mathbb{F}}\}$. Deviating from the standard call strings, we include the callee itself in the call string to detect cycles as soon as they happen instead of after one unrolling. The initial context c'_0 thus is $(\texttt{main})$, while the start state remains unchanged ($d'_0 = d_0$ with $\hat{\mathbb{D}} = \mathbb{D}$). The $\widehat{\text{enter}}^\sharp_{f,e}$ then is given by

$$\textbf{let } \widehat{\text{enter}}^\sharp_{f,e}\ c\ d \quad = \quad (c \star f, (\text{enter}^\sharp_{f,e}\ \bullet\ d)_2)$$

where the callee state is derived by passing d and $\bullet$, an arbitrary element of $\mathbb{C}$, to $\text{enter}^\sharp_{f,e}$ of the *base analysis*, $\cdot_2$ denotes the second component of the resulting pair, and $\star$ denotes appending to the loopfree call string. By construction, every procedure is contained at most once in a call stack, leading to an (albeit large) upper bound on the number of contexts.

Example 5.1. Figure 2b shows the call graph for the *Loopfree Call String* approach. As in Example 4.1, we define $\hat{\mathbb{D}} = \mathbb{D}$ as the map from parameters to intervals, leading to the initial value $d'_0 = \{\}$ for `main`. For the call $r(100)$ the $\widehat{\text{enter}}^\sharp_{r,e}$ transfer function is called with $c = (\texttt{main})$. Since c does not contain r, the context $(\texttt{main}, r)$ and the abstract start state $d' = \{x \mapsto [100; 100]\}$ are returned. The call $r(99)$ in line 8 is reached with $c = (\texttt{main}, r)$. Consequently, a loop is detected and converted to a set, resulting in $(\texttt{main}, \{r\})$. The following calls $r(98)$ to $r(3)$ receive the same context $(\texttt{main}, \{r\})$. Because the function q is not contained in the caller context $(\texttt{main}, \{r\})$, it is appended for the call $q(0)$. The approach coincides with call strings for the second, acyclic call sequence starting from line 12.

6 Context Gas

Intuitively, the *Context Gas (CG)* approach limits the depth of the abstract call stack up to which procedures are analyzed context-sensitively. The lifter tracks a counter value, called

gas, in its abstract domain $\hat{\mathbb{D}} = \mathbb{D} \times \mathbb{G}$, where $\mathbb{G} = \{0, \ldots, \mathrm{g}_0\}$ are the natural numbers up to some initial gas value g_0 with the usual ordering. Upon function call, the gas is decremented for the callee. The analysis uses the same context for all calls of a function with gas zero. For an analysis with initial gas g_0, the initial abstract state $d_0' = (d_0, \mathrm{g}_0)$, and context $c_0' = c_0$, are used. The set of contexts remains unchanged, i.e., $\hat{\mathbb{C}} = \mathbb{C}$. The context and caller state are passed to the $\widehat{\mathsf{enter}}^\sharp_{f,e}$ function, which now maintains the context gas:

$$\textbf{let } \widehat{\mathsf{enter}}^\sharp_{f,e}\ c\ (d, \mathrm{g}) = \textbf{let } c', d' = \mathsf{enter}^\sharp_{f,e}\ c\ d \textbf{ in}$$
$$\textbf{if } \mathrm{g} > 1 \textbf{ then } (c', (d', \mathrm{g} - 1)) \textbf{ else } (\bullet, (d', 0))$$

$\widehat{\mathsf{enter}}^\sharp$ computes the context $c' \in \mathbb{C}$ and state $d' \in \mathbb{D}$ via $\mathsf{enter}^\sharp_{f,e}$. If the gas exceeds one, it is decremented for the callee. Otherwise, the context $\bullet$ is used with gas zero.

Example 6.1. Figure 2c shows the 2-CG approach, i.e., using $\mathrm{g}_0 = 2$ as initial context gas. We use the same base analysis as in Example 4.1 and set $\mathbb{C} = \mathbb{D}$. For the analysis of `main` the context $c_0' = \{\}$ and abstract state $d_0' = (\{\}, 2)$, are used. For the call $r(100)$ in line 11, those values are passed to $\widehat{\mathsf{enter}}^\sharp_{r,e}$. Here, the lifter receives the values $c' = d' = \{x \mapsto [100; 100]\}$ computed by the $\mathsf{enter}^\sharp_{r,e}$ of the base analysis. As $\mathrm{g} > 1$, c' and d' are returned as computed by the base analysis. The function call to $r(99)$ in line 8 with $\mathrm{g} = 1$ leads to the callee gas 0, resulting in the context $\bullet$. Here, all calls before the return of this invocation are analyzed in context $\bullet$, including the one for the call $q(0)$. The second call sequence starts with $r(3)$ in line 12 and $\mathrm{g} = 2$. This call is analyzed context-sensitively, with $c' = \{x \mapsto [4; 4]\}$. Because the call to $q(0)$ yields $\mathrm{g} = 0$, it is analyzed again with context $\bullet$.

Remark. *Instead of switching to a $\bullet$ context, one may decide to hold on to* some *context-sensitivity, e.g., by switching to abstractions of call strings or to partial contexts, keeping only* crucial *information such as points-to-sets for function pointers.*

7 Evaluation

To evaluate our proposed constructions and compare them to well-known approaches from literature, we pose the following research questions:

- **(RQ1)** How do CIS, FC, k-CFA and ∞-CFA compare w.r.t. resource exhaustion and number of correct verdicts?
- **(RQ2)** Do CW, CG, and LCF decrease the number of tasks with resource exhaustion on recursive programs compared to FC and ∞-CFA?
- **(RQ3)** Do CW, CG, and LFC yield more correct verdicts on general tasks compared to FC and ∞-CFA?
- **(RQ4)** How do the different approaches to context-sensitivity scale on larger programs?

We implemented all proposed approaches in the Goblint abstract interpreter. Goblint was configured to run analyses that track interval and pointer information, as well as thread IDs and data-races. As analyses in Goblint are parametric in their contexts, they could be used as-is. As the fixpoint engine, a version of the top-down solver that supports side-effecting constraint systems [45] was used. This solver considers unknowns that correspond to start nodes of functions as widening points. We compare the context-insensitive (CIS), the fully context-sensitive (FC), 2-CFA, 10-CFA, ∞-CFA approaches, as well as the Context Gas (CG), Context Widening (CW), and the Loopfree Call String (LFC). The tests were executed on a machine with two Intel Xeon Platinum 8260 CPUs.

For **(RQ1)**-**(RQ3)**, we compared the approaches on tasks from the SV-COMP [5] benchmark suite. Using BenchExec [4], each task was assigned one CPU core, a memory limit of 15 GB, and a time limit of 15 minutes. Programs in the SV-COMP suite are labeled with properties of interest and expected verdicts. Table 1b shows the results for the properties *unreach-call*, *no-data-race* and *no-overflow*. For the *unreach-call* property, the analyzer is tasked to check for a given program whether a call to an error function is reachable. Tasks with the *no-data-race* and *no-overflow* properties require checking whether the program is data-race free and contains no integer overflow, respectively. Table 1a shows the results for the subcategories *simple recursive* and *recursive*, containing recursive programs that perform computations using machine integers, for the *no-overflow* property. Both tables display the number of correct verdicts, where the analyzer could establish that the property of interest holds, and the number of failing tests due to resource exhaustion, i.e., timeouts and stack overflows. W.r.t. **(RQ1)**, we find that for a considerable share of recursive tasks in Table 1a, ∞-CFA and FC run into resource exhaustion, while this is not the case for CIS and the k-CFA configurations. This indicates that ∞-CFA and FC have scalability problems on recursive tasks, as one would expect. The number of correct verdicts is the lowest for the CIS approach for both recursive subset of tasks as well as overall, indicating that CIS often loses too much precision. Looking at Table 1b, for these approaches, higher differentiation of contexts generally yields more correct verdicts, except for ∞-CFA with fewer correct verdicts than 10-CFA. For **(RQ2)**, Table 1a indicates that CW, CG, LFC, and k-CFA run into resource exhaustion less often than FC and ∞-CFA. For the *recursive-simple* tasks, there are no cases of resource exhaustion with CW, LFC, and any tested CG. There are some tasks in the *recursive* subcategory where 20-CG and 30-CG result in resource exhaustion. For **(RQ3)**, Tables 1a and 1b show that the CG instances with gas 20 and 30 yield a higher number of correct verdicts than any other approach. 30-CG yields 31% more correct verdicts on the two recursive subcategories than FC. CW has fewer correct verdicts than the FC approach but more than CIS. Interestingly, the 10-CFA approach fares well on the considered tasks. The number of correct verdicts found by LFC is in-between 2-CFA and 10-CFA, except for *no-overflow*, where it is lower.

Table 1. Results per approach in terms of programs for which the property of interest was established (in green), failing tests due to resource exhaustion (in red), the average CPU run time (t) in seconds, and the average used memory (m) in MB.

(a) Recursive tests with no-overflow property

	recursive-simple				recursive			
	(63)		t	m	(27)		t	m
CIS	22	0	0.2	24	7	0	0.2	24
2-CFA	26	0	0.2	24	7	0	0.2	24
10-CFA	46	0	0.2	25	8	0	0.8	29
∞-CFA	42	21	19.4	338	3	21	104.4	985
LFC	24	0	0.2	24	7	0	0.2	24
CW	22	0	0.2	25	7	0	0.2	24
10-CG	42	0	0.2	24	8	0	0.8	29
20-CG	50	0	0.2	24	6	2	67.4	433
30-CG	58	0	0.2	25	6	3	69.3	430
FC	45	18	4.7	329	4	22	183.0	1133

(b) All tasks by property

	unreach-call				no-data-race				no-overflow			
	(15014)		t	m	(1014)		t	m	(9236)		t	m
CIS	1862	1691	124	319	251	8	12	67	3894	21	3.2	38
2-CFA	1960	1647	122	353	585	3	6	66	3904	11	2.3	37
10-CFA	1997	1656	120	360	591	2	4	54	3927	17	2.5	37
∞-CFA	1989	1775	125	357	591	2	4	54	3892	189	11.8	49
LFC	1990	1649	121	359	591	2	4	54	3899	15	2.3	37
CW	1978	1916	130	792	695	7	8	66	3911	20	3.1	53
10-CG	2033	1637	119	420	694	11	13	79	3934	23	3.4	44
20-CG	2088	1701	114	503	695	7	8	67	3939	21	3.1	46
30-CG	2092	1694	115	525	695	7	8	64	3947	23	3.2	45
FC	2045	1858	126	507	695	7	8	64	3917	150	4.7	55

For **(RQ4)**, we measured run times on twelve *Coreutils*, again with a time limit of 15 minutes (Table 2). For these programs, a value analysis was performed that flags issues such as `NULL`-pointer dereferences, invalid calls to `free`, and dead code. Here, we focus on the context-sensitive approaches. We find that 2-CFA is often the fastest approach. In contrast, ∞-CFA exhausts its resources on half the programs, as expected. FC is often more expensive than 2-CFA but only failed for two tasks. Our proposed lifters terminate on all but one task, except for CW, which fails on two tasks. Runtimes of LFC are frequently between those of 2-CFA and 10-CFA. Interestingly, the exhibited relationships seem to depend highly on the program, which may warrant further study.

Overall, CG instances perform particularly well on SV-COMP, w.r.t. correct verdicts on both recursive and overall tasks, with 30-CG achieving the highest number of correct verdicts among the tested configurations. The approach limits its context-sensitivity, but not as eagerly as CW, and does not unnecessarily split contexts by call string when abstract start states are the same, combining advantages of both k-call string and full context approaches.

Threats to Validity. For **(RQ1)** to **(RQ3)**, we considered tasks from the benchmark suite of SV-COMP, which is a collection of verification tasks the software verification community contributed. While the set of tasks is diverse and contains programs that are challenging for software-verifiers, not all of them represent real-world code. To mitigate this, for **(RQ4)**, a set of real-world programs from the *Coreutils* was considered. As there are no expected verdicts for these real-world programs, we did not compare the precision of the approaches here, but focussed on run times. Further, the impact of the presented lifters on precision and runtime may depend on the employed analyses and domains. Nevertheless, abstract domains used in the evaluation, such as intervals and points-to-sets, are also commonly used in other static analyzers. The choice of fixpoint algorithm may also influence the results. While a top-down solver was used for the experiments here, further evaluation may consider other solvers for side-effecting constraint systems, such as SLR solvers [3].

8 Related Work

Static analyses can be sensitive w.r.t. a variety of aspects (see Park et al. [38] for a recent survey). Hardekopf et al. [9] study selectively giving up *control-flow* sensitivity as a form of widening. Over the years, different variants of *context*-sensitivity were proposed. Well-known approaches include the call string approach [46, 47], the functional approach [39, 46], object-sensitivity [31, 32, 48, 54], and type-sensitivity [48]. The empirical trade-off between k-call strings and the functional approach for procedural languages has been investigated (see, e.g., [29, 30]). For object-oriented languages, k-object-sensitivity differentiates calls by the k most recent creation sites of objects. Type-sensitivity is similar to 1-object-sensitivity, tracking types instead of allocation sites [33]. Many common approaches suffer from a huge number of contexts, making analysis expensive [20, 23, 35, 36, 52, 53].

Selective context-sensitive analysis [10, 14, 16, 17, 24–27, 35–37, 49–51, 54, 56] tries to mitigate this by deciding per procedure which, if any, context to use. These approaches rely on pre-analyses or machine-learning, while ours do not. Khedker and Karkare [21] observe that when multiple call strings yield the same data-flow value at the start node of a procedure, it suffices to consider one for further analysis, and reuse the result for the others. Such an approach may also be combined with LFC proposed here. The authors also propose an approximating version of the call string approach for recursive programs where the results for call strings that contain a given call site k times or more are merged. While this is similar in spirit to LFC in that call

Table 2. CPU run time in seconds for the different approaches on *Coreutils*. Red cells indicate failed tests.

	cksum	cp	cut	dd	df	du	ls	mv	nohup	ptx	sort	tail
lines	10293	11628	10534	10841	10650	10621	11669	11552	10277	10949	12060	10969
2-CFA	2.0	49.2	4.6	18.9	70.4	71.3	900.0	60.0	3.2	23.7	84.3	9.5
10-CFA	1.9	275.3	3.6	87.9	214.0	184.0	900.0	283.4	2.9	14.2	900.0	14.7
∞-CFA	1.7	900.0	3.7	-	214.3	-	900.0	900.0	2.9	14.1	-	14.8
LFC	1.3	131.5	4.5	86.9	212.3	80.5	900.0	154.5	3.1	14.3	238.7	14.7
CW	1.5	852.7	7.1	271.2	490.3	303.4	900.0	815.3	4.6	39.7	900.0	51.5
10-CG	1.3	160.2	5.3	42.3	300.5	168.9	900.0	128.1	3.7	23.5	521.1	24.5
20-CG	1.3	389.8	5.3	162.1	286.7	171.6	900.0	341.1	3.6	23.6	411.7	21.7
30-CG	1.3	314.1	5.3	162.3	287.0	171.1	900.0	364.8	3.6	23.5	413.3	21.8
FC	1.9	194.0	6.1	124.7	278.2	163.4	900.0	170.2	3.7	23.9	-	21.8

strings are abstracted to deal with recursion, it makes different trade-offs, as it does not ensure that recursion-free prefixes are maintained. Further, the representative call string in Khedker and Karkare [21] depends on the iteration order of the fixpoint algorithm, which may render results brittle between analyzer reruns. Montagu and Jensen [34] consider call strings of bounded length for the analysis of functional programs. The analysis presented there may signal to the fixpoint algorithm when widening should be performed on start states of functions. In our setting with side-effecting constraint systems, the fixpoint algorithm already performs widening on abstract start states when the corresponding unknown receives multiple contributions via side-effect [45]. Lerner et al. [22] hint at giving up all context-sensitivity for a procedure after a certain number of contexts have been encountered for it and call this context widening. The idea of a *gas* parameter has found applications in static analysis before. For example, in the Verasco project, aiming for a formally verified abstract interpreter, Jourdan et al. [18, 19] use a monotonically decreasing gas value to be able to define and reason about possibly non-terminating functions in Coq. In the extracted analyzer, however, the gas value is set to infinity, meaning that Verasco may not terminate, but its result is provably sound whenever it does. Our notion of gas instead is used to bound the number of encountered contexts. Here, the gas is part of the abstract states attained at program points. Thus, for a procedure and a given context, the gas value may increase during fixpoint iteration, when a call to the procedure is encountered that results in the same context but with higher gas. Moreover, when our gas is exhausted, the analysis does not terminate with an exception, but instead locally abandons context-sensitivity. Together with widening, it ensures termination even for recursive input programs. Jaiswal et al. [15] comprehensively survey approaches to context-sensitivity for data-flow analysis, and present a declarative unifying framework. For the tabulating variant of the functional approach, they require underlying domains to be finite. The unifying framework of Jaiswal et al. [15] does not endeavor to lend itself directly to an implementation. In contrast, our framework for abstract interpretation translates into *side-effecting constraint systems* for which efficient solvers and analysis frameworks exist. One of our primary concerns are techniques for bounding the number of *encountered* contexts even when the set of possible contexts is infinite, which Jaiswal et al. [15] do not consider. Beyond more conventional approaches, Hedenborg et al. [11, 12, 13] endeavor to represent context-sensitive information memory-efficiently by so-called χ-terms. This is orthogonal to limiting the number of encountered contexts. Bourdoncle [6] presents the dynamic partitioning framework to construct expressive domains for abstract interpretation and details how context-sensitivity may be seen as an instance. To make *shape-analysis* applicable to data of other invocations on the stack, the data from these stack frames may be included in the abstract state [40, 41]. Ma et al. [28] try to regain some precision lost due to context-insensitivity. They identify three patterns in Java programs where incorporating imprecise information from the callee leads to an unnecessary loss of precision and remove these spurious flows. Exploiting such insights may also be beneficial for context-sensitive analyses.

9 Conclusion

Using a common framework leveraging side-effecting constraint systems, we have recapped call strings and full contexts and presented three lifters that tame context-sensitivity on the fly. They were implemented in Goblint and evaluated on SV-COMP. Compared to full contexts and unbounded call strings, the lifters timed out less often and caused fewer stack overflows on recursive benchmarks. Context Gas, with initial gas 30, performed best w.r.t. correct verdicts. Future work may explore applying such lifters only to parts of the abstract state that may lead down rabbit holes.

Acknowledgments. We thank Ralf Vogler who conceived the Context Widening approach together with Helmut Seidl. This work was supported in part by Deutsche Forschungsgemeinschaft (DFG) — 378803395/2428 ConVeY.

References

[1] Kalmer Apinis. 2014. *Frameworks for analyzing multi-threaded C*. Ph. D. Dissertation. Technical University Munich. https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20140606-1189191-0-9

[2] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2012. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7705)*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer, 157–172. https://doi.org/10.1007/978-3-642-35182-2_12

[3] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2013. How to combine widening and narrowing for non-monotonic systems of equations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 377–386. https://doi.org/10.1145/2491956.2462190

[4] Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 887–904. https://doi.org/10.1007/978-3-662-49674-9_55

[5] Dirk Beyer. 2023. Competition on Software Verification and Witness Validation: SV-COMP 2023. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 495–522. https://doi.org/10.1007/978-3-031-30820-8_29

[6] François Bourdoncle. 1992. Abstract Interpretation by Dynamic Partitioning. *J. Funct. Program.* 2, 4 (1992), 407–423. https://doi.org/10.1017/S0956796800000496

[7] David Bühler. 2017. *Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C*. Ph. D. Dissertation. University of Rennes 1, France. https://tel.archives-ouvertes.fr/tel-01664726

[8] Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings (Lecture Notes in Computer Science, Vol. 631)*, Maurice Bruynooghe and Martin Wirsing (Eds.). Springer, 269–295. https://doi.org/10.1007/3-540-55844-6_142

[9] Ben Hardekopf, Ben Wiedermann, Berkeley R. Churchill, and Vineeth Kashyap. 2014. Widening for Control-Flow. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8318)*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, 472–491. https://doi.org/10.1007/978-3-642-54013-4_26

[10] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. 13–18. https://doi.org/10.1145/3088515.3088519

[11] Mathias Hedenborg, Jonas Lundberg, and Welf Löwe. 2021. Memory efficient context-sensitive program analysis. *Journal of Systems and Software* 177 (2021), 110952. https://doi.org/10.1016/J.JSS.2021.110952

[12] Mathias Hedenborg, Jonas Lundberg, Welf Löwe, and Martin Trapp. 2015. Approximating Context-Sensitive Program Information. In *Programmiersprachen und Grundlagen der Programmierung KPS 2015*.

[13] Mathias Hedenborg, Jonas Lundberg, Welf Löwe, and Martin Trapp. 2022. A framework for memory efficient context-sensitive program analysis. *Theory of Computing Systems* 66, 5 (2022), 911–956. https://doi.org/10.1007/S00224-022-10093-W

[14] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Selective conjunction of context-sensitivity and octagon domain toward scalable and precise global static analysis. *Software: Practice and Experience* 47, 11 (2017), 1677–1705. https://doi.org/10.1002/SPE.2493

[15] Swati Jaiswal, Uday P. Khedker, and Alan Mycroft. 2022. A Unified Model for Context-Sensitive Program Analyses: : The Blind Men and the Elephant. *ACM Comput. Surv.* 54, 6 (2022), 114:1–114:37. https://doi.org/10.1145/3456563

[16] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428247

[17] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28. https://doi.org/10.1145/3133924

[18] Jacques-Henri Jourdan. 2016. *Verasco: a formally verified C static analyzer*. Ph. D. Dissertation. Universite Paris Diderot-Paris VII. https://tel.archives-ouvertes.fr/tel-01327023

[19] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A formally-verified C static analyzer. *Acm Sigplan Notices* 50, 1 (2015), 247–259. https://doi.org/10.1145/2676726.2676966

[20] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices* 48, 6 (2013), 423–434. https://doi.org/10.1145/2491956.2462191

[21] Uday P Khedker and Bageshri Karkare. 2008. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Compiler Construction: 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 17*. Springer, 213–228. https://doi.org/10.1007/978-3-540-78791-4_15

[22] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. 2005. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 364–377. https://doi.org/10.1145/1040305.1040335

[23] Ondřej Lhoták and Laurie Hendren. 2006. Context-sensitive points-to analysis: is it worth it?. In *International Conference on Compiler Construction*. Springer, 47–64. https://doi.org/10.1007/11688839_5

[24] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. https://doi.org/10.1145/3276511

[25] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 129–140. https://doi.org/10.1145/3236024.3236041

[26] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–40. https://doi.org/10.1145/3381915

[27] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–46. https://doi.org/10.1145/3450492

[28] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 539–564. https://doi.org/10.1145/3591242

[29] Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. A correspondence between two approaches to interprocedural analysis in the presence of join. In *European Symposium on Programming Languages and Systems*. Springer, 513–533. https://doi.org/10.1007/978-3-642-54833-8_27

[30] Florian Martin. 1999. Experimental comparison of call string and functional approaches to interprocedural analysis. In *International Conference on Compiler Construction*. Springer, 63–75. https://doi.org/10.1007/978-3-540-49051-7_5

[31] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 1–11. https://doi.org/10.1145/566172.566174

[32] Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

[33] Ana L. Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

[34] Benoît Montagu and Thomas P. Jensen. 2021. Trace-based control-flow analysis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 482–496. https://doi.org/10.1145/3453483.3454057

[35] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 475–484. https://doi.org/10.1145/2594291.2594318

[36] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2015. Selective x-sensitive analysis guided by impact pre-analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 2 (2015), 1–45. https://doi.org/10.1145/2821504

[37] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 572–588. https://doi.org/10.1145/2814270.2814309

[38] Jihyeok Park, Hongki Lee, and Sukyoung Ryu. 2021. A survey of parametric static analysis. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–37. https://doi.org/10.1145/3464457

[39] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61. https://doi.org/10.1145/199448.199462

[40] Noam Rinetzky and Shmuel Sagiv. 2001. Interprocedural Shape Analysis for Recursive Programs. In *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2027)*, Reinhard Wilhelm (Ed.). Springer, 133–149. https://doi.org/10.1007/3-540-45306-7_10

[41] Xavier Rival and Bor-Yuh Evan Chang. 2011. Calling context abstraction with shapes. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 173–186. https://doi.org/10.1145/1926385.1926406

[42] Michael Schwarz, Julian Erhard, Vesal Vojdani, Simmo Saan, and Helmut Seidl. 2023. When Long Jumps Fall Short: Control-Flow Tracking and Misuse Detection for Non-local Jumps in C. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023*, Pietro Ferrara and Liana Hadarean (Eds.). ACM, 20–26. https://doi.org/10.1145/3589250.3596140

[43] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. 2021. Improving Thread-Modular Abstract Interpretation. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 359–383. https://doi.org/10.1007/978-3-030-88806-0_18

[44] Michael Schwarz, Simmo Saan, Helmut Seidl, Julian Erhard, and Vesal Vojdani. 2023. Clustered relational thread-modular abstract interpretation with local traces. In *European Symposium on Programming*. Springer, 28–58. https://doi.org/10.1007/978-3-031-30044-8_2

[45] Helmut Seidl and Ralf Vogler. 2021. Three improvements to the top-down solver. *Math. Struct. Comput. Sci.* 31, 9 (2021), 1090–1134. https://doi.org/10.1017/S0960129521000499

[46] Micha Sharir and Amir Pnueli. 1978. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences.

[47] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda.* Ph. D. Dissertation. Carnegie Mellon University.

[48] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 17–30. https://doi.org/10.1145/1926385.1926390

[49] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 485–495. https://doi.org/10.1145/2594291.2594320

[50] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485524

[51] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (oct 2021), 27 pages. https://doi.org/10.1145/3485524

[52] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer, 489–510. https://doi.org/10.1007/978-3-662-53413-7_24

[53] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 278–291. https://doi.org/10.1145/3062341.3062360

[54] Manas Thakur and V. Krishna Nandivada. 2020. Mix your contexts well: opportunities unleashed by recent advances in scaling context-sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3377555.3377902

[55] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International*

Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 391–402. https://doi.org/10.1145/2970276.2970337

[56] Shiyi Wei and Barbara G Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPICS.ECOOP.2015.712

[57] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William W. Pugh and Craig Chambers (Eds.). ACM, 131–144. https://doi.org/10.1145/996841.996859

Received 2024-03-07; accepted 2024-04-19

ValBench: Benchmarking Exact Value Analysis

Marc Miltenberger
marc.miltenberger@sit.fraunhofer.de
Fraunhofer SIT | ATHENE
Darmstadt, Hessen, Germany

Steven Arzt
steven.arzt@sit.fraunhofer.de
Fraunhofer SIT | ATHENE
Darmstadt, Hessen, Germany

Abstract

Value analysis is an important building block in static program analysis. While several approaches have been proposed, evaluating and comparing them is not trivial. Up to this day, a reliable and large benchmark specifically for value analysis is missing. Such a suite must not only provide test cases, but also a ground truth with the correct values to be found.

In this paper, we propose ValBench, an extensible value benchmark suite consisting of 372 test cases for Java analysis and 59 test cases for Android analysis tools. Furthermore, we present an evaluation framework that automatically generates a ground truth for these test cases, identifies their respective challenges for program analysis and orchestrates the execution and result collection on the various value analysis tools. We further present an evaluation of 7 existing value analysis tools on ValBench and highlight the challenges faced by these tools as an empirical overview over the state of the art in value analysis.

CCS Concepts: • **Security and privacy → Software security engineering**.

Keywords: string analysis, value analysis, benchmarks

ACM Reference Format:
Marc Miltenberger and Steven Arzt. 2024. ValBench: Benchmarking Exact Value Analysis. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3652588.3663322

1 Introduction

Value analysis is a vital building block in static program analysis. When building callgraphs, for example, the target class and method names are required to uniquely identify the callee [1, 9]. Without this information, the callgraph either misses edges or must resort to coarse over-approximation. Android inter-component communication poses similar challenges. Components inside an app may invoke other components in the same or a different app by sending an `Intent`, which contains data for resolving the target component as well as the data to pass to the recipient. To precisely model such calls, the callgraph analysis must infer various fields of the intent [12, 13]. Intents are particularly challenging, because they form a composite value analysis problem. The analysis must ensure not to mix of the values of different fields across objects if multiple intents are in scope.

Value analysis is also essential for vulnerability detection. Many security properties rely on checking API parameter values. A vulnerable program may, for example, pass hard-coded cryptographic keys to a crypto API, may send sensitive information to an unprotected *http* endpoint rather than using TLS, or may pass an outdated algorithm name such as `DES` to an encryption function. In total, the completeness and correctness of the value analysis is of utmost importance.

Most value analysis research has focused on string analysis. Some approaches return concrete strings, while others such as *JSA* [4], *COAL* [12] or *Violist* [8] return regular expressions. The latter approach may support values that depend on external inputs that are unavailable to static analysis e.g., user input or results retrieved from a remote server via a network connection. As we show, in many cases returning regular expressions instead of concrete values can, however, be a consequence of imprecisions inherent to the algorithm. For example, when encountering string decryption methods in obfuscated apps, these approaches may return a generic `.*`, which offers no information.

Despite the existing value analysis research, measuring the accuracy of a value analysis remains non-trivial. It requires a comprehensive dataset with a known ground truth. While real-world applications can provide complex examples, they usually lack the ground truth. Reverse-engineering these examples comes at a prohibitive effort on a large scale. Real-world examples are therefore useful for analyzing the time and memory consumption of the value analysis, but do not help for a precision assessment.

For ensuring a ground truth, papers such as *JSA* propose test suites. These suites, however, are usually limited in size and scope and focus on the features in the focus of the respective tool. When new approaches are published, they extend the existing suites for their own evaluation, leading to a multitude of different tool-specific test suites that do not allow for a broad evaluation of the state of the art in value analysis.

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663322

We consider this a serious drawback hindering progress in value analysis research.

In this paper we present the, to the best of our knowledge, first comprehense test suite for value analysis. Our test suite comprises 431 test cases written in Java based on our test suite framework. To ensure that the ground truth is correct, our framework runs the test case on the JVM and records the computed value. This step generates the ground truth, thereby avoiding the manual effort and the risk for human error. The framework then executes all registered analysis clients and compares the result returned by each analysis client with the ground truth values. As a result, the framework collects detailed evaluation data on a per-analysis and per-test case level as well as summarized recall and precision.

In total, we present the following original contributions in this paper.

- A set of 431 test cases for exact value analysis, which we call ValBench. They can be used to evaluate and compare different exact value approaches on a ground truth. ValBench can be used on Android and JVM analyses tools.
- An extensible evaluation framework, which allows an easy comparison between different value analysis approaches. It supports running various approaches, parsing their results and comparing them. Initially, it supports 7 existing approaches. New approaches can easily be implemented.
- An evaluation of 7 existing value analysis frameworks on the ValBench test suite.

The remainder of this paper is structured as follows: First, we present related work in Section 2. In Section 3 we lay out the construction and evaluation approach of the test suite. In Section 4, we show our evaluation results. Lastly, Section 5 concludes the paper.

2 Related Work

Testing of static analyzers has been explored by prior work. Casso et al [3] proposed a an approach for the Ciao programming language, which generates assertions based on the static analysis results. Using dynamic analysis, these assertions are then used to check whether the generated assertions hold and the static analysis is correct. The approach proposed by Klinger et al [7] generates benchmarks using a given set of seed programs in order to test multiple analyzers. It automatically places assertions regarding numerical properties of variables in the code. In order to check whether the assertion can be violated (true positive) or not (true negative), they use the 'bugs-as-deviant-behavior' strategy proposed by Engler et al [5], where the result of the majority of analyzers is assumed to be correct, while the result determined by the minority is assumed to be incorrect.

ValBench can be used as a tool to test Java and Android value analysis tools. In contrast to both approaches, ValBench offers a ground-truth. Various approaches for extracting values from Java programs or Android apps have been proposed. We used the keywords "Java value analysis" and "Java string analysis" on IEEE Xplore and ACM Digital Library. We tested approaches with an directly available tool or where a tool could be provided upon request by the authors.

JSA [4] focuses on extracting strings. It constructs automata which model the language of all possible strings as elements and string operations as automata transitions. *JSA* returns these automata as regular expressions. *COAL* [12] also yields regular expressions, but is based on static constant propagation and a domain-specific language to model complex data structures such as Intents. *Violist* [8] applies a region-based analysis to identify code segments and loops which compute the value of interest and uses an interpreter to compute values. *Violist* can either perform loop unrolling or output regular expressions similar to *JSA*. The loop unrolling technique does not respect loop conditions. Instead, the loop is unrolled exactly a certain number of times. *BlueSeal* [15] is based on backward slicing and abstract interpretation, but the paper does not give many details on the algorithm.

Other approaches return concrete strings rather than regular expressions. *Harvester* [14] is a hybrid (static & dynamic) approach. It performs a static backward slice and then runs the slices dynamically to obtain the values. *Harvester* uses the term "logging point" to refer to the tuple consisting of the variable of interest with the location of interest. We adopted this notion in this paper. *ValDroid* [10] is similar to *Harvester*, but instead of a dynamic phase, it performs a static simulation. *StringHound* [6] tries to deobfuscate strings. It first uses classification to find string decryption methods. These methods get extracted by static backward slicing and run on the JVM. As the JVM does not have adequate models for Android system APIs, it cannot perform value extraction when Android APIs are involved. Although it was designed primarily for string decryption, it can be easily modified to run a generic value analysis by changing the classification logic.

For evaluation, *JSA*'s test cases are closest to ValBench. *JSA* features a set of 328 test cases and is used by related work[4, 8] to test regular expression based value approaches. Each test cases features three regular expressions as a ground truth. Each regular expression defines a different degree of precision. Listing 1 shows the `AnnotatedField` *JSA* test case. According to the testcase, the most precise regular expression, which models the `field` value is *C*|null*, albeit the exact value *CCCCCCCCCC* is deterministically calculated. While *C*|null* is technically correct, it is imprecise. Also note that the resulting value cannot possibly be **`null`**. We therefore argue that this test-case is insuitable for evaluating precise value approaches.

Furthermore, the *JSA* test suite references *JSA*-specific resolvers in 35 test cases, i.e., the test suite is not independent from the approach under test. Lastly, we observed several mistakes in the ground truth of the *JSA* test suite. The *ArrayOfObjectsToString* test case, for example, expects the `toString()` of an array to equal `"[Ljava.lang.Object; @3e25a5"`. Note that the part after the @ is the hash code, which is implementation-dependent and in most implementations dependent on the memory address of the array.

We conclude that the *JSA* test suite is unsuitable for precisely evaluating arbitrary value analysis tools. Aside from the *JSA* test suite, the DaCapo [2] benchmark suite provides programs for evaluating Java-based static analyses. DaCapo, however, is not specific to value analysis and does not provide a corresponding ground truth.

```
1 private @Type("C*|null") String field;
2 public void foo() {
3    field = "";
4    for (int i=0; i<10; i++) field += "C";
5    StringTest.analyze(field, "C*|null", "C*|
         null", "C*|null");
6 }
```

Listing 1. *JSA* test case `AnnotatedField`

3 Approach

The test framework consists of several steps. It first takes the test case source with optional annotations. If no ground truth is available, it compiles the test case and runs it on the JVM to obtain the ground truth. Afterwards, the framework creates JAR and APK files for each test file that are then presented to the value analysis approaches. Finally, the values returned by the approach are compared to the ground truth for the evaluation report and statistics (overall recall and precision). We next explain these steps in more detail.

3.1 Defining Test Cases

Each entry point method needs to be annotated with the `@ValueComputationTestCase` annotation. Note that the test case is not restricted to the entrypoint, i.e. the entry point may potentially call other methods. There are two different types of test case, return-based and explicit test cases. For return-based test cases, the value analysis must reconstruct the value returned by the test case. Explicit test cases are **`void`** methods and thus have no return value. Instead, they call the method `explicitLoggingPoint(Object)` with a specific variable in the test case, similar to the `analyze` method in *JSA* test case shown in Listing 1. For these test cases, the value analysis goal is to reconstruct the first parameter at the callsite of `explicitLoggingPoint`. Technically, both options are equivalent, because the test framework automatically inserts calls to `explicitLoggingPoint` before every **`return`** statement in each test case. These logging points are used as an input for the value analysis tools.

By default, the ground truth for each test case is obtained automatically by running the test cases on the JVM. The test framework records the values (method return value and explicitly requested values) and places them in a JSON file. Adding new test cases therefore requires only minimal effort. Alternatively, test case authors may use the `expectedValues` attribute of the `ValueComputationTestCase` annotation to manually provide the expected results. In this case, the test case will not be run on the JVM.

Android-specific test cases that are only valid in the context of an app and that cannot be run on a JVM must always be annotated by hand. We decided against integrating an Android runner due to the complexity of integrating an Android build chain and emulator-based execution environment. Such a feature would have hampered the platform independence of our test framework.

Further, each test case is annotated with a machine readable list of challenges. Some test cases may, for example, require the string analysis approach to model certain API calls such as `Integer.valueOf`. Other test cases need field, array handling or loop support. This information is kept for statistics. Each test case is still presented to each tool. If a tool does not support loops, for example, the analyst can cluster the failed test cases around these properties.

If no challenge annotation is given for a test case, ValBench determines the list of challenges per test case automatically using static analysis. It records API calls and notes whether the test case uses fields, arrays, loops etc.

Although ValBench test expect the tools to return strings for easy comparison, many test cases require the tools to handle complex objects. Correctly handling the Android `Intent` class, for example, is a composite value problem including key/value maps that is represented in the string representation of the `Intent`. Note that even though the return value are strings, we consider ValBench a *value* test suite, since the proper handling of complex values is necessary in order to solve many test cases correctly.

3.2 Building JARs and APKs

As most value analysis tools work on binaries rather than source code, our test framework automatically builds JAR files for Java-based test cases and APK files for both Android- and Java-based test cases. Note that Java-based test cases can run on Android as well. Whether a tool is capable of processing JARs, APKs or both is defined by a tool-specific execution adapter class.

In our evaluation we found that some tools run into timeouts when presented with the whole test suite as a single JAR or APK file. To avoid this issue, we have implemented a shrinker utility based on Soot, which creates JARs and APKs with a single test case. Dependencies, e.g., libraries compiled into the APK, are reduced to the minimum set of

methods and classes necessary. This shrinker is best-effort. For JAR files, the framework automatically re-runs the created shrinked JARs and compares the results to the ground truth to avoid introducing bugs into the testcases. Testing the APK versions would also be possible, but would require a phone or emulator during build. We therefore omitted this feature to keep the setup simple. Test cases can be annotated to disable shrinking.

3.3 Running the Approaches

Each approach connects to the test framework via a tool-specific adapter. The adapter is responsible for running the tool with the correct configuration and for collecting the tool's outputs. Adapters may invoke tools via the command-line or may also use APIs offered by the tools.

3.4 Evaluating Results

The test framework compares the values returned by the value analysis approaches with the ground truth using string equality. ValBench focuses on evaluating approaches that return *concrete* values. If a tool instead returns a regular expression, our evaluation framework only checks the concrete parts. An expression `A|B`, for example, is split into the concrete values `A` and `B` which are compared against the ground truth. Recall that the ground truth is always a concrete value. Hence, any wider regular expression is necessarily an over-approximation.

Instead of a single concrete value, test cases may return different values depending on conditions. In such cases, value analyses must return *all* values to achieve 100 % recall in our evaluation framework. Note that running the test case on the JVM may only return a subset of these values if some conditions cannot be fulfilled during execution. In that case, analysts can augment the test case with additional values by hand or ensure that conditions can be forced from the outside, e.g., via environment variables.

4 Evaluation

The ValBench test suite features 372 JVM and 59 Android test cases. In this section, we run the framework against 7 existing value analysis approaches on this test suite.

RQ1: How does ValBench compare to *JSA* test cases?

The *JSA* test suite features only 41 tests (12.5 %) involving arrays and 15 tests (4.57 %) with loops. In contrast, ValBench features 123 array tests (24.83 %) and 100 (23.20 %) with loops. Furthermore, ValBench contains 35 (8.12 %) testcases involving reflection and 29 (6.73 %) involving streams, while *JSA* testcases feature neither.

As noted in Section 3, ValBench automatically infers a list of challenges for each test case. Table 1 and Table 2 show the Top 10 challenges for ValBench and *JSA* test cases. ValBench features a larger number of test case involving

Table 1. Top 10 challenges for ValBench test cases

Challenge	Number of test cases
`StringBuilder.toString`	214 (49.65 %)
`StringBuilder.append`	204 (47.33 %)
`StringBuilder.<init>`	201 (46.64 %)
Primitive type - int	196 (45.48 %)
Interprocedural	134 (31.09 %)
Fields	125 (29.00 %)
Arrays	123 (28.54 %)
Arithmetic calculations	107 (24.83 %)
Loops	100 (23.20 %)
`StringBuilder.append`	91 (21.11 %)

Table 2. Top 10 challenges for *JSA* test cases

Challenge	Number of test cases
Interprocedural	112 (34.15 %)
`StringBuffer.toString`	101 (30.79 %)
If	100 (30.49 %)
`StringBuilder.toString`	97 (29.57 %)
`StringBuilder.<init>`	87 (26.52 %)
`StringBuffer.append`	83 (25.30 %)
Fields	78 (23.78 %)
`String.valueOf`	72 (21.95 %)
`StringBuffer.<init>`	71 (21.65 %)
`StringBuilder.append`	70 (21.34 %)

Fields (29 % in ValBench vs 23% in *JSA*), Arrays (28.54 % vs 12.50 %), Arithmetic Calculations (24.83 % vs 5.79 %) and Loops (23.20 % vs 4.57 %).

Furthermore, ValBench contains 59 Android-specific test cases. These test cases require value analysis tools to support composite data structures such as Intents and Pairs.

RQ2: How effective are value analysis tools on ValBench?

We adapted *JSA*, *Violist*, *BlueSeal* and *COAL* slightly to use the newest Soot release, as the outdated versions shipped with the tools were not capable of analyzing modern Java class files. We modified StringHound to classify testcase methods as string decryption methods in order to use its slicing algorithm. For the evaluation of *Harvester*, *COAL*, *BlueSeal* and *ValDroid* we used the Android version of our test cases. We supply all changed tools (except *BlueSeal* and *ValDroid*, which we cannot redistribute due to missing the license) in our data package.

Note that ValBench supports multiple values per test case, either by using different values in **`return`** statements (e.g. in `testIf` test case) or calling `explicitLoggingPoint` multiple times (e.g. in a loop as in `testExplicit2`). We merely require that the set of results must be finite. For these cases, all expected results must be specified manually in the annotation. In the evaluation, we treated all expected values as if they were individual test cases. For example,

Table 3. Precision & Recall on the ValBench test suite in %. Tools with an asterisk are exact value approaches.

	JVM cases		Android cases	
Approach	**Prec.**	**Rec.**	**Prec.**	**Rec.**
BlueSeal	2.39	1.95	-	0
COAL	10.34	6.57	0	0
*Harvester**	67.67	54.50	92.86	86.67
JSA	0.02	18.73		
*StringHound**	72.53	16.06		
*ValDroid**	91.81	90.02	94.83	91.67
Violist	0.39	9.00		

`testInputStream3` has 4 different expected values. Harvester reports two values correctly and no spurious value, resulting in a precision of 100 % and a recall of 50 %. As stated in Section 3.4, for tools that only output a singular regular expression, we split `A|B` into two results `A` and `B`.

COAL returned .* for 243 JVM test cases, while *JSA* delivered .* for 117 JVM test cases. The other tools did not deliver such results. We counted .* as false negatives, as they do not contain any information. Note that this initial iteration of test cases only features test cases which have a low and fixed number of specific expected string values. Since each expected value is concrete, a precise regular expression tool ideally returns the most precise regular expression, which is a combination of all expected values using the | operator.

In order to calculate the precision, we calculated $t_p/(t_p + f_p)$, while recall was calculated via $t_p/(t_p + f_n)$. To aggregate precision and recall over multiple test cases, we calculated the sum of t_p, f_n and f_p of all aggregated cases.

Table 3 shows the precision and recall of each tool on the JVM and Android test cases. For the Android test cases, we only list tools which support the Android platform. The regular-expression based approaches *BlueSeal*, *COAL*, *JSA* and *Violist* have lower precision than approaches for concrete values such as *Harvester*, *StringHound* and *ValDroid*. Most tools offer a relatively low (< 50 %) recall on the 372 JVM test cases. Note that *BlueSeal* did not return any value for any of the Android test cases. *COAL* returned wildcards for all Android test cases. *Harvester* and *ValDroid* have a high precision and recall on the 59 Android-specific test cases.

RQ3: What insights can we gain by evaluating value tools on ValBench?

Table 4 shows the precision and recall of all test tools on all test cases. We next discuss how the tools perform on various challenges to showcase room for further research in the field.

```
1        String[] x = new String[2];
2        x[1] = "Right";
3        x[0] = "Wrong";
4        return x[1];
```

Listing 2. ValBench test case `testArraySensitivity`

Listing 2 shows an array sensitivity testcase. *JSA* gives the results `"Wrong"`, `"Right"` and **`null`**, despite that they "give a precise treatment of `String`, `StringBuffer`, and multidimensional arrays of strings" [4]. *COAL* returns `"Wrong"` and `"Right"`. *StringHound* has not given any result for that particular test case, but has correct results for other Array test cases. The other tools give the correct results.

```
1        Fields f = new Fields(), f2 = new
             Fields();
2        f.a = "A";
3        f2.a = "Wrong";
4        return f.a;
```

Listing 3. ValBench test case `objectSensitivityTest`

JSA fails the object sensitivity test in Listing 3, giving `"Wrong"`, `"A"` and **`null`** as result. *COAL* and *Violist* yield the strings `"Wrong"` and `"A"`. *BlueSeal* returns the empty String. The other tools return the correct string. *StringHound* has not given any result for that particular test case, but has correct results for other object sensitivity test cases (e.g., `complicatedFieldTest`). *Harvester* and *ValDroid* return the correct result.

```
1        String start = "Start-";
2        StringBuilder b = null;
3        for (int i = 0; i < 5; i++) {
4                if (i == 0) b = new
                     StringBuilder(start);
5                b.append(i);
6        }
7        b.append("-afterLoop");
8        return b.toString();
```

Listing 4. ValBench test case `simpleLoopTest`

In the loop test case in Listing 4, *StringHound*, *Harvester* and *ValDroid* report the correct string `Start-01234-afterLoop`, *Violist* reports `nullafterLoop` and *JSA* overapproximates with `Start-(.*)`. *BlueSeal* only reports `Start-`. *COAL* returns four different regular expressions: `Start-(.*)-afterLoop`, `NULL-CONSTANT-afterLoop`, `(.*)(.*)-afterLoop` and `NULL-CONSTANT(.*)-afterLoop`.

Challenge: Streams. *JSA* overapproximates on the stream related test cases `testInputStream3` and `testInputStream4`, resulting in 4 true positives for each of both test cases, but 253,948 false positives for each possible ASCII character combination which can be returned by the input stream in this test case. Only *Harvester* and *ValDroid* return the correct values for these cases.

Table 4. Results on ValBench in precision %/recall %. Tools marked by * output exact values. # denotes the number of tests.

Test suite name	#	*BlueSeal*	*COAL*	*Harvester**	*JSA*	*SH**	*ValDroid**	*Violist*
AdvancedString	3	0/0	-/0	20/33.33	-/0	-/0	100/100	-/0
Alias	7	0/0	0/0	85.71/85.71	0/0	100/57.14	100/71.43	0/0
Android	59	-/0	0/0	92.86/86.67	50/11.67	0/0	94.83/91.67	0/0
Annotation	8	0/0	0/0	100/75	0/0	85.71/75	100/100	-/0
ApacheHex	4	0/0	-/0	100/25	-/0	100/25	-/0	-/0
Arithmetic	11	-/0	0/0	100/100	-/0	-/0	100/100	0/0
Array	8	0/0	0/0	100/100	25/62.50	100/25	100/100	80/50
ArrayCopyOf	7	0/0	-/0	85.71/85.71	50/14.29	100/57.14	100/100	-/0
BasicString	10	9.09/9.09	14.29/9.09	70/63.64	100/72.73	100/9.09	100/100	0/0
ByteBuffer	6	0/0	0/0	16.67/16.67	-/0	80/66.67	100/100	-/0
Callee	2	50/50	0/0	50/50	100/50	-/0	100/100	-/0
Caller	2	0/0	0/0	0/0	-/0	100/50	100/100	0/0
Calls	1	0/0	-/0	100/100	-/0	-/0	100/100	-/0
Character	2	0/0	-/0	100/100	50/100	100/50	100/100	-/0
Class	21	0/0	0/0	68.42/61.90	0.78/4.76	100/4.76	85.71/85.71	0/0
Collection	20	0/0	0/0	60/60	-/0	-/0	85/85	0/0
ComplexArray	1	-/0	-/0	100/100	-/0	100/100	100/100	-/0
Composite	3	33.33/16.67	50/100	100/50	50/100	-/0	60/100	50/100
ContextSensitive	4	0/0	-/0	0/0	40/33.33	-/0	100/100	-/0
Crypto-JavaImpl	10	0/0	-/0	37.50/30	0/0	-/0	80/40	-/0
CryptoAPI	1	0/0	0/0	100/100	-/0	-/0	100/100	-/0
Dictionary	4	0/0	-/0	100/75	-/0	100/25	100/100	-/0
DynamicInvoke	1	-/0	-/0	-/0	-/0	-/0	-/0	-/0
Enum	11	0/0	0/0	88.89/72.73	-/0	100/9.09	100/90.91	0/0
EnumSet	12	0/0	-/0	83.33/41.67	50/25	100/25	100/100	-/0
Field	17	0/0	0/0	64.71/61.11	30/16.67	66.67/44.44	90.91/55.56	33.33/11.11
FieldDepth	1	16.67/100	0/0	100/100	-/0	-/0	100/100	0/0
File	5	-/0	0/0	80/80	-/0	-/0	100/100	0/0
FlowSensitivity	1	0/0	0/0	100/100	100/100	-/0	100/100	100/100
HashCodeEquals	6	0/0	-/0	16.67/16.67	6.25/16.67	-/0	33.33/33.33	-/0
IOCommons	8	0/0	-/0	0/0	-/0	-/0	100/100	-/0
If	5	50/14.29	42.86/42.86	100/57.14	44.44/57.14	-/0	58.33/100	36.36/57.14
InheritanceLibrary	1	0/0	-/0	0/0	-/0	-/0	100/100	-/0
InstanceOf	13	0/0	-/0	100/92.31	-/0	100/76.92	100/100	-/0
JSON	5	0/0	-/0	100/100	-/0	-/0	100/100	-/0
Loop	19	0/0	0/0	94.74/94.74	0/0	100/10.53	94.74/94.74	0/0
Map	6	0/0	0/0	100/83.33	-/0	0/0	83.33/83.33	0/0
Math	5	0/0	-/0	100/100	100/80	100/100	100/100	-/0
Merge	1	0/0	0/0	0/0	60/100	-/0	60/100	-/0
Misc	4	0/0	0/0	100/100	0/0	-/0	100/100	0/0
MultiArray	3	0/0	-/0	-/0	-/0	-/0	0/0	-/0
MutableObjects	1	0/0	-/0	100/100	-/0	-/0	100/100	-/0
PatternTest	2	0/0	0/0	100/100	0/0	-/0	100/100	-/0
Polymorphic	10	0/0	-/0	0/0	100/10	-/0	100/90	-/0
Properties	2	0/0	0/0	0/0	-/0	-/0	100/100	-/0
PullParserTest	3	0/0	-/0	100/100	-/0	0/0	100/100	-/0
Recursive	1	0/0	-/0	0/0	-/0	-/0	0/0	-/0
Reflection	23	5.88/8.33	0/0	54.55/50	-/0	0/0	91.67/91.67	5.88/4.17
ReflectionArray	1	0/0	0/0	0/0	-/0	0/0	100/100	-/0
SimpleString	3	0/0	0/0	66.67/66.67	50/33.33	-/0	100/100	0/0
Simplest	1	0/0	-/0	0/0	-/0	-/0	100/100	-/0
Stream	25	0/0	0/0	22.22/19.35	0.00/25.81	57.14/12.90	100/100	-/0
String	6	0/0	0/0	66.67/66.67	0/0	50/16.67	100/100	-/0
StringUtils	11	0/0	-/0	-/0	50/9.09	100/36.36	100/100	-/0
Switch	10	25/4.55	36.17/77.27	100/45.45	61.54/72.73	-/0	100/100	0.20/86.36
TestBitwise	11	-/0	0/0	100/100	-/0	100/9.09	100/100	-/0
Thread	1	-/0	-/0	0/0	-/0	-/0	100/100	-/0
Type	1	-/0	0/0	100/100	-/0	-/0	100/100	-/0
UUID	1	-/0	-/0	100/100	-/0	-/0	100/100	-/0

Out of 29 streams test cases, *ValDroid* returns the correct value for 27, *StringHound* returns the correct value for 4 and *Harvester* for 5 test cases.

Challenge: Complex Crypto Algorithms. ValBench features 10 complex user-code implementations of cryptographic algorithms, which require correct array and loop semantics in order to compute the correct values. Only *Harvester* and *ValDroid* could yield the correct values for some crypto test cases. Furthermore, *ValDroid* yields the correct value for the Blake2s algorithm.

Challenge: Recursion. All approaches except *Harvester* do not seem to support recursion and thus fail all recursive test cases. *Harvester* failed the Fibonacci test case and reports a wrong value (0), but yields the correct value in the `findRecursion` test case.

5 Conclusion

In this paper, we have presented ValBench, a benchmark suite for evaluating value analysis approaches, and a testing framework for running value analysis tools against the benchmark suite. As future work, we plan to further extend ValBench and to improve the automation for the Android-based test cases. Furthermore, we plan to add test cases that use partly user-input to compute values.

6 Data Availability

We have made ValBench fully open source [1] under a permissive license, allowing researchers to use and extend the test suite. We also include all approaches except BlueSeal and Harvester and ValDroid [10] due to their software license. However, we include the precomputed results of all approaches. The results can be replicated when obtaining the approaches from their respective authors. While the GitHub page may feature a newer version of ValBench, we made the original replication package available under the DOI 10.1145/3580436 [11].

Acknowledgments

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D Ernst. 2015. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 669–679.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[3] Ignacio Casso, José Francisco Morales, Pedro López-García, and Manuel V. Hermenegildo. 2020. Testing Your (Static Analysis) Truths. In *International Workshop/Symposium on Logic-based Program Synthesis and Transformation*. https://api.semanticscholar.org/CorpusID:221317992

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 1–18. http://dl.acm.org/citation.cfm?id=1760267.1760269

[5] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001). https://api.semanticscholar.org/CorpusID:1377888

[6] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. 2020. Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Oct 2020). https://doi.org/10.1145/3320269.3384745

[7] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2018. Differentially testing soundness and precision of program analyzers. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018). https://api.semanticscholar.org/CorpusID:54474665

[8] Ding Li, Yingjun Lyu, Mian Wan, and William G. J. Halfond. 2015. String Analysis for Java and Android Applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 661–672. https://doi.org/10.1145/2786805.2786879

[9] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 756–761.

[10] Marc Miltenberger and Steven Arzt. 2024. Precisely Extracting Complex Variable Values from Android Apps. *ACM Trans. Softw. Eng. Methodol.* (feb 2024). https://doi.org/10.1145/3649591

[11] Marc Miltenberger and Steven Arzt. 2024. Valbench Artifact. https://doi.org/10.1145/3580436

[12] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 77–88. https://doi.org/10.1109/ICSE.2015.30

[13] Damien Octeau, Daniel Luchaup, Somesh Jha, and Patrick McDaniel. 2016. Composite Constant Propagation and its Application to Android Program Analysis. *IEEE Transactions on Software Engineering* 42, 11 (2016), 999–1014. https://doi.org/10.1109/TSE.2016.2550446

[14] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *NDSS*.

[15] J. D. Vecchio, F. Shen, K. M. Yee, B. Wang, S. Y. Ko, and L. Ziarek. 2015. String Analysis of Android Applications (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 680–685. https://doi.org/10.1109/ASE.2015.20

[1] https://github.com/Fraunhofer-SIT/ValBench

Static Analysis for Transitioning to CHERI C/C++

Irina Dudina
irina.dudina@ed.ac.uk
The University of Edinburgh
Edinburgh, United Kingdom

Ian Stark
ian.stark@ed.ac.uk
The University of Edinburgh
Edinburgh, United Kingdom

Abstract

We describe and evaluate custom static analyses to support transitioning C/C++ code to *CHERI* hardware. CHERI is a novel architectural extension, implemented for RISC-V and AArch64, that uses *capabilities* to provide fine-grained memory protection and scalable software compartmentalisation. We provide custom checkers for the Clang Static Analyzer to handle capability alignment, copying through memory, and manipulation as integers; as well as evaluating these on a sample of packages from the CheriBSD ports library. While the existing CHERI toolchain can recompile large code collections for the platform with only a few source changes, we demonstrate that static analysis can help to identify where and what those changes must be to avoid later runtime faults.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; *Software testing and debugging*; • **Theory of computation** → *Semantics and reasoning*.

Keywords: Static Analysis, CHERI, software porting

ACM Reference Format:
Irina Dudina and Ian Stark. 2024. Static Analysis for Transitioning to CHERI C/C++. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3652588.3663323

1 Introduction

CHERI (Capability Hardware Enhanced RISC Instructions) is a contemporary capability architecture with multiple hardware implementations, including for RISC-V and AArch64, that provide low-overhead hardware support for memory protection and software compartmentalisation [15, 16]. At the software level this is supported by a C/C++ dialect known as *CHERI C/C++* with the distinctive feature that all program pointers (explicit and implied) are implemented with architectural capabilities [17].

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663323

On CHERI hardware these capabilities are a complete replacement for pointers, extending purely numerical addresses with upper and lower bounds on access, permissions, and other fields for features like compartmentalisation and control-flow integrity (Fig. 1). These make capabilities larger than the underlying address width, typically 128 bits for a 64-bit architecture.

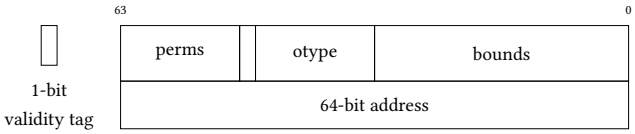


Figure 1. 128-bit CHERI Capability

Every capability has a software-inaccessible out-of-band *validity tag* that follows it through registers and memory. This prevents forgery, with CHERI providing hardware-enforced limitations on modification and use of capabilities.

One significant benefit of the CHERI architecture is ease of transition: large amounts of existing C/C++ code will recompile with minimal modification and immediately gain hardware-backed memory protection. CHERI design choices mean compilers can routinely implement C/C++ pointers with capabilities, and there are Clang/LLVM and GCC versions supporting this. The precise meaning of the CHERI-C/C++ dialect is given by an executable mechanised semantics [21] as an extension of the Cerberus C semantics [11].

Moving existing C/C++ code to a CHERI platform provides strong guarantees of memory safety, but may still require code changes to ensure smooth execution — most notably, where minor violations of memory safety have previously escaped detection. Certain legacy idioms also interact poorly with capabilities, and a very few need rethinking entirely. Looking forward, embracing capabilities opens up opportunities for robust compartmentalisation and other protection, but again requiring work to integrate these new features.

Tackling portability across architectural change is not new: moving from 32-bit to 64-bit architectures being a landmark example. It's therefore familiar that a key source of trouble is where C code makes assumptions, perhaps unwittingly, that are true in its original setting but no longer hold in the new context. For example: (1) assumptions on type size and alignment, in particular that certain types can be stored interchangeably; (2) assumptions on pointer representation,

used to manually inspect or modify address values; (3) assumptions on type representation, hard-coded into "magic constants". All these show up in transitioning C/C++ code to CHERI. Some are reliably detected in compilation, with suitable errors or warnings. What remains, though, presents a classic opportunity for static analysis: subtle errors, scattered sparsely across large volumes of code, where any eventual fault may be distant from its original cause. Some may show up in testing; but even where packages have high-coverage test suites those can share the same architectural assumptions and portability problems. To address this we demonstrate custom static checkers that can identify up-front the crucial changes necessary to reliably adapt C/C++ codebases for CHERI.

1.1 Adapting Code for CHERI C

The CHERI C/C++ Programming Guide [17] provides extensive commentary on potential issues in porting existing code to a capability platform. We pick out here the specific problems we address with our work.

Capability size and alignment. In CHERI C all pointers are implemented as capabilities, typically 128-bit wide and requiring 128-bit alignment. This can cause problems when code assumes pointers are the same size as some other, smaller, type. For example, Listing 1 below shows a pattern we have seen in several real projects.

Listing 1. Capability size and alignment

```
unsigned long ret, *retPtr = &ret;
pthread_join(threadId, (void**)retPtr);
```

The address of local variable **unsigned long** ret is passed as a **void**** value to the second argument of pthread_join function, which will then write a **void*** value to this address. As long as **unsigned long** has size and alignment requirements at least as strict as **void***, this code will work fine. However, for CHERI this is no longer true: attempts to write a 128-bit **void*** capability value to the narrower **unsigned long** stack allocation will result in a capability bounds fault.

Copying objects containing capabilities. To maintain integrity, CHERI capabilities must be copied as a whole using special capability load/store instructions that propagate validity tags. In CHERI C/C++ these instruction are emitted when copying evident capability values (pointers and **(u)intptr_t**). However, custom implementations of general-purpose memory copy routines may copy data in smaller units (e.g. memcpy from Listing 3). When used for copying objects containing capabilities, these will silently invalidate the capabilities leading to errors downstream.

Pointers as integers. This category includes using non-capability integer types to hold pointer values and manipulating capabilities with bitwise and arithmetic operations on **(u)intptr_t** values.

In general, adaptation of contemporary C/C++ source code to CHERI C/C++ is quite straightforward and requires very small changes to the source code (e.g. 0.026% LoC reported for porting a desktop stack based on X11 and KDE [14]). Low-level system and heavily platform-specific C/C++ code tends to require more porting effort [2, 6, 10, 13], as does some non-standards-compliant code.

While many CHERI-compatibility issues are detected and well explained by the compiler, there remain certain classes of issue that it does not report and surface only at runtime. Uncovering these during porting depends on test coverage, which can be haphazard when the tests naturally predate CHERI. A further complication arises when a developer responds to compiler warnings with changes that silence the warning rather than fix the issue. For example, introducing a type cast that may not be valid under CHERI but is enough to make the warning go away.

Taken together these mean that remaining CHERI-related faults in ported software may be small in number, widely spread across large bodies of code, possibly concealed behind attempted fixes, and yet cause immediate software failure if triggered on execution. This is where a static analysis tool add clear value: with less strict performance and false-positive requirements than a compiler it can perform wider and more thorough analysis resulting in better coverage overall.

We have implemented our analysis in the Clang Static Analyzer (CSA) [9], a static analysis tool for C, C++, and Objective-C programs within the Clang/LLVM project. CSA provides two analysis methods: static symbolic execution for path-sensitive inter-procedural analysis and AST matchers for simple AST pattern-matching checks. Both levels are extendable with custom checkers, which we provide. In addition, we have identified some of the existing CSA checkers as especially important in the context of porting code to CHERI: 1) Use of hardcoded address (alpha.core.FixedAddr) 2) Subtraction of pointers to distinct objects (alpha.core.PointerSub) 3) Allocator **sizeof** operand mismatch (unix.MallocSizeof).

1.2 Related Work

As was noted above, the CHERI Clang compiler provides its own diagnostics [–Wcheri] that flag up many types of CHERI-incompatible code patterns [12].

The *CHERIseed* tool is a software emulation of capabilities that supports running CHERI-C/C++code on a conventional (non-capability) host machine [1]. This provides a dynamic analysis of portability issues by exposing unsafe code that could fault on real CHERI hardware.

The ESBMC model-checker has an extension *ESBMC-CHERI* to support C program verification on CHERI-enabled platforms [3]. This aims to statically detect memory safety violations and compatibility issues for CHERI-C, with a capability-aware memory model and CHERI Clang integration.

Listing 2. Underaligned capability storage

```
typedef struct S_s {
    /* other fields */
    uint8_t extra[1];
} S;

S* alloc(ssize_t extra_size, void** extra) {
    // ... 73 lines skipped ...
    *extra = &g->extra[0];
    return g;
}

typedef struct T_s {
    /* has pointer fields */
} T;

T* p; /* T requires 16-byte alignment */
S* q = alloc(sizeof(T), (void**)&p);
```

Several studies have investigated the application of static analysis techniques to facilitate seamless API migration. Desynchronizer tool relies on static analysis to determine where calls to synchronous JavaScript API functions can be replaced with their asynchronous counterparts [5]. Meditor, an API migration tool for Java libraries, employs static analysis to identify the control and data dependencies between statements, that allows it to align many-to-many statements between versions and infer API replacement operations together with other related editing operation [19].

2 Capability Alignment

2.1 Capability Alignment Restrictions

As was mentioned before, all pointers in CHERI C must only be stored in memory at addresses aligned to a 128-bit boundary. This is due to the fact that capability tags that are responsible for ensuring the unforgeability of capabilities apply only to memory locations that are capability-aligned and capability-sized. Therefore, storing capabilities at underaligned addresses will result in a trap or, in case of copying capabilities with memcpy, the stored copy having its validity tag stripped. Similarly, reading a capability from an underaligned address will return an invalid capability and therefore is a clear indication of an error.

Consider the code snippet from Listing 2 that we extracted from a library written in C++. Function alloc is called to allocate a memory for the structure of type T, which contains pointer fields and therefore requires capability alignment. It writes an address of the allocated memory by the pointer provided as a second argument. In this case, this is a pointer to the local variable p. The implementation of the alloc function allocates a structure S with a flexible array member **uint8_t** extra and passes the address of this field as a start address of the allocated memory available for the user. As long as **uint8_t** type has a trivial alignment requirement, this address may not be properly aligned for storing structure T there, in this case any object of type T stored there will have its capabilities invalidated.

2.2 Impact of Misaligned Pointers

Accessing objects in memory at misaligned addresses is a problem that is not unique to CHERI C. On a conventional architecture, underaligned access can result in a fault, affect atomicity or performance. However, for some C programs, even when the proper object alignment is not ensured by using the correct type with appropriate alignment requirements, this issue may not manifest in runtime due to several reasons. First, runtime alignment of the object may be enforced by platform-specific ABI alignment guarantees (e.g. on static variables). Second, correct object alignment may be an implication of unintentional and, therefore, fragile properties of a current data layout (e.g. structure field alignment may be guaranteed by the combination of aligned field offset and parent structure alignment, regardless of the field type). Similarly, other features of memory layout controlled by the compiler (e.g. static variables order) may prevent the problem from manifesting, but minor changes in the source code or compilation process may break this. At the same time, even when manifesting in runtime, the issue may remain undiscovered on some architectures due to subtle ill effects. C compilers report –Wcast–align warning for simple cases, but it's too noisy and is turned off by default.

2.3 Detection of Misaligned Pointers

To detect this bug, we developed a symbolic execution checker in CSA. For every analysed path, we track the *smallest guaranteed alignment* for pointers and trailing zeroes count for integers.

For the case of tracked allocations (*Typed Memory Regions* in CSA), we use the type alignment of the objects allocated on stack, heap, or in static memory, taking into account alignas and similar attributes, if present for this allocation. For a symbolic pointer, CSA creates a *Symbolic Region* to represent the memory block pointed to by this symbolic pointer [20]. For this case, we cannot always rely on the pointer type, as we did not track the actual allocation of this object. We rely on pointer type only for pointer variables, which are function arguments at the top of the analysed call stack or global variables, as these variables can be seen as arbitrary input values to the analysed function. Exceptions are **void*** and **char*** pointer arguments and variables, as they are often used to pass pointer values of more than one type, depending on the other input values.

We track trailing zeroes count for integers to propagate alignment values for pointer arithmetic, including pointer shifts and manual pointer alignment via binary operations (e.g. ALIGNUP macro).

We report a warning when a pointer value with the smallest guaranteed requirement A is cast to a pointer to the type with an alignment requirement stricter than A or stored as such pointer (e.g. see Listing 2).

2.4 CHERI-Specific Issues

There are aspects of this issue specific to CHERI C. First, in CHERI C, pointers require 16-byte alignments, which often breaks legacy code assumptions — aligning to **sizeof double**, **sizeof uint64_t**, and sometimes even **sizeof long double** is not enough. Secondly, capability-aware unaligned load or store of a capability will generate a run-time hardware exception, therefore, previously undetected misalignment, sometimes almost harmless on a conventional architecture, will result in a crash on CHERI. Finally, unlike other types, capabilities cannot be moved around as a sequence of bytes and stored at a misaligned address, even temporarily. Thus, attempting to memcpy a capability through unaligned memory will result in tag stripping during the first copying (storing); copying it back to a properly aligned address will not bring the tag back. An actual crash will happen later during an attempt to use this capability for memory access, making this issue very hard to investigate.

Therefore, we also need to track memory regions that may contain capabilities and report a warning when we detect a memory copy to/from such region from/to an underaligned memory region. To track memory regions potentially containing capabilities, we rely on pointer types for both *Symbolic* and *Typed* regions and propagate this property through memcpy-like library calls.

To be able to detect the usage of underaligned allocations for storing capabilities, like in the example in Listing 2, we need to detect a single intra-procedural path that contains both the initialization of the pointer with a misaligned value and usage of this pointer as capability storage. In practice, these two events are often distant; a single path covering both of them may be very long and span across multiple functions or even modules, making it very difficult to analyse. However, even if we do not have a call instruction in the analysed path that explicitly uses the alloc function to allocate memory for a capability-aligned structure, we can still suspect the implementation of alloc in being not CHERI-friendly. We can argue that *extra pointer, passed by address for the initialization to this function, has type **void***, which indicates that it may be used to store arbitrary data there — including objects containing capabilities. As a heuristic, we report assigning addresses of underaligned memory with unknown content (potential storage allocations) to **void*** variables that are top-level function arguments, global variables, or structure fields (potential pointers to capability storage).

Listing 3. Tag-stripping memcpy

```
1 #define BIGBLOCKSIZE (sizeof(long) << 2)
2 void *memcpy (void *dst, const void *src, size_t len) {
3   if (!TOO_SMALL (len) &&
4       !UNALIGNED (src, dst)) {
5     long *aligned_dst = (long*)dst;
6     const long *aligned_src = (long*)src;
7     while (len >= BIGBLOCKSIZE) {
8       *aligned_dst++ = *aligned_src++;
9       *aligned_dst++ = *aligned_src++;
10      *aligned_dst++ = *aligned_src++;
11      *aligned_dst++ = *aligned_src++;
12      len -= BIGBLOCKSIZE;
13    }
14  else { ... }
15 }
```

Although this approach leads to a significant number of false warnings (see section 6 for evaluation), it allows detection of a misaligned allocation even if it was never explicitly used to store or load capabilities, but should support this case.

3 Capability Copy

The only way to construct a valid CHERI capability is from another valid capability: known as the *provenance validity* rule. Therefore, whenever there is a need to copy, move, or swap a capability, it must be copied as a whole using capability-aware instructions that preserve the validity tag. The compiler will emit such instructions whenever capability-sized and capability-aligned data is explicitly copied. However, if memory that contains a valid capability is copied in parts, piecemeal, then that will strip the tag to give an invalid capability; and any general-purpose routines that could potentially copy or move capabilities (memcpy, realloc, qsort, etc.) need to take care with this. We developed a capability copy checker aimed at detecting functions that move data in a way that could potentially result in a tag-stripping capability copy.

We report a capability copy warning for a copy operation iff

1. memory chunk of size S is loaded from memory pointed by a *source* pointer (+offset),
2. this memory chunk is then stored to memory pointed by a *destination* pointer (+offset),
3. *source* and *destination* point to memory regions that hold capabilities, or may hold capabilities (discussed below),
4. S is less than capability size,
5. *source* and *destination* can be capability-aligned,
6. copying is performed in a loop or a sequence of assignments, that can be long enough to copy a capability.

Conditions 5 and 6 are required to avoid reporting false warnings for the typical implementation of memcpy, where the case of unaligned or small copy is handled separately (line 14 in Listing 3).

As in the case of detecting underaligned generic storage, we cannot practically rely on always having a call to memory-copying function with a request to copy capabilities as a confirmation that this function is indeed intended to safely move capabilities around. In this case this problem can be even more pronounced, as such functions are often implemented as library functions, therefore such invocation may just not exist in the analysed codebase, even less likely in the same module. Therefore, as with generic storage case, we assume that top-level **void*** function arguments can potentially point to objects containing capabilities.

We also had a hypothesis that this assumption may be extended for **char*** arguments as well, as **char*** pointers can be used to iterate over arbitrary data. The obvious exception would be functions that expect C strings as **char*** arguments. Therefore we implemented a detector that would disable this assumptions if there are signs of C-string-like manipulation involving this pointer (e.g. checking bytes against character literals or NULL-terminator, passing pointer to string processing functions like strcpy, etc.). However, this approach did not bring a significant amount of true warnings, while adding a lot of false reports related to C strings (we got around 10 times more warnings, only 2% of which were true alarms). String detection could be improved, but the amount of true issue detected using this assumption does not provide a motivation for that.

Since this checker detects loading chunks of a capability value representation, it also reports situations, where individual bytes of a capability are used in arithmetic. This may be an indication of using pointer values to compute hash value or for the similar purposes. CHERI C/C++ Programming Guide recommends extracting address value from a capability and using it instead for the sake of performance.

4 Capabilities as Integers

According to the C standard, **(u)intptr_t** is an integer type capable of holding object pointers. Therefore, in CHERI **(u)intptr_t** must be implemented with capabilities as well. If **(u)intptr_t** holds an integer value rather than a pointer value, then it is represented with so-called NULL-*derived capability*, which is an untagged (invalid, not dereferenceable) capability with its address value set to the given integer value.

For binary arithmetic or bitwise operations on **(u)intptr_t** values, CHERI-C applies a *single-provenance rule* that the resulting capability has its validity derived from just one of the arguments. Unless one of the arguments is a non-capability (integer) type promoted to **(u)intptr_t**, the compiler cannot know from which argument to derive the resulting capability and will emit a [–Wcheri–provenance] warning. Currently for these cases the result will be derived from a left-hand-side by default. Fixing this warning requires explicitly telling the compiler what side to pick by casting the other argument to a non-capability type. A static analyser can help resolve these warnings by tracking the origin of the **(u)intptr_t** values, highlighting which will be a valid capability, and suggesting the appropriate fix. We developed a checker that tracks integer and pointer values stored as **(u)intptr_t** type and reports the following warnings for **(u)intptr_t** binary operations with ambiguous provenance source.

Both operands are derived from pointers. This code is not compatible with CHERI and should be rewritten. Real-world examples include using XOR of pointers for branchless select and XOR-linked lists.

LHS is* NULL*-derived, and RHS is derived from a pointer. Under compiler defaults, the resulting capability will be NULL-derived. The RHS should likely be used to derive the result instead.

Other cases. Those are fine with the default compiler behaviour and are not reported if [–Wcheri–provenance] is disabled in the original project build.

Another issue is related to the accidental loss of tag when casting a valid capability (pointer or **(u)intptr_t**) to a non-capability integer type (e.g. **ptrdiff_t** or **size_t**). If the value is converted back to a capability type (either directly or via **(u)intptr_t**) then it will become a NULL-derived invalid capability. The compiler can detect this in the simplest case of a direct cast; static analysis can report more general cases of any NULL-derived **(u)intptr_t** capability being cast to a pointer type, which will always be invalid.

5 Insufficient Pointer Size Check

We also implemented a simple AST-checker to detect the code patterns, where the **sizeof** of a pointer is explicitly checked, but the case of 128-bit pointers is not considered. In this case, it technically falls in the "default" (e.g. **else**) branch and gets processed as the default pointer size, which is often assumed to be 64 bits, leading to incorrect behaviour.

6 Implementation and Evaluation

The *CheriBSD* operating system is an extension of FreeBSD that takes advantage of capability hardware to implement memory protection and software compartmentalisation. It supports CHERI extensions to both RISC-V and AArch64 (Arm Morello) and has matured through several releases: we work with CheriBSD 23.11 [4, 7].

CheriBSD provides versions of third-party software in the *CheriBSD ports* collection, a fork of the FreeBSD ports collection with CHERI-related changes. It also uses the FreeBSD concept of *compatibility layers* to allow programs compiled

Table 1. Warning evaluation

Warning	Total	Unique	True	TP-rate
Cast increases required alignment to capability alignment	103	37	29	78%
Not capability-aligned pointer stored as capability pointer	62	30	11	37%
Copying capabilities through underaligned memory	28	21	7	33%
Tag-stripping copy of capability	14	11	4	36%
Part of capability value used in binary operator	9	1	1	100%
CHERI-incompatible pointer arithmetic	0	0	0	N/A
Capability derived from wrong argument	0	0	0	N/A
Binary operation with ambiguous provenance	11	9	9	100%
Invalid capability used as pointer	339	11	11	100%
Only a limited number of pointer sizes checked	1	1	1	100%

for different ABIs to coexist within one system. It supports CheriABI (aka pure-capability ABI), which permits memory accesses only via capabilities, and *hybrid* ABI, which allows raw legacy pointers for incremental software adaptation. The current CheriBSD ports collection has ~9,000 pure-capability packages and ~24,000 hybrid ABI packages [18].

We implemented our checkers within the Clang Static Analyzer (all source on GitHub [8]) and added this tool as a port to the CheriBSD ports collection. For evaluation, we ran the analyzer on a selection of pure-capability packages from the CheriBSD ports collection. Most of these packages had been recompiled for CheriABI without extensive testing or necessarily even fixing all the CHERI compiler warnings.

We selected 56 pure-capability packages from the databases and lang categories (as we believe those might be not trivial to port) that are mainly written in C/C++ (2,544 KLOC of C/C++ total). According to our experience, project build+analysis is 5.5 times slower than project build alone on average. The results of the warning assessment are presented in Table 1. The first column lists warning types, column *Total* gives the overall number of warning of this type. For the third column *Unique*, multiple instances of the same code pattern spread across the project code were considered duplicates and counted as a single instance. Columns *True* and *TP-rate* give the absolute and relative number of unique true warnings, respectively.

Although this codebase includes some C++, and our tool does analyze that, in practice most of the warnings are for C code. None of the portability issues we check are C++-specific, and most occur in low-level code anyway — true C++ will likely avoid them.

6.1 Capability Alignment

Warnings of the first group were produced by the capability alignment checker. Warnings that report alignment-increasing casts are the most reliable from this group. For this evaluation, we only included issues related to capability alignment, but this checker produces warnings for all pointer types.

Reports of copying capabilities through underaligned memory and for storing a non-capability-aligned pointer as a capability pointer are often fired simultaneously, since these two events are often adjacent. The true positive ratio for these warnings is not as good as for cast warning type, owing to the assumption that "**void**∗ *may be a pointer to capability*". However, this heuristic can identify alignment issues in cases that are hard to detect otherwise: for example, when the possibility of storing pointers at this address is implied but never exercised in the analysed code.

6.2 Capability Copy

We encountered very few warnings related to tag-stripping capability copying, those were for the custom implementations of qsort and memcpy-like functions, as expected, and mainly originated from a single project. Warnings that were classified as false positives are reported for functions that accept **void**∗ arguments and perform partial copying but are not intended to be used with capabilities.

Additionally, 9 instances of the same pattern of a hash value calculation of a capability representation were found and correctly reported by this checker.

6.3 Capabilities as Integers

Pointer vs. integer conversion is a pattern specific only to particular projects. Typically, there is either no warning of this type in the project or dozens (sometimes hundreds) of them reported for the same idiom. These warnings are primarily true positives, but the majority of these cases are simple enough and most likely intentional.

6.4 Insufficient Pointer Size Check

For the pattern with insufficient pointer size check, there was only one match and that was a true alarm.

7 Conclusion and Future Work

Our contributions are the following.

- Four capability-specific CSA checkers: for capability alignment, tag-stripping capability copying, pointer vs. integer conversions, and insufficient pointer size checks; and identification of three existing CSA checkers essential to CHERI adaptation.
- Implementation and evaluation of these checkers against samples from the CheriBSD ports library.

We believe that the capability alignment checker is the most useful of the implemented checkers, as it has more warnings, those are not reported by a compiler reasonably well, and often hard to investigate when they cause a run-time fault.

We plan to extend this tool to include more checkers and in particular, to go beyond simply picking up problems to identifying areas where code can directly take more advantage of capabilities. For example, coding patterns for custom allocators where explicitly narrowing capability bounds can improve memory safety. Another area is subobject bounds, where there exist compiler switches to emit very tight capabilities but in practice this requires precise code changes for reliable porting.

Acknowledgments

This work was supported by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme, to deliver a DSbDtech enabled digital platform (grant 105694).

References

[1] Arm. 2022. CHERIseed – port effortlessly to CHERI. https://www.morello-project.org/resources/cheriseed-port-effortlessly-to-cheri/. Accessed: (2 May 2024).

[2] Jacob Bramley, Dejice Jacob, Andrei Lascu, Jeremy Singer, and Laurence Tratt. 2023. Picking a CHERI Allocator: Security and Performance Considerations. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*. Association for Computing Machinery, New York, NY, USA, 111–123. https://doi.org/10.1145/3591195.3595278

[3] Franz Brauße, Fedor Shmarov, Rafael Menezes, Mikhail R. Gadelha, Konstantin Korovin, Giles Reger, and Lucas C. Cordeiro. 2022. ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (, Virtual, South Korea,) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 773–776. https://doi.org/10.1145/3533767.3543289

[4] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 379–393. https://doi.org/10.1145/3297858.3304042

[5] Satyajit Gokhale, Alexi Turcotte, and Frank Tip. 2021. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 160 (oct 2021), 27 pages. https://doi.org/10.1145/3485537

[6] Brett Gutstein. 2022. *Memory safety with CHERI capabilities: security analysis, language interpreters, and heap temporal safety*. Technical Report UCAM-CL-TR-975. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-975

[7] SRI International and the University of Cambridge. 2023. *CheriBSD 23.11 Release Notes*. https://www.cheribsd.org/release-notes/23.11

[8] Irina Dudina. 2024. *CHERI CSA*. https://github.com/rems-project/llvm-project

[9] LLVM Project. 2016. *Clang Static Analyzer*. https://clang-analyzer.llvm.org/

[10] Duncan Lowther, Dejice Jacob, and Jeremy Singer. 2023. Morello MicroPython: A Python Interpreter for CHERI. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (, Cascais, Portugal,) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 62–69. https://doi.org/10.1145/3617651.3622991

[11] Kayvan Memarian. 2022. *The Cerberus C semantics*. Ph. D. Dissertation. Apollo - University of Cambridge Repository. https://doi.org/10.17863/CAM.96843

[12] Alexander Richardson. 2020. *Complete spatial safety for C and C++ using CHERI capabilities*. Technical Report UCAM-CL-TR-949. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-949

[13] Kui Wang, Dmitry Kasatkin, Vincent Ahlrichs, Lukas Auer, Konrad Hohentanner, Julian Horsch, and Jan-Erik Ekberg. 2024. Cherifying Linux: A Practical View on using CHERI. In *Proceedings of the 17th European Workshop on Systems Security* (, Athens, Greece,) *(EuroSec '24)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/3642974.3652282

[14] Robert N. M. Watson, Ben Laurie, and Alexander Richardson. 2021. *Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem*. Technical Report. Capabilities Limited. https://www.cl.cam.ac.uk/research/security/ctsrd/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf

[15] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. 2019. *An Introduction to CHERI*. Technical Report UCAM-CL-TR-941. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-941

[16] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)*. Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-987

[17] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. 2020. *CHERI C/C++ Programming Guide*. Technical Report UCAM-CL-TR-947. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-947

[18] Konrad Witaszczyk. 2023. CheriBSD Ports and Packages: Pure-capability third-party software for Arm Morello and CHERI-RISC-V CheriBSD. *FreeBSD Journal* (March 2023), 12–22. https://freebsdfoundation.org/wp-content/uploads/2023/05/

CheriBSD_ports.pdf

[19] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, 335–346. https://doi.org/10.1109/ICPC.2019.00052

[20] Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. A Memory Model for Static Analysis of C Programs. In *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 535–548. https://doi.org/10.1007/978-3-642-16558-0_44

[21] Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alexander Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2024. Formal Mechanised Semantics of CHERI C: Capabilities, Undefined Behaviour, and Provenance. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (, La Jolla, CA, USA,) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 181–196. https://doi.org/10.1145/3617232.3624859

Received 2024-03-04; accepted 2023-04-19

Misconceptions about Loops in C

Martin Brain
City, University of London
London, United Kingdom
martin.brain@city.ac.uk

Mahdi Malkawi
City, University of London
London, United Kingdom
mahdi.malkawi@city.ac.uk

Abstract

Loop analysis is a key component of static analysis tools. Unfortunately, there are several rare edge cases. As a tool moves from academic prototype to production-ready, obscure cases can and do occur. This results in loop analysis being a key source of late-discovered but significant algorithmic bugs. To avoid these, this paper presents a collection of examples and "folklore" challenges in loop analysis.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Automated static analysis**.

Keywords: Loop Analysis, Software Verification, Static Analysis

ACM Reference Format:
Martin Brain and Mahdi Malkawi. 2024. Misconceptions about Loops in C. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3652588.3663324

1 Introduction

When developing a new static analysis tool, there is little distance between the developers and users. Often they are the same. Diagnosing and debugging issues with the tool is no more difficult than regular development. The distance between users and developers grows as a tool becomes more successful. Identifying faults requires the users to be willing and able to file bug reports. Reproduction may require non-disclosure agreements or travel to customer premises. Confirming fixes may require a full release and deploy cycle. Bugs that would have taken days to fix in early development now take weeks, highlighting the importance of early bug detection.

Bugs in static analysis tools vary in terms of severity and impact. Crashes early in inputs can normally be isolated relatively easily. Correctness bugs from mistakes in the core algorithms are some of the most challenging to find and fix. The worst of these are cases with several edge cases, and it *seems* like only one needs to be fixed. This can lead to a cycle of bug reports and fixes, consuming time, budget and user goodwill frighteningly quickly.

This paper addresses bugs that arise from tool developers' mistaken beliefs and incorrect assumptions about loops and their structure. Experience with various tools including CBMC [17, 18, 21], SPARK [16], 2LS [4] and goto-analyzer [14, 23] at different stages of development has shown that these are a persistent source of complex and late-detected bugs. Loops that do not meet the developer's assumptions are often fixed with a simple workaround for the specific example, rarely solving the entire problem, leading to additional bugs with precarious fixes and ballooning code complexity.

This paper makes the following contributions:

- Section 2 reviews several definitions of loops using different program representations and gives examples highlighting their differences.
- Section 3 explains mistakes (marked ↯) that the authors and others, have made about the structure of loops. These were identified as root causes of correctness bugs found in several tools, with many leading to severe and time-consuming bugs.
- Section 4 provides recommendations for handling loop analysis, bug prevention and checks to avoid some mistakes outlined in section 3.

2 What Are Loops?

To illustrate our definitions we use the "first" loop in C [11]:

```
while (fahr <= upper) {
  celsius = (5.0/9.0) * (fahr-32.0);
  printf("%4.0f %6.1f\n", fahr, celsius);
  fahr = fahr + step;
}
```

This has all the classical features of a loop [19]:

1. The program first checks the *loop condition*...
2. ...which is the only way out of the loop.
3. If the condition is true, the *loop body* is run.
4. The program jumps back to the top of the loop after the loop body finishes and everything repeats.

Unfortunately none of these is true for all loops.

When a program is represented as a parse tree, a loop is any instance of the loop grammar rules, such as the `while` or `for` rules. In the example, we have a single `while` loop, with a loop condition (`fahr <= upper`) and a loop body.

SOAP '24, June 25, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663324

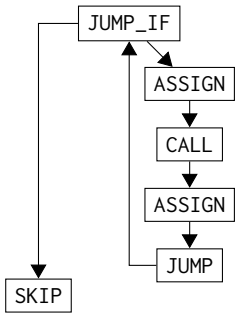


Figure 1. A control flow graph for the first loop example

An alternative representation for the program is a list of instructions. Jumps transfer execution to a different part of the instruction list, dependent on some condition or unconditionally forgoing execution of instructions between the jump and target label. The running example is representable using six instructions:

```
A: JUMP_IF  !(fahr <= upper), B
   ASSIGN   celsius, (5.0/9.0) * (fahr-32.0)
   CALL     printf, "%4.0f %6.1f\n", fahr, celsius
   ASSIGN   fahr, fahr + step
   JUMP     A
B: SKIP
```

The first jump is a conditional, the second is unconditional and is a *backwards jump* as the target is earlier in the list.

A third representation is a control flow graph (CFG) [3], which abstracts the list of instructions into a directed graph. Each instruction becomes a node. Conditional jumps have two edges: to the jump target and to the next instruction. Unconditional jumps have a single edge to their target. All other instructions (except the last) have a single edge to the next instruction. Figure 1 gives the example's CFG.

There are two widely used definitions of loops in CFGs: a natural loop and a cycle. Following [2, 10] we describe a *back edge* as any edge in the graph from $t \rightarrow h$ where every path from the start of the function to t passes through h. We call h the head of the loop and t the tail of the loop. The corresponding *natural loop* is h plus the set of nodes that can reach t without passing through h. Natural loops with the same head are merged and regarded as a singular loop. Following [9] we describe a *cycle* as a maximal strongly connected component (SCC). Cycles are more general and can contain multiple entry points, unlike natural loops.

Loop Definitions Are Not Equivalent. As all four definitions seek to describe the same fundamental structure, it is not unreasonable to treat them as the same or at least "essentially equivalent". For example, deductive verification tools provide annotations to allow the programmer to state loop invariants. Abstract interpreters and counterexample-guided inductive synthesis based tools also compute loop invariants. Deductive verification tools typically use the parse tree definitions of loops, whilst other tools use the CFG or list definitions. These definition mismatches and differing properties of loops can cause considerable misunderstanding and bugs when trying to combine different kinds of tools [4, 21]. This section presents some sources of these mistakes.

↯ Loop conditions are the same in all definitions. Loop conditions are a feature of the parse tree representation as they are a syntactic feature of most languages. Unfortunately, loop conditions are more fragile than often assumed. In the following code, the parse tree representation has a single loop condition `true` suggesting the loop does not terminate:

```
do {
  open_socket();
  if (connect() == SUCCESS) { break; }
  close_socket();
} while (1);
```

In languages with alternative means of exiting a loop, the syntactic loop conditions are sufficient but not necessary for loop termination . The other routes must be detected and added to precisely reason about the loop exit. See Section 3.3 for more details about the subtleties involved.

The parse tree loop conditions are not directly definable in the instruction lists and CFGs. Compare the following two programs that make use of "shortcut" operators :

```
while (A() && B()) {
  green();
}
```

```
while (A()) {
  if (!B()) { break; }
  green();
}
```

The CFGs are equivalent but the parse tree loop conditions are {`A && B`} and {`A`} (or {`A`,`B`} if augmented as above).

It is tempting to think there is a CFG definition of a loop condition along the lines of "A conditional branch where one branch is outside of the SCC", but this does not match the parse tree definition:

```
while (A() || B() || C()) {
  if (D()) { break; }
  pink();
  if (E()) { break; }
}
```

Here {`C`, `D`, `E`} would be "CFG loop conditions" but {`A || B || C`} (and possibly `D` and `E`) would be parse tree loop conditions.

↯ Loop bodies are the same in all definitions. Similarly to loop conditions, loop bodies are a syntactic feature in most languages and are clearly defined. They do not necessarily correspond to what instructions are regarded to be in the loop in other representations. For example, consider:

```
while (choose()) {
  if (choose()) { red(); break; }
  else if (choose()) { goto out; }
}
```

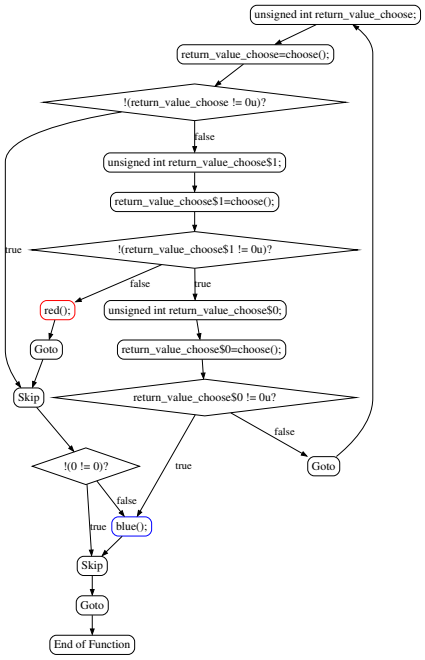


Figure 2. In the CFG, neither `red` or `blue` are in the loop

```
if (0) { out: blue(); }
```

In the parse tree representation, `red` is in the loop body and `blue` is not. The CFG given in Figure 2 shows that neither `red` or `blue` are in the loop as they cannot reach themselves. They are also largely indistinguishable suggesting that the parse tree notion of loop body is not expressible in the CFG.

↯ Back edges are the same as backwards jumps. Backwards edges in a list representation are not guaranteed CFG back edges. Consider the following list representation:

```
    goto A;
B : second();
    assert(counter == 2);
    goto C;
A : first();
    assert(counter == 1);
    goto B;
C : third();
    assert(counter == 3);
    return;
```

In the list representation, `goto B` gives a backwards jump but in the CFG representation it is not a back edge.

↯ The number of loops is the same in all definitions. Compare the following programs:

```
do {
  do {
    blue();
  } while (A());
} while (B());
```

```
do {
  blue();
} while (A() || B());
return;
```

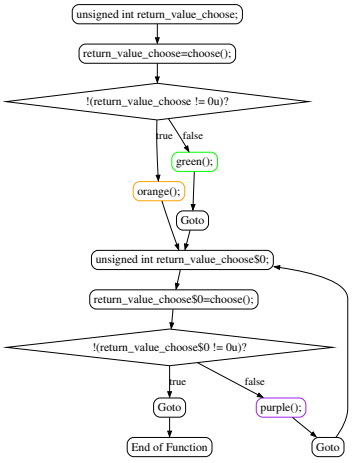


Figure 3. Control flow merges can create multiple loop entry edges

In the parse tree representation, the left example has two loops while the right has only one. However in the CFG representation both have one natural loop.

3 Common Mistakes About Loop Structure

3.1 Entering Loops

↯ Loops have one entry edge. If two or more control flow paths merge at the loop entry and there is no explicit merge node, it is possible to have multiple entry edges. Consider the following program and its CFG given in Figure 3:

```
if (choose())
  green();
else
  orange();

while (choose())
  purple();
```

↯ All entry edges go to the same location. C and some other languages allow writing loops with multiple entry points. Arguably an unintentional "feature" of the language, the labels used by `switch` statements are normal labels and can appear within other control flow structures. Duff [8, 13] used this to provide a manual version of loop unrolling for older compilers and hardware:

```
switch (n & 0x3) {
  do {
    case 0 :   dest[i++] = src[j++];
    case 1 :   dest[i++] = src[j++];
    case 2 :   dest[i++] = src[j++];
    case 3 :   dest[i++] = src[j++];
  } while (j < n);
}
```

Simon Tatham's implementation of coroutines uses a more advanced version of the same idea [22]. Studies suggest cycles with multiple entry points are rare [20].

↯ Multiple loop entry locations can be fixed by one unrolling. As entry locations only affect the first iteration, a tempting solution is to unroll the first iteration of any loop with multiple entry locations. In most cases this is a simple and reasonably efficient solution. However [6, 15] shows there are pathological cases which result in an exponential size increase.

↯ The first instruction must be an entry location. It is possible for no entry to exist to the obvious first instruction, and that the entry can be from inside a nested loop.

```
if (choose())       goto one;
else if (choose()) goto two;
else                goto three;

while (choose()) {
  while (choose()) {
          red();
  one:    orange();
  two:    yellow();
  three:  green();
  }
}
```

3.2 Inside Loops

↯ Loops have contents. In all representations, it is possible to create an empty loop. For example busy-wait loops for low latency inter-thread communication:

```
void busy_wait (void) {
  while (zero_to_unlock);
  return;
}
```

SV-COMP[1] and others [7] use empty loops as an idiom to terminate an analysis path.

```
loop : goto loop;
```

↯ The entry location is a test for exiting the loop. C and many other languages have a syntactic construct where this is not true; the `do/while` loop:

```
do {
  blue();
} while (choose());
```

It is tempting to think that marking `do/while` loops or unrolling the loop once will make this belief true. There are a number of less obvious cases where this does not hold. Depending on whether function calls, pointer validity checks, modifications of variables, etc. need separate instructions this may not start with a conditional exit:

```
while (f00(*(pointer++)) == value)
  yellow();
```

This is also a case where inlining can cause significant changes to the properties of the loop, see Section 3.4.

↯ End of a loop is an unconditional jump. Only the parse tree and list representations primarily have a concept of a last instruction. In the list representation, the last instruction must be a backwards jump for it to be a loop, but it can be conditional, for example in the `do/while` loop.

↯ There is a single backwards jump or back edge. Within the list representation, this depends on whether `continue` is implemented as a jump to the end or as a jump to the start:

```
while (choose()) {
  pink();
  if (choose()) { continue; }
  blue();
}
```

Some languages have control flow statements for "redo this loop iteration (without testing the loop condition)". Perl and Ruby have the `redo` keyword, useful for iterating over data structures or streams where there may not be a way of undoing the loop counter update. The simplest implementation of these is using additional back edges.

↯ Loops may have multiple back edges but they all go to the same place. In nested loops the back edge of the inner loop will appear as a back edge into the middle of the outer loop.

```
while (choose()) {
  while (choose()) {
    yellow();
  }
}
```

3.3 Exiting Loops

↯ Loops have an exit. Infinite loops can and do happen in correct code. Event driven systems and some control systems will often have one main control loop without an exit.

```
do {
  handle_request();
} while (1);
```

↯ Loops have a single exit edge. The `break` gives a simple way to have multiple exit edges to the same location:

```
while (choose()) {
  orange();
  if (choose()) { break; }
  pink();
}
```

↯ All `break` statements go to the same location. A subtle consequence of the example in Figure 2 is that the CFG loop exits as soon as a `break` statement is unavoidable, not at the `break`. The instructions between the `if` and `break` statements are the actual exit locations. The following loop

has three exit locations, one for each `break` and one for the loop condition:

```
for (int i = 0; i < firewall->rule_count; ++i) {
 if (firewall->rule[i].matches(packet)) {
  if (firewall->rule[i].type == WHITELIST) {
   packet.status = ACCEPTED; // 1st exit location
   break;
  }
  else if (firewall->rule[i].type == BLACKLIST) {
   packet.status = REJECTED; // 2nd exit location
   break;
  }
 }
}
// 3rd exit location
```

In effect this inserts code "after" the loop exit when using `break` but not when exiting via the loop condition. Python supports `else` statements attached to loops to resolve this asymmetry. Another version of this problem occurs when the loop contains multiple variable declaration scopes and the source or intermediate language requires actions at the end of scope (marking dead variables, C++ destructors, etc.). A `break` may need to perform end of scope actions while the loop condition does not as it is before the start of the scope.

***↯ Loops can only exit using the loop condition or* `break`.** A `return` statement provides a third kind of exit location that is not the same as the loop condition or `break`:

```
while (choose()) {
  orange();
  if (choose()) { return; }
  pink();
}
```

***↯ Loops can only exit using the loop condition,* `break` *or* `return`.** Many languages allow `break` statements to specify the depth of nested loops they are exiting. Kosaraju [12] shows this to be vital to prove a version of the structured programming theorem that does not require additional variables. Thus it can be argued to be a fundamental control flow statement. C does not have break to label so this is regarded as one of the legitimate uses of `goto`, for example:

```
for(unsigned int i = 0; i < WIDTH; ++i) {
  for(unsigned int j = 0; j < HEIGHT; ++j) {
    for(unsigned int k = 0; k < DEPTH; ++k) {
      if (next_cell(i, j, k)) { continue; }
      else if (next_column(i, j, k)) { break; }
      else if (next_row(i, j, k)) { goto nextRow;}
      else process_cell(i, j, k);
      }
    }
nextRow:;
}
```

Another legitimate use of `goto` is the creation of multiple return paths with different resource deallocation and error handling strategies. This is a common pattern in the Linux kernel and can give additional exit locations. The following example provides a degree of robustness against programming errors as unhandled branches and conditions will default to the error exit path:

```
struct subsys *s = kalloc(sizeof(struct subsys));
int err = subsys_init(s, parameters, policy);
if (err) { goto fail_subsys; }

for (unsigned int i = 0; i < s->hook_count; ++i) {
  err = subsys_register_hook(s, i);
  if (err) { goto fail_hook; }
}

if (subsytem_status(s) == FUNCTIONAL)
  { goto success; }

fail_hook:   subsys_unregister_hooks();
fail_subsys: kfree(s);
  klog("Failed to configure subsys %d", err);
  return false;

success:
  klog("Subsys %s configured with %d hooks",
       s->identifier, s->hook_count);
  return true;
```

Some languages have additional control flow statements which provide additional different exit locations for example exception handlers.

3.4 Simplification, Preprocessing and Optimisation

↯ Simplifications do not affect loop analysis. Semantic reasoning can alter the results of loop analysis. For example, a common pattern to force a macro to be used like a statement requires a `do`/`while` loop to "swallow" the semi-colon:

```
#define INIT_SUBSYSTEM(X) do { \
  bzero((X), sizeof(struct subsystem)) ; \
  load_system_config((X)); \
  register_subsystem((X)); \
} while (0)

INIT_SUBSYSTEM(networking);
```

Whether this is regarded as a loop depends on the kind and amount of simplification before the loop analysis.

↯ Simplification only affects control edges decisions. The previous example can be handled during the construction of the CFG by omitting edges that cannot be taken. Unfortunately not all simplifications are so straight-forward:

```
#define POINTER_RW(p) do { *(p) = *(p); } while(0)

do {
  POINTER_RW(p);
  do {
    some_code(p);
```

```
  } while (condition1());
  p = p->next;
} while (p != null);
```

If p is always an accessible address, then POINTER_RW can be removed such that there is one CFG loop rather than two. The converse effect is also possible; adding instructions or instrumentation at the start of loop bodies can increase the number of loops.

↯ Preprocessing can alter the number of loops but not create them from nothing. Depending on the language's requirements and where in the compilation process loop analysis is performed, this may not be true. If the language requires that tail recursive functions are rewritten it is possible for loops to "appear" in an acyclic (but recursive) function.

↯ Inlining is harmless. Inlining allows simulation of context-aware analysis to potentially simplify the handling of small utility functions, and "syntactic sugar". On inlining, `return` statements act as a lightly restricted form of forward `goto` statements. The `return` is replaced with a jump to the calling context. This can give a loop exits which are a significant distance from the loop, may or may not merge with flow control from the loop and are not easily recognised as a special case in the way normal `return` statements are. If the tool performs inlining then a lot of developer intuitions, "This works as long as the input programs do not have `goto`" no longer hold. Many uses of `goto` in this paper can be simulated with inlined `return` statements[1].

↯ Jump threading is harmless. If a jump instruction J targets a second, unconditional jump K, then J can be redirected to directly jump to the target of K. In most cases this simplifies the resultant CFG but it can interact in unexpected ways with other preprocessing steps and undo other simplifications. For example, if `continue` is implemented as a jump to the end of the loop body (see Section 3.2) then jump threading can covert these to additional back edges. If the instruction after a function call is a jump then inlining and then jump threading can create a `goto` from anywhere in the callee to anywhere in the caller.

4 Recommendations

Faced with some of the oddities and edge cases described in this paper, a reasonable reaction is to ask how common they are. In our experience, loop edge cases are more common than anticipated in user-supplied code. Sources include code compiled from languages with non-C control flow constructs like Python's `for-else`, Ada's `loop` or nested `break` instructions, autogenerated code from DSLs (like Simulink), combinations of preprocessing steps, such as inline, loop acceleration, unrolling then simplification, legacy code, hand optimised high-performance code, decompiled optimised code, particularly loop unrolling, splitting, fusion and i-cache layout optimisations and decompiled obfuscated code.

As a tool becomes prosperous, the chances of encountering one or more edge cases rise dramatically. Thus we recommend the following steps:

1. Design for the most general case. For example, if you are using the CFG definitions of a loop, the API should return the set of entry locations and the (possibly empty) set of exit locations. It is fine to have more specific calls such as getting *the* entry location or *the* exit condition but these should have at least a run-time check that the loop is in the required form.
2. Make explicit which cases are not handled. It is reasonable to initially not handle irreducible loops or to handle them with a naïve exponential algorithm. However this should be explicitly checked at run time and documented to the user. Extending support by adding extra cases is strongly preferable to having a complete algorithm for some loops and adding rewrite steps before and after to expand the set of supported loops.
3. If your system makes use of more than one representation then handling the translation between them needs to be part of the design. For example, a system that generates source/parse tree loop invariants using a CFG-based technique will need to track the correspondence between the two definitions of loop. Annotating parse tree concepts such as the loop body and loop conditions into the CFG is one approach but care needs to be taken to preserve them (see Section 3.4).
4. Loop analysis code needs careful and systematic testing with awareness of the possible edge cases. The examples given in this paper are available as a test suite [5]. Each program is a minimal test for the specific issue, with a single input that controls the path taken through the program, providing a single output that records the branches taken. This can be used for back-to-back testing with compilers, interpreters, source-to-source translation, or dynamic analysis tools. If the input/output relation changes, the transformation is buggy and is either inserting or omitting edges. To test static analysis tools, the programs also contain assertions that are only true for valid paths through the program. If an assertion fails, it indicates a bug in the static analysis tool. We provide a usable set-up to test the examples supplied with CBMC.

[1] `return`s should be thought of as domesticated `goto`s but domesticated in the same way that cats are domesticated.

Acknowledgments

Research reported in this publication was supported by an Amazon Research Award, Fall 2022 CFP. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of Amazon.

The examples in this paper have been collated over a long period, from bug reports, folklore, edge cases found during development, etc. The authors wish to thank everyone who has had input. Where possible, citations are provided. In particular, Martin Brain would like to thank Peter Schrammel for his development model, Holly Nyx for editorial support, and John Galea for being the person saying "But Martin, that doesn't work because...".

References

[1] 2023. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org/

[2] Alfred Vaino Aho, Monica Sin-Ling Lam, Ravi Sethi, and Jeffrey David Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.

[3] Frances Elizabeth Allen. 1970. Control Flow Analysis. 5, 7 (1970). https://doi.org/10.1145/390013.808479

[4] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety Verification and Refutation by k-Invariants and k-Induction. In *Static Analysis*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 145–161.

[5] Martin Brain and Mahdi Malkawi. 2024. [artifact] Misconceptions About Loops in C. https://doi.org/10.5281/zenodo.11113582

[6] Larry Carter, Jeanne Ferrante, and Clark Thomborson. 2003. Folklore Confirmed: Reducible Flow Graphs Are Exponentially Larger. *SIGPLAN Not.* 38, 1 (jan 2003), 106–114. https://doi.org/10.1145/640128.604141

[7] Byron Cook, Björn Döbel, Daniel Kroening, Norbert Manthey, Martin Pohlack, Elizabeth Polgreen, Michael Tautschnig, and Pawel Wieczorkiewicz. 2020. Using model checking tools to triage the severity of security bugs in the Xen hypervisor. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 185–193. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_26

[8] Tom Duff. 1988. Re: Explanation, please! Usenet. http://doc.cat-v.org/bell_labs/duffs_device

[9] Paul Havlak. 1997. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.* 19, 4 (jul 1997), 557–567. https://doi.org/10.1145/262004.262005

[10] Matthew Sterling Hecht and Jeffrey David Ullman. 1974. Characterizations of Reducible Flow Graphs. *J. ACM* 21, 3 (jul 1974), 367–375. https://doi.org/10.1145/321832.321835

[11] Brian Wilson Kernighan and Dennis MacAlistair Ritchie. 1978. *The C Programming Language*. Bell Telephone Laboratories Incorporated.

[12] Sambasiva Rao Kosaraju. 1974. Analysis of structured programs. *J. Comput. System Sci.* 9, 3 (1974), 232–255. https://doi.org/10.1016/S0022-0000(74)80043-7

[13] Chloé Lourseyre. 2021. Duff's device in 2021. https://belaycpp.com/2021/11/18/duffs-device-in-2021/

[14] Daniel Neville, Andrew Malton, Martin Brain, and Daniel Kroening. 2016. Towards automated bounded model checking of API implementations. *CEUR Workshop Proceedings* 1639, 31–42.

[15] Carl D. Offner. 2013. *Notes on graph algorithms used in optimizing compilers*. Technical Report. University of Massachusetts Boston. https://www.cs.umb.edu/~offner/files/flow_graph.pdf

[16] Florian Schanda and Martin Brain. 2012. Using Answer Set Programming in the Development of Verified Software. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 17)*, Agostino Dovier and Vítor Santos Costa (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 72–85. https://doi.org/10.4230/LIPIcs.ICLP.2012.72

[17] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. 2015. Successful Use of Incremental BMC in the Automotive Industry. In *Formal Methods for Industrial Critical Systems*, Manuel Núñez and Matthias Güdemann (Eds.). Springer International Publishing, Cham, 62–77.

[18] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. 2017. Incremental Bounded Model Checking for Embedded Software. *Form. Asp. Comput.* 29, 5 (sep 2017), 911–931. https://doi.org/10.1007/s00165-017-0419-1

[19] Kaitlyn Siu and Marcelo Badari. 2022. *What's A Loop : A Tree House Adventure*. Wayland.

[20] James Stanier and Des Watson. 2012. A Study of Irreducibility in C Programs. *Softw. Pract. Exper.* 42, 1 (jan 2012), 117–130. https://doi.org/10.1002/spe.1059

[21] Youcheng Sun, Martin Brain, Daniel Kroening, Andrew Hawthorn, Thomas Wilson, Florian Schanda, Francisco Javier Guzmán Jiménez, Simon Daniel, Chris Bryan, and Ian Broster. 2017. Functional Requirements-Based Automated Testing for Avionics. In *2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS)*. 170–173. https://doi.org/10.1109/ICECCS.2017.18

[22] Simon Tatham. 2000. Coroutines in C. https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

[23] Tomoya Yamaguchi, Martin Brain, Chirs Ryder, Yosikazu Imai, and Yoshiumi Kawamura. 2019. Application of Abstract Interpretation to the Automotive Electronic Control System. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 425–445.

Received 03-MAR-2024; accepted 2023-04-19; accepted 3 May 2024

Author Index