Universal Scalability in Declarative Program Analysis (with Choice-Based Combination Pruning)

ANASTASIOS ANTONIADIS*, University of Athens, Greece
ILIAS TSATIRIS*, Dedaub, Greece
NEVILLE GRECH, University of Malta, Malta and Dedaub, Malta
YANNIS SMARAGDAKIS, University of Athens, Greece and Dedaub, Greece

Datalog engines for fixpoint evaluation have brought great benefits to static program analysis over the past decades. A Datalog specification of an analysis allows a declarative, easy-to-maintain specification, without sacrificing performance, and indeed often achieving significant speedups compared to hand-coded algorithms.

However, these benefits come with a certain loss of control. Datalog evaluation is bottom-up, meaning that all inferences (from a set of initial facts) are performed and *all* their conclusions are outputs of the computation. In practice, virtually every program analysis expressed in Datalog becomes unscalable for some inputs, due to the worst-case blowup of computing all results, even when a partial answer would have been perfectly satisfactory.

In this work, we present a simple, uniform, and elegant solution to the problem, with great practical effectiveness and application to virtually any Datalog-based analysis. The approach consists of leveraging the *choice* construct, supported natively in modern Datalog engines like Soufflé. The choice construct allows the definition of functional dependencies in a relation and has been used in the past for expressing worklist algorithms. We show a near-universal construction that allows the choice construct to flexibly limit evaluation of predicates. The technique is applicable to practically any analysis architecture imaginable, since it adaptively prunes evaluation results when a (programmer-controlled) projection of a relation exceeds a desired cardinality.

We apply the technique to probably the largest, pre-existing Datalog analysis frameworks in existence: Doop (for Java bytecode) and the main client analyses from the Gigahorse framework (for Ethereum smart contracts). Without needing to understand the existing analysis logic and with minimal, local-only changes, the performance of each framework increases dramatically, by over 20x for the hardest inputs, with near-negligible sacrifice in completeness.

CCS Concepts: • **Theory of computation → Program analysis**; **Constraint and logic programming**.

Additional Key Words and Phrases: Static analysis, program analysis, logic programming, datalog, optimization

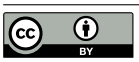
ACM Reference Format:
Anastasios Antoniadis, Ilias Tsatiris, Neville Grech, and Yannis Smaragdakis. 2025. Universal Scalability in Declarative Program Analysis (with Choice-Based Combination Pruning). *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 351 (October 2025), 28 pages. https://doi.org/10.1145/3763129

1 Introduction

Declarative static program analysis has received significant attention in the past two decades, with a wealth of research publications and open-source tools [6, 17, 19, 20, 22–24, 26, 28–30, 34,

*Both authors contributed equally to the paper

Authors' Contact Information: Anastasios Antoniadis, University of Athens, Athens, Greece, anantoni@di.uoa.gr; Ilias Tsatiris, Dedaub, Athens, Greece, ilias@dedaub.com; Neville Grech, University of Malta, Msida, Malta and Dedaub, San Gwann, Malta, me@nevillegrech.com; Yannis Smaragdakis, University of Athens, Athens, Greece and Dedaub, Athens, Greece, smaragd@di.uoa.gr.



ACM 2475-1421/2025/10-ART351
https://doi.org/10.1145/3763129

35, 39, 42, 48, 49, 55, 58]. The essence of declarative analysis is to express a program analysis algorithm as monotonic inference rules that get evaluated up to fixpoint. This style of analysis specification is an excellent fit for the highly-recursive nature of static analysis algorithms, as well as the interdependencies between the analysis of various distinct program features.

The Datalog language has emerged as the primary platform for declarative program analysis. Datalog is syntactically similar to Prolog but its evaluation semantics are strictly declarative. Ordering of rules or of clauses inside an inference rule does not affect the output of a computation. To maintain this property, Datalog evaluation (in contrast to Prolog evaluation, which is goal-directed/top-down) is *bottom-up*: it infers *all* results that follow from the initial input facts and the transitive application of inference rules, instead of merely attempting to find one result, or allowing any user control over the search space. In this way, the Datalog engine is free to decide the evaluation order of rules as well as the exact implementation of deriving the inference results for a single rule.

Declarative program analysis in Datalog has yielded elegant, concise analysis specifications that have helped with the invention of new algorithms (e.g., *type-sensitive* analysis [40] or *data-driven context tunneling* [17]). At the same time, the analysis enjoys great execution efficiency. For instance, when the Doop framework was introduced [6], in 2008, it outperformed a pre-existing manual implementation of fully equivalent analyses by a factor of 10x. The reason for this efficiency is primarily that Datalog conducts "set at a time" computation, evaluating rules by joining large tables—a computation that is very efficient in terms of cache locality, grouping for minimization of overheads, and inherent parallelism.

At the same time, the not-so-hidden weakness of Datalog-based program analysis has been its bottom-up evaluation, which starts from input facts and computes *all* possible inferences from these facts, recursively, until no more inferences can be made. This means that the analysis is very efficient in common cases, but completely unscalable for pathological inputs (which may have nothing truly "pathological", outside the context of the analysis itself). This occurs most commonly if the analysis input is very large, if the analysis fails to maintain precision (so that its—final or intermediate—results are large), or if it otherwise explodes in complexity for *part* of the analyzed program (typically because of a very large number of contexts, in the case of a context-sensitive analysis). This is a phenomenon commonly identified in the literature—e.g., publication [42] discusses the abrupt switch to unscalability in length, but virtually any declarative analysis paper mentions unscalability in some form.

Much of the program analysis literature has been about addressing analysis scalability issues, by producing better algorithms. However, the state of the art remains unscalable for precise context-sensitive analysis of large inputs (e.g., web applications [2]). Even worse, the boundary of unscalability can remain entirely unpredictable.

A reliable way to make static analysis scalable is via *pruning*: once intermediate sets in a computation reach a certain size, avoid enlarging them further. For recent instances, the "saturation" work of Wimmer et al. [56] enforces a maximum threshold for the points-to sets of variables, in the context of GraalVM's Native Image analysis, which produces standalone binaries for Java applications. This allows analyzing "*large Java applications with hundreds of thousands of methods in less than two minutes.*" [56]. The "indirection-bounded" work of Chakraborty et al. [7] bounds the number of indirections covered by the analysis, achieving 2X speedups "*with little sacrifice in recall*" [7].

However, pruning techniques appear incompatible with the bottom-up nature of Datalog evaluation, i.e., with an engine optimized to produce all true results in parallel. No past work has managed to employ pruning-based approaches fruitfully in the context of a Datalog-based program analysis. Nor has it been clear that the combination of pruning with bottom-up evaluation could be

done efficiently, even if a high-throughput Datalog engine were to be completely re-engineered to support pruning. (This concern is far from theoretical. Other "short-circuiting" techniques have introduced significant overhead in the context of Datalog evaluation, to the extent that they make analyses much slower instead of faster. Our related work in Section 6 discusses further.)

In this work, we address the need for Datalog analysis scalability via pruning, introducing *Choice-Bound*: a simple, universal technique, achieving excellent results—often outperforming past analysis innovations by more than an order of magnitude, for the most challenging benchmarks. Interestingly, the technique not only proves the compatibility of pruning with bottom-up evaluation, but does it without requiring any re-engineering, leveraging facilities already existing in the foremost Datalog engine, Soufflé.

There are three significant aspects of the *Choice-Bound* technique: a) it is simple; b) it is very efficient to implement and leads to great efficiency/scalability gains; c) it introduces a large design space that enables the analysis designer to perform intelligent choices.

The main idea of *Choice-Bound* is straightforward. We first observe that Datalog engines can efficiently support functional dependencies: combinations of certain variables (i.e., columns, when the relation is viewed as a table) can only occur once. (This is a standard feature from the data processing days of Datalog, equivalent to declaring primary keys in database tables.) One semantics for functional dependencies is *non-deterministic choice*: keep any one of the value combinations for the variables/columns that are outside the function domain/key. The Soufflé Datalog engine (generally considered to be the most mature, performant, and widely used) supports this via the *choice-domain* feature [15].

Then, if it is possible to efficiently support non-deterministic functional dependencies inside a Datalog engine, we can hijack the same mechanism to implement *multiplicity* dependencies: all that is needed is an extension of the relation with a shadow variable and a simple hash function. In this way, instead of "this column combination can only occur once" (a functional/key dependency), *Choice-Bound* expresses "this column combination can occur at most N times" (a multiplicity dependency). This leads to an excellent way to bound and control analysis complexity, both in terms of internal metrics (e.g., "number of contexts") and in terms of final results (e.g., "values per variable"). This adaptation is highly powerful and opens up a wide design space that the analysis designer can employ for very efficient tuning.

Bounding the number of combinations of key variables is a standard pruning approach and directly addresses the worst-case behavior of the analysis. *Choice-Bound* effectively solves the problem of scalability of Datalog-based analyses, at the expense of introducing some analysis incompleteness and non-determinism.

In overview, the contributions of this work are as follows:

- We introduce *Choice-Bound*: a way to easily make *any* pre-existing Datalog-based program analysis implementation significantly more scalable, using pruning of the analysis evaluation.
- We identify the design space behind *Choice-Bound* and introduce a vocabulary for expressing the analysis designer's choices.
- We, thus, establish that pruning and bottom-up evaluation can be combined with high efficiency. In fact, the declarative specification of analyses becomes an excellent domain for experimenting with pruning approaches, with easy machinery for tuning the pruning parameters. Pruning via *Choice-Bound* is not only feasible but also elegant.
- We show the effectiveness of the technique in major pre-existing analysis frameworks and the toughest benchmarks tackled in past literature, achieving speedups of over an order of magnitude (and often two or more). We also show several metrics of completeness (from coverage to ability to detect bugs) and establish that the tradeoff of *Choice-Bound* is excellent

in practice.

2 Background

We begin with some background on program analysis in Datalog and the choice-domain construct.

2.1 Program Analysis in Datalog

Declarative static program analysis (in the Datalog language or its variants, such as Flix [28, 30] or Inca [49]) has seen great progress in recent years. The reason is the straightforward specification of highly complex algorithms and the ability to execute efficiently.

The essence of the Datalog language is its ability to define recursive relations. Relations (or equivalently *predicates*) are the main Datalog data type. Computation consists of inferring the contents of all relations from a set of input relations. For instance, consider the domain of *pointer analysis*: computing which (object) values flow to which pointer, throughout the program. If the program-to-analyze is in, say, Java, it is easy to represent the relevant statements of the Java program as relations, typically stored as database tables. Consider two such relations, `AssignHeapAllocation(var, obj)` and `Assign(to, from)`. (We follow the convention of capitalizing the first letter of relation names, while writing variable names in lower case.) The former relation represents all occurrences in the program of an instruction "`a = new A();`" (in Java syntax) where a heap object is allocated and assigned to a variable. That is, a pre-processing step takes a Java program (most likely in intermediate, bytecode, form) as input and produces the relation contents. A static abstraction of the heap object is captured in variable `obj`—it can be concretely represented as, e.g., a fully qualified class name and the allocation's bytecode instruction index. Similarly, relation `Assign` contains an entry for each assignment between two program (reference) variables ("`p = q;`") in the program. The mapping between the input program and the input relations is straightforward and purely syntactic. After this step, a simple, *context-insensitive* pointer analysis can be expressed in Datalog as a transitive closure computation, as shown in the first five lines of Figure 1.

```
VarPointsTo(var, obj) :-
  AssignHeapAllocation(var, obj).

VarPointsTo(to, obj) :-
  Assign(to, from),
  VarPointsTo(from, obj).

VarPointsTo(to, obj) :-
  LoadField(base, fieldname, to),
  VarPointsTo(base, baseobj),
  InstanceFieldPointsTo(baseobj, fieldname, obj).

InstanceFieldPointsTo(baseobj, fieldname, obj) :-
  StoreField(base, fieldname, from),
  VarPointsTo(base, baseobj),
  VarPointsTo(from, obj).
```

Fig. 1. Simple specification of value-flow (points-to) analysis, with *field sensitivity*.

The Datalog program consists of a series of *rules* that are used to establish facts about derived relations (such as `VarPointsTo`, which is the points-to relation, i.e., it links every program variable, `var`, with every heap object abstraction, `obj`, it can point to) based on a conjunction of previously established facts. We use a Prolog-style left arrow symbol (`:-`) to separate the inferred fact (the

head) from the previously established facts (the *body*). For instance, the second rule in Figure 1 says that if (for some values of `from`, `to`, and `obj`) `Assign(to,from)` and `VarPointsTo(obj,from)` are both true, then it can be inferred that `VarPointsTo(obj,to)` is true. Note the base case of the computation (first rule), as well as the (linear) recursion in the definition of `VarPointsTo` (second rule).

To see the benefits of a declarative specification, consider the rest of Figure 1, which refines the rudimentary points-to analysis of the first two rules, without needing to change them. The refinement adds to our analysis *field sensitivity*: heap objects can be stored to and loaded from instance fields and the analysis keeps track of such actions. (This example ignores other language features such as method calls—i.e., we assume the analyzed program is just a single `main` function.) Two new input relations are derived from the code of a Java program: `LoadField(base, fieldname, to)` and `StoreField(from, base, fieldname)`. The former tracks a load from the object referenced by variable `base` in the field identified by `fieldname`. If, for instance, the Java program contains a statement "`x = v.fld;`", then `LoadField` contains an entry with the value of `base` being (a unique identifier of) Java variable "`v`", `fieldname` equal to field "`fld`", and `to` corresponding to "`x`". `StoreField` tracks store actions in a similar manner: Every Java program statement "`v.fld = u;`" corresponds to an entry in `StoreField(from, base, fieldname)`, with `v` represented by logical variable `base`, `u` represented by `from`, and an identifier for field `fld` captured by `fieldname`.

The bottom two rules in Figure 1 define and use a new relation, `InstanceFieldPointsTo`. This computes which heap object (`baseobj`) can point to which other (`obj`) through a given field (`fieldname`). The simple definition hides a lot of conceptual complexity: `InstanceFieldPointsTo` is defined by appeal to `VarPointsTo` (in two different ways in the same rule), which is, in turn, defined in terms of `InstanceFieldPointsTo`, and in terms of itself. The control flow of the analysis is now quite complex (employing non-linear recursion), but its specification remains simple. The result is a realistic Andersen-style points-to analysis [1], in compact form.

The striking aspect of the approach is that a) it generalizes to very complex analyses, both in breadth (i.e., covering all language features) and in depth (i.e., having a much more precise model); b) it maintains the simple analysis as its core, so that understanding Figure 1 conveys intuition about analyses with thousands of rules; c) the Datalog code can be executed highly efficiently.

2.2 Soufflé's Choice Construct

The *Choice-Bound* approach leverages the ability to have functional constraints on a relation, with non-deterministic choice of a representative. This is a feature already implemented highly efficiently in the foremost (in terms of maturity, performance, and adoption) current Datalog engine, Soufflé, as the `choice-domain` operator.

Extending Datalog with non-deterministic choice—the ability to choose an arbitrary item from a set—has a long history in the database literature[9, 13, 15, 31]. While there have been various mechanisms proposed to achieve this goal, Soufflé's variant—the choice-domain (or just "*choice*") construct—is based on enforcing functional constraints on a relation. The important features of the construct are, in the words of its authors [15], *(1) the simplicity of its semantics, (2) its ease of implementation, and (3) its efficiency in contrast to having no choice construct in the language.*

To see how the choice-domain construct works, consider the following program:

```
.decl r(x: number, y: number) choice-domain (x)
.output r

r(1,1).
r(1,2).
r(1,3).
r(2,4).
```

```
r(2,5).
r(3,6).
```

Let us first focus on the relation declaration:

```
.decl r(x: number, y: number) choice-domain (x)
```

The start of the declaration is quite standard; we are declaring a relation `r(x,y)` where both `x` and `y` are numbers. We then impose a functional constraint on the relation using the `choice-domain` operator. Essentially, we are enforcing that for every `x` there can only be one `y`. This is very similar to defining a primary key in relational databases. With this in mind, and assuming the inferences are processed in the order of declaration, it is easy to see that the output of the program will be:

```
r(1,1) r(2,4) r(3,6)
```

The rest of the facts get discarded due to the functional constraint. In general, whenever a new inference is made either directly by a fact or as a result of a rule inference, it is only added to the relation only if there is no existing tuple in the relation for the columns listed in the choice-domain declaration.

The choice-domain construct does not add theoretical expressiveness to the language (which is Turing-complete anyway), but expressing the same constraint in standard Datalog (with stratified aggregation/negation) would have been extremely inefficient (in addition to very cumbersome and error-prone). It would require computing the unconstrained relation, ordering its tuples through an arbitrary ordering, and iterating over the ordered tuples to only pick one per functional domain. In experiments with just a single application of choice-domain to compute a spanning tree, the speedup is typically over 5 orders of magnitude [15].

Thus, in practice, having language support for non-deterministic choice boosts the expressive power of the language and allows for more natural and performant implementations of algorithms that rely on a choice mechanism. The explicit motivation for the construct [15] has been worklist algorithms, where picking a representative (e.g., when forming a spanning tree) is an essential part of the algorithmic logic. Such algorithms arise in program analysis (e.g., in computing control-flow graphs or strongly-connected components), a domain of prominent use of Datalog.

The construct can be used for applications far exceeding the original intent, as we shall show.

3 *Choice-Bound*

We next present the *Choice-Bound* technique at a high-level, discussing its value proposition.

3.1 Outline

The essence of *Choice-Bound* is to lift the choice construct from expressing *functional dependencies* to expressing *multiplicity dependencies*, and creating an expressive parameterization space using hash functions. This compact wording may appear cryptic at first glance, but its essence is simple.

Consider a relation (a.k.a. a *predicate*) `R(x,y,z)`. Applying the choice construct on `x`, `y` means that the final contents of `R` will contain a single `z` value for each distinct combination of `x`, `y`. Essentially, `R` is declared to be a function from its first two fields to the last. If, without the choice construct, many combinations of `x` and `y` were to appear in the final contents of `R`, there is no guarantee as to which will be kept: the implementation can make an arbitrary choice, and it is up to the programmer to ensure that any such choice yields an acceptable output.

The goal of *Choice-Bound* is to relax this constraint from "a combination of `x`, `y` can only appear once" to "a combination of `x`, `y` can only appear up to N times." Keeping with the spirit of the choice construct, if more than N combinations were to arise (without the choice construct), there

is no guarantee as to which N combinations will be kept. The choice will be arbitrary, based on the evaluation order that the implementation chooses.

This relaxation is achieved by adding an extra field to the relation that is choice-bound, denoting a unique identifier of the "choice" that needs to be made. If these identifiers are then computed from a bounded domain of size N, we have the desired guarantee. Specifically, in our example:

- relation `R(x,y,z)` becomes `R(x,y,z,i)` with choice-domain (in the standard "functional" sense) `x,y,i`.
- Field `i` can then be computed via a hash function over the values of `z`, projected modulo N.
- In this way, `R(x,y,z,i)` is computed to have a single instance of each unique combination of `x,y,i`, i.e., up to N different `z` values for each combination of `x,y` values.

This seemingly simple approach yields tremendous power (discussed extensively in Section 4.1), by appropriately selecting the choice-bound fields, the bound parameter N itself, but also the domain of the hash function. This permits elegant exploration of a large design space of pruning parameters. In effect, a declarative static analysis also offers a declarative, high-level way of experimenting with pruning choices that are occasionally far from straightforward.

3.2 Discussion of Impact

Before we delve into the technical specifics of *Choice-Bound*, it is worth understanding its cost-benefit proposition, in terms of performance vs. completeness.

In substance, *Choice-Bound* enables an analysis designer to:

- express their code declaratively, as before, with virtually no changes to the rules;
- tune performance by adding extra constraints of the form "relation `R` shall never have more than N instances with the same values for fields `x,y`". If more results are going to be produced during rule evaluation, an arbitrary choice of which results are kept (typically the ones derived earlier) is made.

Thus, the value of *Choice-Bound* is that it can bound the cost of computation, prevent worst-case blowup of analysis time, as well as prune analysis paths that are unlikely to be fruitful (e.g., cover the same code element or combination of conditions, when they have already been covered N times).

If the underlying analysis already scales well, *Choice-Bound* has little to offer. However, even the most well-tuned, practical analyses can easily become unscalable, for a variety of reasons. These reasons can include larger inputs [2], or corner cases where precision is not preserved, leading to unscalability. (Even the simplest points-to analysis, for instance, is fundamentally an $O(n^3)$ computation [45]. It is not rare for a small part of the program to trigger such worst-case behavior, even leaving the rest of the program's analysis unaffected. However, if left unbounded, this worst-case behavior can easily dominate analysis time.)

In practice, bounding computation costs allows *Choice-Bound* to gain orders-of-magnitude increases in efficiency and scalability. The apparent trade-off is to sacrifice completeness and determinism.

This is an excellent trade-off in practice: realistic analyses have already made design decisions to sacrifice completeness and determinism, in order to get much more minor benefits than those offered by *Choice-Bound*. What *Choice-Bound* does is empower the analysis designer to find the sweet spot in this trade-off.

The scalability problem with most large-scale Datalog-based analyses stems from their bottom-up nature: the analysis computes *all* results, following from the inference rules. However, in practice it is commonly the case that not all results are necessary and that *which specific* results end up being computed matters little.

Consider a points-to analysis, such as the ones supported in the Doop framework—probably the largest single artifact of Datalog analysis code in existence, with over 45KLoC of Datalog code (in over 5,000 rules). The analyses produce points-to/value-flow inferences, together with a call-graph. The main final results of the analysis (consumable by other clients, human eyes or programmatic) are:

- The points-to information itself, i.e., the values that a given variable can take.
- The call-graph/reachability information for program functions.
- Information that results in bug warnings, such as taint information [12].

All of these outputs already tolerate incompleteness and non-determinism. Notably, whole-program static analysis (which is the kind that can become unscalable) does not typically establish the *absence* of some values from a value set, only estimates the *presence* of values. For instance, although the analysis may compute virtual calls with a unique target and casts that can never fail, this is done as a metric and not because this information is actually actionable for optimization: complex language features, such as reflection and dynamic loading [25], make this information incomplete, i.e., unreliable for automatic application. In general, missing some points-to/value-flow analysis results is an inevitable fact of life that analysis designers are already comfortable with. The vast majority of the published literature (effectively all realistic whole-program analysis algorithms) consists of unsound algorithms (i.e., incomplete with respect to producing all values that arise during run-time) [2, 6, 12, 17, 19, 20, 22–24, 26, 29, 30, 34, 35, 39, 40, 42, 48, 49, 55, 58]. Such analyses are mainly used for bug detection and security warnings (e.g., for information-flow vulnerabilities), which can tolerate incompleteness because the results are interpreted by humans. Secondarily the analyses are used in program understanding and programming assistance (e.g., inside an IDE, for auto-complete, refactorings, editor suggestions).

Similar to incompleteness, non-determinism is also expected. The Doop output is non-deterministic, due to multiple factors, such as multi-threaded fact generation, multi-threaded core analysis, arbitrary choice of representatives (e.g., for string objects or reflection [11]), and much more. To quantify just one of these sources of non-determinism: a quick text search over the Doop code reveals over 680 uses of the Soufflé Datalog "`ord`" operator, which returns an "ordinal", i.e., a unique identifier (typically an implementation-internal pointer address or index) for an entity. Each such use is a point of non-determinism in the analysis logic: re-running the analysis can make `ord` return a different value. The operator is typically employed in order to come up with *some* ordering of entities (e.g., to form spanning trees by selecting a representative from an equivalence class, or to exhaustively iterate over all entities of a certain kind).

Therefore, in principle, *Choice-Bound* sacrifices nothing that analysis designers have not already decided to sacrifice. The question remaining is one of the *extent* of the sacrifice, which is an engineering trade-off question, answered experimentally. As we shall see in Section 5, analysis incompleteness is very minor, whereas the performance benefits are dramatic. This is also largely due to the expressive richness of *Choice-Bound*, discussed next.

4 *Choice-Bound* Design Space and Applicability

Choice-Bound admits a large design space for parameterizing an analysis. We discuss the options, as well as practical implementation, illustrated with realistic examples.

4.1 Design Space and Vocabulary

Choice-Bound opens up a large design space for parameterization of a Datalog computation. This design space permits bounding, e.g., the internal complexity of an analysis, the final observed

values, key intermediate concepts, etc. This expressiveness permits precise tuning of a bounded computation.

To encode the design space available to the programmer, we introduce a vocabulary for describing choice-domain decisions under the *Choice-Bound* technique. The vocabulary admits three parameters:

- The *bound* variables, i.e., the dimensions/fields of a predicate whose combinations will be limited;
- the *limit*, i.e., the numeric bound;
- the *counting* variables, i.e., the dimensions/fields of the predicate that participate in the hash function, getting mapped to a unique identifier. (In all examples so far, we have set the counting fields to be all the fields that are not bound, but this is not necessary, as we shall soon discuss.)

We use the notation `Relation`::{`bound vars`}$_{limit}$ [[`counting vars`]] to capture this design space.

For illustration, let us consider predicate `VarPointsTo`, the main relation of a context-sensitive points-to analysis. This relation appears in numerous practical frameworks and research publications [6, 20, 23, 24, 39, 42] and forms the core concept behind most published pointer analysis algorithms.

The relation has 4 dimensions: `VarPointsTo(var,ctx,hobj,hctx)`. The intuitive meaning is that local variable `var`, under context `ctx`, may point to abstract *heap object* `hobj`, which was originally created under *heap allocation context* `hctx`. (The exact definition of contexts is orthogonal to our present discussion. It is only worth noting that it can vary tremendously, to yield different analysis algorithms.)

We consider several design choices over the same relation (and even the same underlying hash function). All of the design choices yield different performance/completeness profiles for the exact same core analysis. We illustrate, non-exhaustively, some of the options for predicate `VarPointsTo`, together with an intuitive explanation of the design intent behind each option. For concreteness, we use specific numbers for the *limit* parameter, but changing this limit is an obvious parameterization choice, which can be guided by observation and experimentation. Effectively, *Choice-Bound* offers a vocabulary for elegantly expressing the desired constraints, but insight and experimentation remain the responsibility of the programmer.

- `VarPointsTo`::{`var`}$_{541}$ [[`ctx,hobj,hctx`]]: This is a straightforward "cost-conscious" design choice. It limits the total output tuples/entries per local variable to at most 541 and every combination of context, heap object, and heap allocation context counts as a different tuple against this limit. Effectively, such a design choice bounds the *cost* of the analysis and not its externally observable outputs: each variable can contribute at most 541 entries to the final relation, without consideration regarding the meaning of these entries. For instance, all entries could have the variable possibly pointing to the same abstract value.
- `VarPointsTo`::{`var,ctx`}$_{101}$ [[`hobj,hctx`]]: This is a balanced design choice that mixes externally observable quantities (variables and heap objects) with internal elements that add precision at the expense of extra computation cost (contexts). The design choice limits every context-qualified variable to appearing at most 101 times in the analysis output. These 101 occurrences are identified by hashing both the heap object and the heap allocation context. It is perfectly possible for all 101 combinations of a context-qualified variable to point to the same heap *object* `hobj` but with different heap allocation contexts, `hctx`. This is notably different from the next design option.
- `VarPointsTo`::{`var,ctx`}$_{67}$ [[`hobj`]]: This option maintains the spirit of the preceding one, but with a restrictive twist. Only the heap *object* participates in the unique identifier of the

up-to-67 allowed combinations for each context-qualified variable (`var,ctx`). This means that if the same heap object under two different heap allocation contexts, `hctx`, is computed to reach the same context-qualified local variable, only one of the two tuples will be kept. In effect, heap allocation contexts are treated as less valuable information than heap objects: we can have many heap objects appear for a context-qualified variable, but with only one heap allocation context each.

- `VarPointsTo`::$\{$`var,ctx,hctx`$\}_{31}$ [[`hobj`]]: This design option captures yet another tradeoff, where the limit is in the number of values finally computed under the full precision of the analysis, i.e., as much context information as can be kept. One can see such a choice as saying "if the analysis computes too many heap objects, even with the full precision of variable contexts and heap allocation contexts, then it is not fruitful to keep all of them—just keep 31".
- More choices are possible. These include `VarPointsTo`::$\{$`var,ctx,hobj`$\}_{13}$ [[`hctx`]] (where the bound is on the number of heap allocation contexts kept, for all other dimensions being identical); `VarPointsTo`::$\{$`var,hobj`$\}_{277}$ [[`ctx,hctx`]] (where there is a total limit in the number of appearances of a variable-value pair, which is an externally-observable quantity).

Empirically picking constants. The approach that we have used for deriving specific numerical bounds (e.g., 541, above) has been purely empirical.

An analysis writer starts by specifying the analysis with no concern of bounds or pruning. This is an optimization concern, to come later. Once the analysis is run on programs of representative size and complexity (together with the libraries, which often contain the program points with the largest volume of analysis results), the analysis writer should wonder: a) which relations are the largest and what result volumes (e.g., number of values per variable, number of contexts per abstract value or variable) are expected inside the large relations; and b) whether analyzed programs can lead to analysis scaling problems.

Then the analysis writer can set bounds over the columns of key relations such that completeness will be virtually unaffected in common cases but scalability will benefit in difficult cases. It is typically not hard to come up with these bounds, because they are not too sensitive. Minimal experimentation was required to produce the constants we use in our evaluation. Numbers that are 50% higher or lower will give rise to very similar results. Sensitivity experiments (presented later, in Section 5.4) show that making radical changes to the parameters (e.g., a 5x increase) still yields benefit, but at different points of the scalability-completeness tradeoff. Smaller changes nicely track this complex tradeoff curve.

4.2 Applications

We applied *Choice-Bound* to several pre-existing Datalog analyses (with minimal change to the analysis specification, as discussed in Section 4.3). We discuss next the design choices for each of these frameworks, using our vocabulary of the previous section.

4.2.1 Doop. As already mentioned, the Doop framework for points-to and taint analysis of Java+Android bytecode comprises over 5,000 rules, in over 45KLoC. Applying *Choice-Bound* consisted of bounding only the `VarPointsTo` relation. (`VarPointsTo(var,ctx,hobj,hctx)` is the main points-to relation, discussed in Section 4.1.)

The bound used for the 2-object-sensitive+heap (2objH) analysis of Doop (i.e., an object-sensitive [33] analysis with 2 elements of context for local variables and 1 for heap objects) is:

- `VarPointsTo`::$\{$`var,ctx`$\}_{101}$ [[`hobj,hctx`]].
  That is, each context-qualified variable is limited to pointing to 101 context-qualified abstract objects. This is the "balanced" choice discussed earlier.

Good results can be obtained with various other design choices. However, the above settings are indicative of the potential of the approach.

Notably, Doop is a framework that encompasses many tens of analysis algorithms, for different flavors of context sensitivity. The *Choice-Bound* bound will likely need to be adapted for different kinds of context sensitivity, since, for instance, it makes no sense to use a bound of 101 context-qualified objects for an analysis with shallower (or none), or different-flavor context. However, there are not many analyses that are heavy or useful enough to be worth defining bounds for (or if they do become useful enough, experimenting with an appropriate bound is minor overhead). The 2objH analysis is by far the most prominent analysis: it is highly-precise for real applications but has failed in the past to scale to anything but small ones.

4.2.2 Symbolic Value-Flow Analysis. Symbolic value-flow (*symvalic*) analysis [41] is a recent large-scale analysis framework for Ethereum VM smart contracts, built on top of the Gigahorse decompiler [10]. The symvalic analysis core and its clients comprise some-3,000 Datalog rules, over nearly 30KLoC. We applied *Choice-Bound* to two relations:

- `ExprFlowsToVar(complexity,expr,ctx,var)` is the relation responsible for the symvalic pre-analysis [41, Sec.4.2], which creates a universe of symbolic expressions. The meaning of the relation is "variable `var` under context `ctx` can refer to expression `expr`, which has the listed `complexity`." The complexity is used for only allowing a limited number of processing steps while defining the universe of expressions, since this analysis can create an unbounded number of expressions (e.g., all integers) even for a very small expression size.
  The *Choice-Bound* specification for this relation is:
  `ExprFlowsToVar::{var}`$_{1373}$ **[[**`complexity,expr,ctx`**]]**.
  Thus, in this case we have a pretty large bound but over all possible tuples that pertain to a single variable. Intuitively, the reason that this design choice is different from others is that the relation being bounded does not keep any real precision: it computes *all* possible expressions in the universe, up to a finite number of combinations of program operations. Therefore, it makes sense to merely bound the total weight of a single variable, i.e., to have a more cost-conscious bound, as opposed to a balanced one.
- `VarMayBe(var,expr,deps)` is the main relation of the symbolic value-flow analysis, indicating that a variable `var` may have as its value a concrete or symbolic expression, `expr`, and for this to happen several dependencies, `deps`, also need to hold. (The shape of "dependencies" is orthogonal to our discussion. Briefly, they are of the form "argument a of the enclosing function needs to have value v", "global variable g needs to have value o", etc. [41].)
  The *Choice-Bound* specification for this relation is:
  `VarMayBe::{var,deps}`$_{257}$ **[[**`expr`**]]**.
  That is, the bound is straightforward, limiting the possible values of a variable, under the full precision of the analysis (i.e., without limiting the dependencies kept) to at most 257.
  Note how these bounds are higher than the earlier bounds in the Doop framework analyses. This is easy to understand by considering that the Doop framework is meant for whole-program analysis of applications with many tens of thousands of classes (and many tens of MB in binary form), whereas symvalic analysis attempts a much more precise analysis but over programs (smart contracts) that are at most 24KB in binary form.

4.2.3 Other applications. We have additionally applied *Choice-Bound* to other analyses, not as prominent as the above, nor as mature and independently developed. For instance, one of the applications is to a symbolic execution engine, where the Datalog analysis leads to queries to an SMT engine. An excellent bound in this case is to the number of symbolic conditions that "cover"

the same program block. This leads to bounding the number of SMT invocations and not just Datalog analysis, yielding very substantial speedups.

The overall message is that the technique applies to virtually any demanding Datalog analysis and its expressiveness permits very powerful tuning in order to find sweet spots in the performance-completeness trade-off.

4.3 Implementation

Implementing *Choice-Bound* is simple, requiring only local changes to a Datalog program (modulo a global semi-automatic search-and-replace). As a result, we are able to apply *Choice-Bound* to virtually *any* Datalog program analysis, with limited understanding of its logic or internals, as long as we can reason about the parameter space in the terms defined in Section 4.1.

The specific implementation technique that we use consists of the following steps.

- Identifying the relation to bound and deciding on the choice of parameters, per Section 4.1. As a running example, consider relation `VarPointsTo` of the Doop framework, bounded as `VarPointsTo`::$\{$`var,ctx`$\}_{101}$ **[[**`hobj,hctx`**]]**.
- Creating a "bounded" version of the relation, in addition to the original:

```
.decl VarPointsTo_Bounded(var:Var, ctx: Context,
                          hobj: Value, hctx: HContext, n: number)
      choice-domain (ctx, var, n)
```

- Having the original relation obtain its contents from the bounded one:

```
VarPointsTo(var,ctx,hobj,hctx) :-
   VarPointsTo_Bounded(var,ctx,hobj,hctx,_).
```

- Defining the hashing that will produce ids for combinations of the *counting* variables, via a macro:

```
#define BOUND_VAR_POINTS_TO(var,ctx,hobj,hctx) \
   VarPointsTo_Bounded(var,ctx,hobj,hctx, (ord(hobj)*ord(hctx)) % 101)
```

  The macro uses the Soufflé `ord` operator to obtain an (arbitrary) internal identifier per object.
- Performing a global search-and-replace to substitute `BOUND_VAR_POINTS_TO` in the heads of rules that would otherwise produce `VarPointsTo` results. The *uses* of the original relation (i.e., in the bodies of rules and not in the head) remain unaffected.
  Furthermore, not even all rules with the bounded relation in their head need to be affected. If the analysis author wants to minimize the surface affected by the introduction of *Choice-Bound* (e.g., for practical concerns, such as minimizing the number of files that should be inspected before accepting the change), they can do so, provided they are aware of which rules are the most central (i.e., produce most results) in an analysis. Tedious rules that produce few results (e.g., initialization logic) can remain in their original form, since there is little need to limit their inferences. If we want the results of such rules to also participate in the bounding (so that we have a guarantee for the total tuples, as per the bounding policy), we can also include a feed-back rule:

```
BOUND_VAR_POINTS_TO(var,ctx,hobj,hctx) :-
   VarPointsTo(var,ctx,hobj,hctx).
```

  In this way, if tuples of the relation can be derived by non-bounded rules, they still participate in the bounding policy.

As an end result, *Choice-Bound* can be used to obtain dramatic scalability benefits within mere hours of adaptation work, agnostically, for nearly any Datalog-based program analysis.

5 Evaluation

We applied *Choice-Bound* to large pre-existing Datalog static analysis frameworks, as discussed in Section 4.2. We next evaluate the performance and completeness, relative to the unmodified code.

5.1 Doop

We evaluate *Choice-Bound* in the Doop framework, executed with the Soufflé Datalog engine. We perform the evaluation on a dataset that consists of all programs we could find in past literature for which Doop had trouble scaling. Most of these programs are well-known open-source web applications [2], whereas some more are from the DaCapo benchmark sets [5].

We evaluate with the default 2objH analysis configuration in Doop (a.k.a. "2-object-senitive+heap" or just "2-object-sensitive") using the Java 23 library. This is the Doop analysis that one would always *want to* run for precision, but generally cannot because it fails to scale to large programs.

We use a machine with two Intel(R) Xeon(R) Gold 6136 CPUs @ 3.00GHz (each with 12 cores x 2 hardware threads) and 640GB of RAM. The Soufflé version used is 2.1. Each benchmark is analyzed separately, with 12 threads, and we evaluate on two dimensions: completeness and speed.

Our benchmarks include:

- *alfresco*: An open-source content management system (CMS) and business process management software. It helps organizations manage documents, collaboration, records, and content online, enabling digital workflows and secure document handling. (GitHub, 142 stars, 82 forks.) Application classes: 9164. Total classes: 37163.
- *batik*: An open-source software toolkit for handling Scalable Vector Graphics (SVG) in Java that provides a suite of tools for viewing, generating, and manipulating SVG graphics. Developed by the Apache Software Foundation. (DaCapo benchmark) (GitHub, 214 stars, 143 forks.) Application classes: 2511. Total classes: 11993.
- *bitbucket-server*: A self-hosted Git repository management tool designed for collaborative software development. Bitbucket Server allows teams to host their Git repositories on their own servers (rather than in the cloud) and provides features like code review, branch permissions, and integration with other tools. Developed by Atlassian. Application classes: 581. Total classes: 29984.
- *bloat*: Despite being a mere software engineering project by a Ph.D. student, bloat is notorious for its complexity and the scalability challenges it poses. (DaCapo benchmark) Application classes: 360. Total classes: 4813.
- *dotCMS*: An open-source, hybrid content management system (CMS) designed for businesses that require a flexible and scalable platform to manage content across multiple channels. (GitHub, 864 stars, 468 forks.) Application classes: 5473. Total classes: 46027.
- *opencms*: An open-source content management system (CMS) offering a powerful solution for creating and managing websites, intranets, and online applications. Developed by Alkacon Software. (GitHub, 528 stars, 575 forks.) Application classes: 2143. Total classes: 16183.
- *pybbs*: PyBBS is an open-source, web-based bulletin board software developed in Java, built on top of the Spring Boot framework. PyBBS provides a platform for online discussions, allowing users to post messages, reply to threads, and communicate in a community setting. (GitHub, 1,900 stars, 716 forks.) Application classes: 172. Total classes: 24692.
- *shopizer*: Shopizer is an open-source Java-based e-commerce software designed to help businesses build online stores and manage product catalogs, orders, customers, and other

Benchmark	2objH Execution Time	*Choice-Bound* Execution Time
alfresco	86400 (Timeout)	1,576
batik	751	277
bitbucket-server	2,888	383
bloat	1,269	76
dotCMS	86400 (Timeout)	2,687
jython	86400 (Timeout)	1,511
opencms	6,414	567
pybbs	5,654	506
shopizer	8,871	555

Fig. 2. Execution times shown in seconds for the default 2objH analysis and *Choice-Bound*.

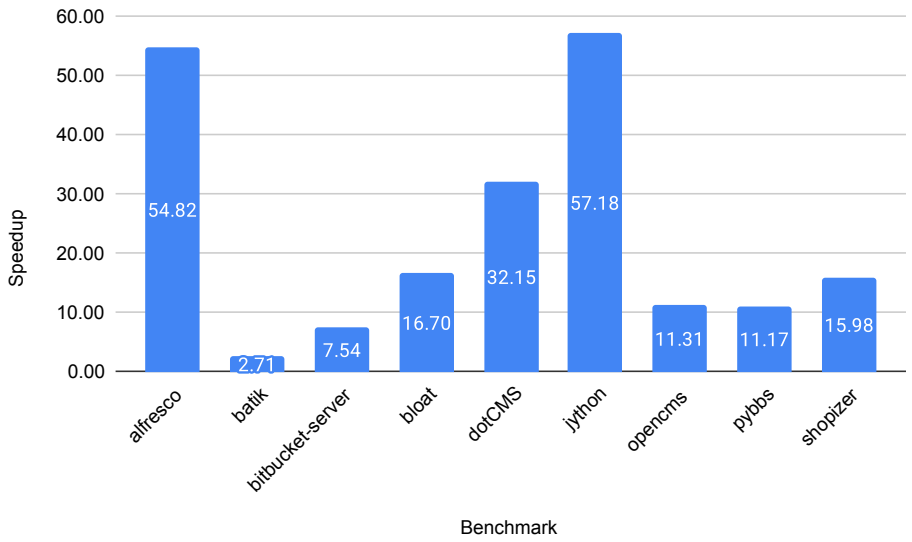


Fig. 3. Speedup per benchmark. The numbers shown are *multiplicative factors*. The three tallest bars are under-estimates, since the default analysis never terminated, in 24hrs.

e-commerce operations. (GitHub, 3,600 stars, 3,000 forks.) Application classes: 1151. Total classes: 35484.

- *jython*: An implementation of the Python programming language that runs on the Java platform. Essentially, Jython allows Python code to be executed within the Java environment, providing compatibility and interoperability between Java and Python. (DaCapo benchmark) (GitHub, 1,200 stars, 192 forks.) Application classes: 919. Total classes: 7303.

5.1.1 Performance. The most important aspect of *Choice-Bound* implementation is its impact on analysis performance. All 9 benchmarks terminated in under 1 hour and typically in under 30 minutes. This includes benchmarks that do not terminate in multiple days with the default analysis.

Figure 2 tabulates the execution times of the original analysis and the *Choice-Bound* analysis over the benchmark set. Figure 3 plots the speedup.

As can be seen, the speedup is dramatic. In the case of *alfresco*, *dotCMS*, and *jython*, if we consider the 24-hour timeout limit as a lower bound for their execution time, we see a >54.82x speedup for *alfresco*, a >32.15x speedup for *dotCMS* and a >57.18x speedup for *jython*. In more realistic terms,

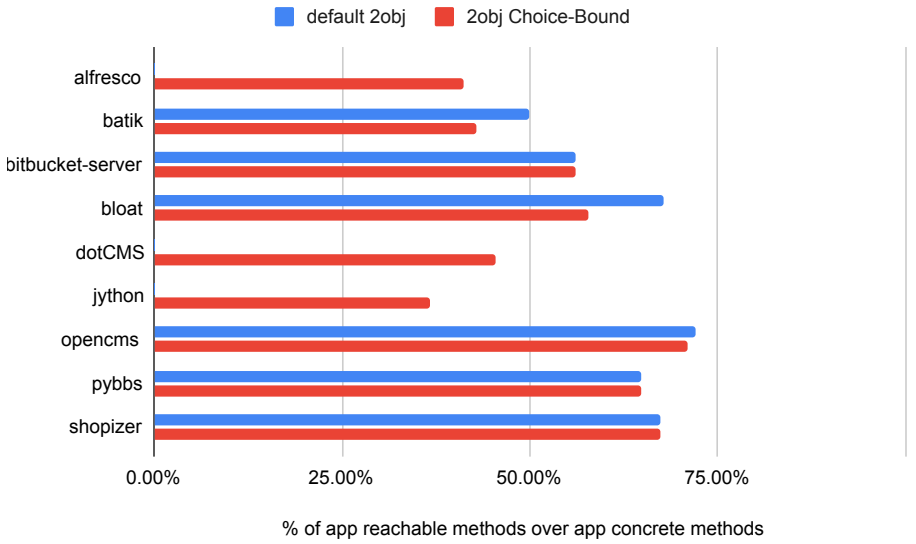


Fig. 4. App methods reachability for the default 2objH analysis and 2objH with *Choice-Bound*.

however, a terminating analysis is arguably immeasurably better than an analysis that times out. It is also worth noting that even a timeout of 48 hours is not enough for a 2-object sensitive analysis to terminate for either *alfresco*, *jython* or *dotCMS*.

The average speedup is 23.29x, ranging from a 2.71x speedup for *batik*, one of the smallest benchmarks, to 57.18x for *jython*.

5.1.2 Completeness. To evaluate the impact of *Choice-Bound* in analysis completeness, we first use the most straightforward coverage metric: the percentage of "app reachable methods" (i.e., methods deemed reachable in code that is part of the application or its immediate libraries and not in standard libraries) over all concrete methods in the application. The average reachability of app methods for the 2objH analysis is *62.99%* without taking into account *alfresco*, *dotCMS*, and *jython*, as these benchmarks timed out with a limit of 24 hours. Meanwhile, the 2objH *Choice-Bound* analysis yielded *59.96%* app method reachability over the same benchmarks.

Figure 4 plots the percentage of reachable methods in the application (plus the—non-system—libraries it is packaged with). In most cases, the loss of app method coverage is negligible, especially in the case of web applications.

Consider that all numbers are obtained with a uniform parameterization in the design space of *Choice-Bound*, namely $\texttt{VarPointsTo}::\{\texttt{var},\texttt{ctx}\}_{101}\ [\![\texttt{hobj},\texttt{hctx}]\!]$, as discussed in Section 4.2. If one adapts the parameters per benchmark or per rough benchmark characteristics (e.g., large web applications vs. DaCapo benchmarks), even better coverage and speedup should be possible. However, the goal of our evaluation is to demonstrate that one can get very good results with little-to-no tuning, for a globally uniform choice of parameters.

For a second, more targeted metric of analysis completeness, we consider one of the main reports of Doop: the number of reachable casts that may potentially fail (in application-level code). Since we have already quantified reachability (i.e., percentage of application methods that are covered by the analysis), we now focus on the subset of methods that are reachable for both *Choice-Bound* and the 2-object-sensitive analysis, for each benchmark. On the six benchmarks for which the 2-object-sensitive analysis terminates, we evaluate the loss of completeness as the number of application casts that may fail, where these casts are discovered by the 2-object-sensitive analysis

Benchmark	app may fail casts for 2objH∩*Choice-Bound* reachable methods	app *Choice-Bound* may fail casts	Completeness Loss
alfresco	N/A	3,593	0.00%
batik	747	734	1.74%
bitbucket-server	101	101	0.00%
bloat	705	677	3.97%
dotCMS	N/A	4,548	0.00%
jython	N/A	702	0.00%
opencms	1,642	1,621	1.28%
pybbs	55	55	0.00%
shopizer	268	263	1.87%

Fig. 5. App May Fail Casts and Completeness Loss.

but not by the *Choice-Bound* analysis.

Figure 5 tabulates the results of the comparison. As can be seen, the completeness loss identified in this case ranges between 0% for *pybbs* and 3.97% for *bloat*. Considering that a Doop analysis is unsound by default, the extra loss of identified *application casts that may potentiall fail* is almost negligible between the two analyses.

Overall, the experiment with Doop validates that *Choice-Bound* offers the analysis designer a powerful tool in tuning scalability vs. completeness. Although *Choice-Bound* will incur some completeness loss, this is typically minimal compared to the gains, with potential speedups of over an order of magnitude, along with multiple cases where a normally non-terminating analysis becomes a very realistic sub-hour task.

5.2 Symbolic Value-Flow Analysis

We applied *Choice-Bound* to the symbolic value-flow (*symvalic*) analysis framework [41] for Ethereum VM smart contracts. At first glance, this is a surprising domain of application, since the setting of smart contracts has fewer scalability problems (compared to large Java applications): smart contracts are small in size, only up to 24KB each in binary form. However, the symvalic analysis itself is very precise (path-sensitive, with symbolic evaluation), therefore it does occasionally fail to scale. In a sense, this analysis serves as validation of the universality of *Choice-Bound*, in diverse settings. The framework itself is quite large, with some-3,000 Datalog rules, so it certainly serves to validate the ease of application of the technique.

Notably, since the analysis is used to find several tens of real-world vulnerabilities, we can compare completeness a lot better, with in-depth metrics instead of just coverage. (Although coverage remains a reliable indicator.)

The input dataset consists of all smart contracts on the Ethereum blockchain that are:

- deployed between Ethereum block number 21,000,000 (produced on Oct. 19, 2024) and 21,090,000 (produced on Nov. 1, 2024);
- at least 15KB in size, to eliminate very small "proxy" contracts and other trivial contracts;
- deduplicated by "normalized bytecode", i.e., if two contracts have the same code modulo configuration constants, only one is kept.

This yields 754 unique contract codebases. The Gigahorse decompiler [10], underlying the analysis, fails to decompile 10 of the contracts (at least with its default settings). The remaining 744 comprise the input dataset.

	Choice-Bound	Default symvalic analysis
out of time (1500sec)	6	16
out of memory (50GB)	1	41
average analysis time	98.1sec	173.8sec

Fig. 6. Cumulative failures to analyze and analysis time for symvalic analysis, over 744 Ethereum smart contracts, with and without *Choice-Bound*.

	Choice-Bound over all contracts	*Choice-Bound* over common contracts	Default symvalic analysis
block coverage	92.9%	93.4%	93.5%
variable coverage	91.9%	92.5%	92.6%

Fig. 7. Analysis coverage metrics over all successfully-analyzed smart contracts and over contracts analyzed successfully by both analyses.

We use a machine with two Intel Xeon E5-2687W v4 3.00GHz CPUs and 512 GB of RAM (each with 12 cores x 2 hardware threads). We analyze 24 contracts at a time, each by a single thread. We set a timeout of 1500sec and a maximum RAM consumption of 50GB. (Most analyses take a lot less memory than that—typically under 5GB—so no global RAM pressure arises throughout our benchmarking, even when some individual contract analyses reach the limit of 50GB.)

5.2.1 Performance. The table in Figure 6 shows a summary of the performance of the symvalic analysis (core analysis + clients) with and without *Choice-Bound* over all contracts in the input dataset. As can be seen, the use of *Choice-Bound* virtually eliminates instances of out-of-memory and out-of-time execution, dropping the total to just 7 (out of the 744 contracts) instead of 57 for the original analysis. In terms of average time, *Choice-Bound* results in a 1.77x speedup. This is diluted by the large number of contracts with no scalability issues.

If we focus our attention to the smart contracts that run into problems with the default analysis (i.e., the 57 contracts with either out-of-memory errors or 1500sec timeouts), *Choice-Bound* exhibits an average execution time of 569.0sec. (It times out for 6 of these contracts and runs out of memory for 1, as shown in the table.) The lower bound for the average speedup over this set is 2.53x. (It is a lower bound both because executions that time out at 1500sec would take a lot longer if allowed to complete, but also because we conservatively charge executions that run out of memory only the time it took them to reach the out-of-memory error and not the full 1500sec.)

5.2.2 Completeness. We evaluate completeness in several different ways. The most straightforward is the overall analysis coverage (in terms of block coverage and variable coverage, i.e., how many low-level program variables have values) for all smart contracts in the input dataset.

Figure 7 shows the standard coverage metrics, per analysis. The first *Choice-Bound* column shows results for all successfully analyzed contracts (737 in total). The second *Choice-Bound* column and the default analysis column show results for the 687 contracts analyzed successfully by the default analysis (as well as by the *Choice-Bound* analysis). As can be seen, *Choice-Bound* maintains high coverage over all contracts (e.g., 92.9% block coverage vs. 93.5% for the original analysis) and, if we compare over the same set of contracts, there is virtually no loss of coverage (93.4% vs. 93.5% for block coverage).

For a more in-depth comparison of completeness, we consider the vulnerability warnings that the analysis issues, over the contracts analyzed by both configurations. Figure 8 tabulates the contracts flagged with of the two highest confidence levels. (The results for other confidence levels are very analogous.) As seen in the table, the report instances are nearly-identical, with very minimal completeness loss.

Warning type	*Choice-Bound*	Default symvalic analysis
HIGH: Call to Tainted Function	1.46%	1.75%
HIGH: Chainlink data feed may provide stale answers	2.18%	2.18%
HIGH: DoS Call can cause failure	0.44%	0.44%
HIGH: FlashLoan unchecked callback	0.29%	0.29%
HIGH: Guard can be overwritten	2.91%	2.91%
HIGH: Inconsistent Reentrancy guards	6.40%	6.40%
HIGH: Merkle node can be used as leaf	0.29%	0.29%
HIGH: Rare tainted money-sensitive var in external call	3.20%	3.20%
HIGH: Reentrancy	12.81%	12.95%
HIGH: SSTORE to tainted address	0.29%	0.29%
HIGH: Stale value in storage	0.73%	0.73%
HIGH: Suspicious decimal arithmetic	0.29%	0.15%
HIGH: Swap publicly reachable	5.09%	5.09%
HIGH: Tainted Ownership Guard	2.18%	2.33%
HIGH: Tainted delegatecall	1.02%	1.02%
HIGH: Tainted money-sensitive var in external call	7.57%	7.57%
HIGH: Twin calls	1.75%	1.75%
HIGH: Unchecked Low-Level Call	2.04%	2.04%
HIGH: Uniswap price manipulation potential	0.58%	0.58%
HIGH: Uniswap tainted token	0.15%	0.15%
HIGH: Unrestricted approve proxy	1.16%	1.16%
HIGH: Unrestricted transfer proxy	2.93%	3.93%
HIGH: Unrestricted transferFrom Proxy	0.87%	0.87%
HIGH: this.call()	3.06%	3.06%
HIGHEST: Call to Tainted Function	0.73%	0.73%
HIGHEST: Inconsistent Reentrancy guards	5.68%	5.82%
HIGHEST: Rare tainted money-sensitive var in external call	2.18%	2.18%
HIGHEST: Reentrancy	7.28%	7.42%
HIGHEST: Stale value in storage	0.58%	0.58%
HIGHEST: Unrestricted transferFrom Proxy	0.58%	0.58%

Fig. 8. Analysis high- and highest-confidence warnings (percentage of flagged contracts) over 687 total contracts: all that are successfully-analyzed by both analyses.

In all client analyses, the results are very close in number, with differences at the noise level. For 23 of the 30 warning categories, the sets of flagged contracts are exactly identical, whereas for the rest they differ by tiny amounts. This confirms that *Choice-Bound* largely maintains the completeness of the original analysis when the original analysis scales well.

An interesting observation concerns the results for the *Suspicious decimal arithmetic* warning category, where the *Choice-Bound* analysis yields more reports than the original. The reason is that this client analysis uses negation over the main bounded predicates: the results of an arithmetic operation are not by themselves suspicious, they are suspicious if *no other* checks over the arguments exist in the code. Therefore, incomplete results in the core analysis predicates yield *more* final reports.

5.3 Non-Determinism of Analysis Results

As discussed in Section 3.2, a potential "cost" of *Choice-Bound* is non-determinism. Although, in principle, most Datalog-based analyses contain non-deterministic elements, *Choice-Bound* magnifies the non-determinism, in ways that may be practically relevant: it introduces an arbitrary *choice* of elements at a point in the analysis where making this choice (as opposed to allowing all results) is

expected to be a highly performance-critical decision.

Thus, running a *Choice-Bound* version of an analysis is expected to be less robust, in terms of experimental outcomes, than the baseline analysis. In pathological cases, there is nothing to bound the variance, since an adversarial program can be made to render most of its code unreachable to an analysis that can only hold a certain number of inferences at a certain program point (e.g., points-to values for a key variable). In practice, the effect may be small, but should be factored into the considerations of any downstream analysis client.

To get an intuitive grasp of the expected variation, let us refer back to Figure 5 and compare the numbers for different runs. The figure shows 734 casts reported in the analysis of *batik* (with 747 for the original, full 2objH analysis over the same methods). Repeating the analysis another 8 times yields numbers that range from 724 to 740. Similarly, for 8 runs of *bloat* (which exhibits the largest completeness loss), the number will range from a low of 659 to a high of 683. (For perspective, per Figure 5, one run of the *Choice-Bound* analysis yielded 677 reports over the same methods for which the original 2objH analysis yields 705.) This variation is non-negligible, but is likely entirely tolerable for a client of an inherently-unsound analysis (e.g., for bug detection), whose results will inform human understanding and will be filtered by human inspection.

5.4 *Choice-Bound* Sensitivity Experiments

As presented in Section 4.1, *Choice-Bound* opens up a large design space and permits easy customization, much in the spirit of declarative analysis itself. This gives the analysis designer the freedom to experiment with different settings, targeting different sweet spots in the tradeoff curve between scalability and completeness. We next return to the analysis setup of Section 5.1, evaluating on the Doop framework, and examine different design choices, beyond our initial setup.

For illustration purposes, we pick two example possibilities. The first aims to produce a more complete analysis, partly sacrificing scalability. The second produces ultra-scalable analyses while sacrificing completeness.

5.4.1 `VarPointsTo`$::\{$`var,ctx`$\}_{503}$ $[[$`hobj,hctx`$]]$. In this case, we use the same choice-domain as our initial evaluation, however, our hash function modulo is increased to 503 from 101. The design choice limits every context-qualified variable to appearing at most 503 times in the analysis output. Once again, these 503 occurrences are identified by hashing both the heap object and the heap allocation context.

Naturally, this will allow for more tuples for each (var,ctx) combination, producing an analysis with higher completeness at the expense of scalability. Indeed, average speedup drops from *23.29x* to *5.25x* for the same nine benchmarks as shown in Figure 9. If we exclude the three benchmarks for which the default 2objH analysis does not terminate, we see a jump in average completeness (reachable app methods) from *59.96%* to *61.90%* as shown in Figure 10 (with the default analysis at *62.99%*).

5.4.2 `VarPointsTo`$::\{$`var`$\}_{541}$ $[[$`ctx,hobj,hctx`$]]$. In contrast, the analysis designer may opt for a "cost-conscious" design choice as described in Section 4.1. This choice limits the total output tuples/entries per local variable to at most 541 and every combination of context, heap object, and heap allocation context counts as a different tuple against this limit. This design choice does not care about "fairly distributing the pain" (of pruning values), instead choosing to make sure that the cost is bounded, no matter what: each variable can only contribute 541 entries to the final result.

This choice sacrifices analysis completeness, but achieves extreme scalability, with the average speedup jumping from *23.29x* to a whooping *88.85x* as shown in Figure 11. It is worth noting that even the largest benchmark in our suite, *dotCMS*, is analyzed in under 15 minutes under this configuration. Meanwhile, the average completeness (in reachable app methods, shown in

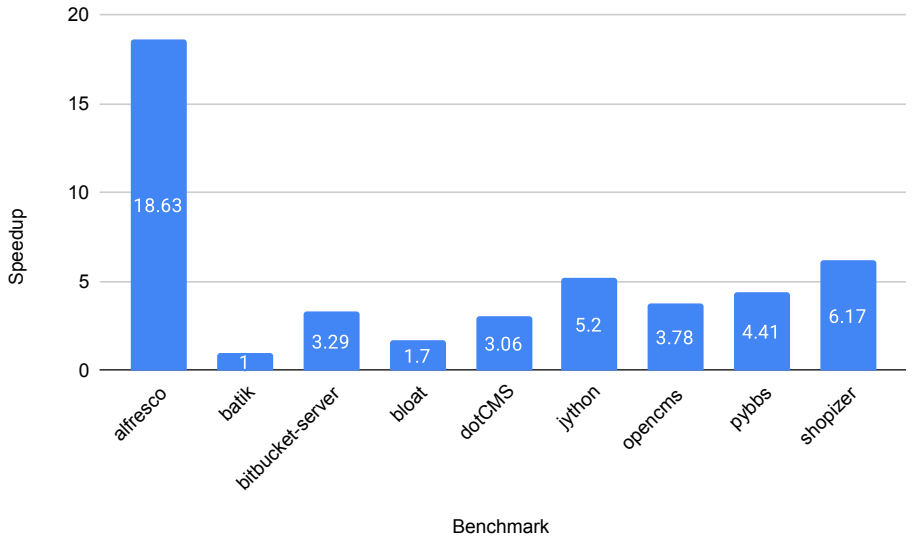


Fig. 9. Speedup per benchmark (multiplicative factors), for *Choice-Bound* `VarPointsTo`::{`var`,`ctx`}$_{503}$ [[`hobj`,`hctx`]] configuration. The three tallest bars are under-estimates, since the default analysis never terminated, in 24hrs.

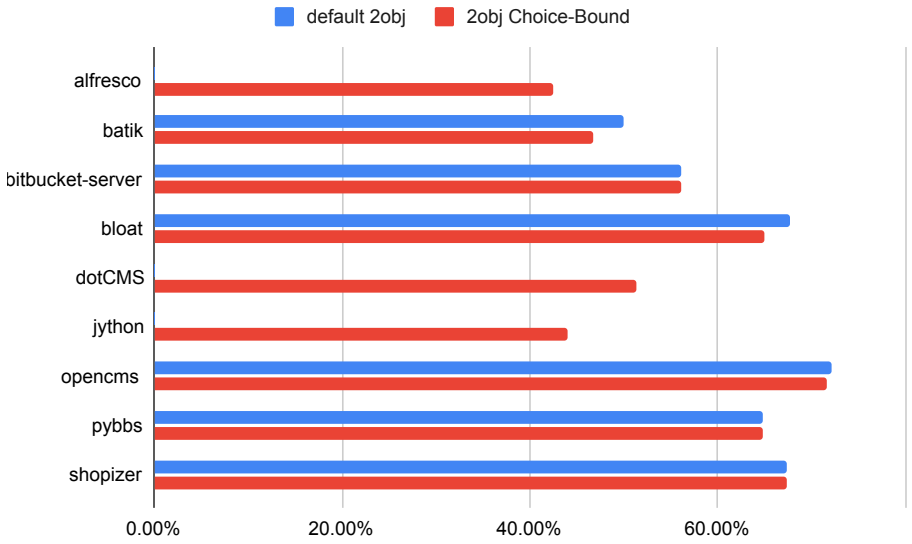


Fig. 10. App methods reachability for the default 2objH analysis and that with *Choice-Bound* `VarPointsTo`::{`var`,`ctx`}$_{503}$ [[`hobj`,`hctx`]] configuration.

Figure 12) drops from *62.99%* to *55.60%*. Again, the completeness loss is not noticeable over very large benchmarks but over smaller benchmarks for which the analysis loses precision and the computed results blow up. Arguably, this completeness loss is even less important than the numbers suggest: imprecisely-analyzed programs or program parts are less useful than precisely-analyzed ones.

The above experiments demonstrate the flexibility of *Choice-Bound*, as it is very easy for the

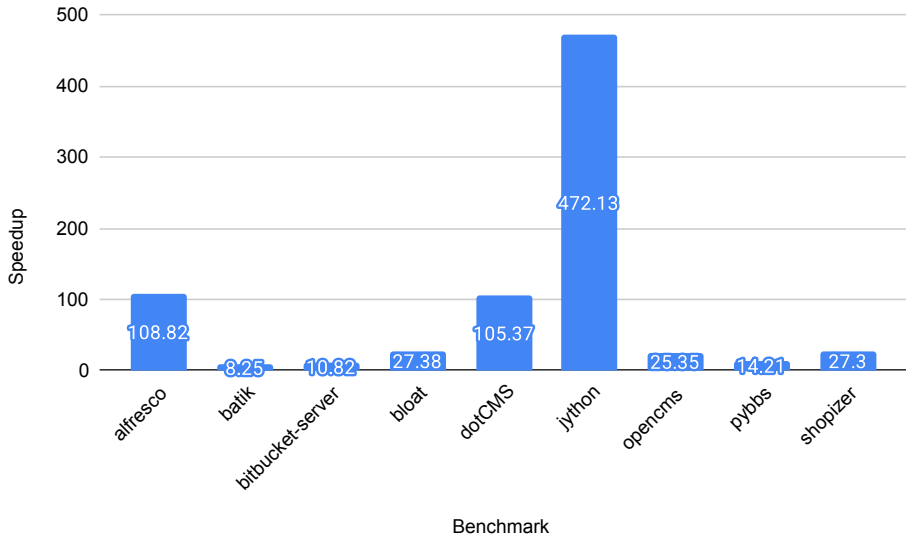


Fig. 11. Speedup per benchmark (multiplicative factors), for *Choice-Bound* `VarPointsTo`::{`var`}$_{541}$ [[`ctx`,`hobj`,`hctx`]] configuration. The three tallest bars are under-estimates, since the default analysis never terminated, in 24hrs.

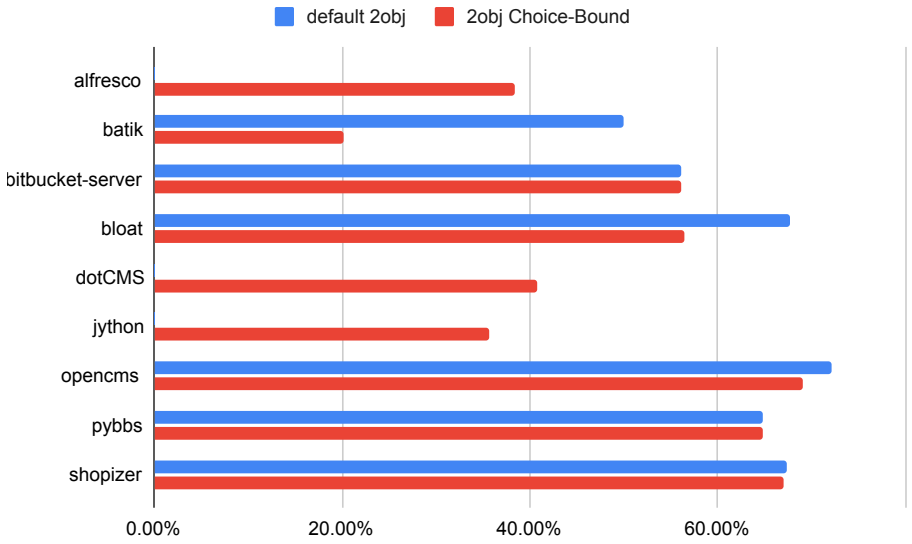


Fig. 12. App methods reachability for the default 2objH analysis and that with *Choice-Bound* `VarPointsTo`::{`var`}$_{541}$ [[`ctx`,`hobj`,`hctx`]] configuration.

analysis designer to manipulate the tradeoff between scalability and completeness, producing an analysis tailored to their needs.

6 Related Work

Program analysis optimization. There are numerous recent publications in making program analysis faster, with implementations purely in Datalog-based analyses. All of these techniques

(e.g., [2, 18, 20, 23, 24, 27, 36, 50] but also many more) improve the core analysis logic, in an effort to avoid unscalability for specific cases. This is a great endeavor and very significant, since improving the core analysis does more than avoid scalability issues: it often improves precision, i.e., yields a fundamentally better analysis, performance notwithstanding. However, as means to improve *performance*, such techniques still fall short of the goal: they neither provide a transparent, universal optimization, nor address the problem in all instances.

Generally, *Choice-Bound* is orthogonal to any algorithmic improvements in the analysis itself and its benefits can contribute to any other improvements. For instance, our evaluation of *Choice-Bound* was over (a superset of) the enterprise application benchmarks that the authors of the JackEE analysis [2] assembled. Yet, unlike JackEE, *Choice-Bound* achieves significant scalability benefits without having any special logic for Java enterprise applications or for custom versions of Java library classes. Despite being agnostic to the domain, the performance benefit of *Choice-Bound* is much greater than that of JackEE: both based on published numbers (and over Java 8, with a much smaller JDK) [2] and in our own experiments, the *Choice-Bound* analysis is 4-40x faster than that of JackEE for demanding enterprise applications, such as *opencms*, *alfresco*, or *dotCMS*. Combining the two approaches can be done transparently and may well lead to ever further improvements. (This and other combination experiments can be fruitful future work.)

Datalog with lattices. In terms of general, cross-cutting techniques, a modern trend that indirectly helps with scalability is to short-circuit an analysis by employing a *lattice* domain [30, 48, 49]. By introducing an ordering over the domain of abstract values, lattices allow the analysis to merge specific, fine-grained values into coarser ones, such as replacing a set of values with a top element ($\top$) when the set grows too large [28, 48, 49]. This technique can short-circuit computations and prevent the explosion of intermediate results.

In practice, this approach does not adequately address the scalability challenges of high-performance Datalog analyses. First, porting an analysis from a domain of explicit, enumerated values to a lattice domain may be non-trivial. There are language features that greatly automate the process. A prominent representative is Soufflé's *subsumption*. Per the documentation[1] "*[s]ubsumption permits to delete more specific tuples by more general tuples. A programmer can express this by declaring a partial-order in the form of a subsumptive clause.*"

However, the implementation of subsumption incurs significant overheads. It requires checking the conditions for short-circuiting and managing the lattice operations. At a high-level, every single tuple ever produced (for a relation that employs subsumption) needs to be checked to infer whether it is extraneous and should be ignored in favor of a more general value. Specifically, in the case of Soufflé, this necessitates the use of updatable data structures (such as BTreeDelete[2]). The result is degraded performance due to the evaluation overhead, but possibly also increased complexity in memory management.

It is telling that, while experimental engines such as Flix [28, 30] support lattices and subsumption, and Soufflé offers a subsumption construct, these features have not been widely adopted in major third-party research or industrial analysis artifacts. Anecdotally, we have conducted limited experiments with Soufflé's subsumption, which resulted in significant slowdowns, rather than speedups, for complex analyses. (In more detail, we wrote a context-sensitive analysis that short-circuits context to $\top$ when the contexts for a certain inference become too numerous. The subsumption rule then is "if variable v has abstract value u for the $\top$ context, this inference subsumes having that value for any specific context". We found that the approach slows down rule evaluation, since

[1] https://souffle-lang.github.io/subsumption

[2] https://github.com/souffle-lang/souffle/blob/master/src/include/souffle/datastructure/BTreeDelete.h

the subsumed tuples—i.e.,"v has value u under context c"—need to be suppressed from any rule that would infer them, if "v has value u under $\top$" was previously established.)

This limitation underscores the need for alternative techniques, such as our choice-based combination pruning, which can provide scalability without incurring the overhead associated with lattice operations.

Saturation-based techniques. The idea of stopping analysis at a certain threshold is fairly straightforward and used before in points-to analysis. Recent work by Wimmer et al. [56] uses the evocative name "*saturation*". The analysis stops enlarging the points-to set of a variable once it reaches a certain threshold in number of values. *Choice-Bound* has significant differences: it is applied to a very different setting (declarative analysis instead of imperative, context-sensitive analyses instead of context-insensitive that tries to become as fast as type-based); is transparent and virtually automatic instead of requiring intrusive changes; and introduces a general framework for considering the design parameters, especially for context-sensitive analyses.

Soufflé Datalog already supports a primitive form of saturation by means of the `limitsize` construct, which stops evaluation of a relation when its size (in tuples) reaches a threshold. The `limitsize` construct is too crude for applications such as those of *Choice-Bound*, however. It limits the whole relation and not combinations of its columns. As a result, it is very hard to invent appropriate `limitsize` values for benefit without sacrificing analysis completeness, especially if these values are not selected per-analyzed-program. Notably, the problem cannot be solved by just projecting out some columns of the relation and applying `limitsize` to the result: this would bound the projected relation, but not the original. *Choice-Bound* has the important benefit of applying directly to the bounded relation itself, at the core of its evaluation.

Other Bounding/Pruning. The theme of putting in place thresholds that stop analysis, sacrificing completeness in favor of performance, is quite general. The threshold does not need to be an automatic cut-off on the size of explicit result sets (as in saturation techniques), but can instead bound the processing performed by the analysis. Chakraborty et al. [7] bound the number of indirections covered by the analysis [7], for the purposes of call-graph generation for large JavaScript programs. An interesting result is that bounding the analysis processing can result in *asymptotic* improvements: Mathiasen and Pavlogiannis [32] show that a bounded version of the Andersen points-to analysis [1] (with the bound on how many statements of a certain type can be used to establish a points-to fact) is computable in sub-cubic time, in contrast to the unconstrained analysis. Relatedly, one can prune the inputs of the analysis beforehand, with the same goal of attaining performance while sacrificing completeness. Utture and Palsberg [52] propose a pre-processing tool that selects only the most relevant parts of the library for a given analysis. The tool then feeds this partial library plus the application code into a static analysis engine and manages to maintain 100% precision with a geometric mean speedup of 10x while only slightly sacrificing recall.

Widening. In intuitive terms, the *Choice-Bound* is reminiscent of standard *widening* techniques in Abstract Interpretation [8]. In fact, it is a matter of formulation whether one wants to view all pruning techniques (and not just *Choice-Bound*) as widening. (If the entire analysis is formulated in an abstract interpretation framework, this view is natural.) Although pruning can be viewed as widening, widening is a much broader concept: several non-pruning optimizations or other analysis ideas can also be thought of as widening. In practice, unlike *Choice-Bound*, one typically defines a widening operator, e.g., via human ingenuity, and does not just "apply" widening to oblivious analyses using uniform techniques.

Demand-driven and incremental analyses. Demand-driven program analyses (e.g., [14, 37, 43, 44, 46, 47, 54, 57, 59]) compute information only for program points of interest for a specific query.

This line of work is related to *Choice-Bound* in the sense that a demand-driven analysis benefits from only needing *part* of the results that an analysis is capable of producing. However, sacrificing some completeness for performance is distinctly different from only needing some of the results to begin with. *Choice-Bound* is aimed at analyses that do in principle make use of most of the potential results, and does not require fundamental changes to the analysis itself.

Similarly, incremental analyses (e.g., [48, 49]) can avoid re-computing results if these are guaranteed to not have changed, upon a small change of the input facts. Again, the intuition is similar: a part of the analysis inferences are sufficient for computing the desired result. However, the setting of *Choice-Bound* is one of full analysis evaluation, making no assumptions on having small input changes.

Non-deterministic choice in databases and in logic programming. In Prolog, non-deterministic choice is inherent to the language's operational semantics. Prolog explores multiple execution paths through backtracking, allowing for the representation of non-deterministic computations naturally.

The integration of Prolog and relational database technology (used by modern high-performance Datalog implementations) was a longstanding goal for both the logic programming and the database communities [21]. Early attempts focused on straightforwardly applying a Prolog language processor to a relational database system [4, 21, 53]. However, these efforts faced challenges due to the mismatch between the computational models of the two systems. Prolog employs a tuple-at-a-time model of computation [51], while relational databases operate on a set-at-a-time basis.

Datalog, as a purely declarative logic programming language, resolves the dilemma in favor of set-at-a-time computation. This means that non-determinism does not come "for free" in the language. To introduce controlled non-determinism into Datalog, Giannotti et al. [9] proposed a choice construct, allowing the expression of queries that require selecting arbitrary elements from a set. This construct enables the definition of functional dependencies within relations by enforcing a global constraint that, for a given key, only one tuple is selected. It is useful for tasks like breaking symmetry or selecting representative elements in data processing, such as in the implementation of greedy algorithms. Soufflé [15], as a modern Datalog engine, has incorporated the choice construct natively, building upon these foundational ideas. Soufflé's choice implementation is subsequently extended and leveraged in our approach to scale program analysis.

Other logic programming languages have explored similar concepts. IDLOG [38] extends Datalog with a more sophisticated choice construct that enables sampling queries. For example, the following IDLOG program defines the sampling query: "find an arbitrary set of employee samples which contains exactly two employees from each department."

```
select_two_emp(Name) <- emp[2](Name, Dept, N), N < 2
```

In this example, the notation `emp[2]` specifies that, for each department, exactly two employees are selected based on the tuple identifier (tid). The condition `N < 2` ensures that only the first two tuples (with tid 0 or 1) are considered in the computation of the relation `select_two_emp`. This mechanism is similar to the SQL:2003 window function with a condition on `ROW_NUMBER()` [16]:

```
SELECT name FROM (
  SELECT name, ROW_NUMBER() OVER (PARTITION BY dept) AS N FROM emp
) WHERE N < 3
```

While such sampling queries or window functions can be useful for limiting results in non-recursive queries, they cannot be employed within a recursive stratum of a Datalog program (due to recursion through aggregation). They operate on already-computed relations and do not prevent the previous computation from generating all possible tuples. In contrast, *Choice-Bound* can be applied

within the recursive evaluation itself, effectively preventing the computation of unnecessary tuples and thus enhancing scalability.

Relational databases provide features like `LIMIT` or `TOP` to restrict the number of tuples returned by a query. However, these mechanisms are crude and unwieldy when applied to recursive computations or complex analyses. They indiscriminately cut off the result set without considering the semantic importance of the data, leading to incomplete or inconsistent results. Moreover, they cannot be easily used to prevent the generation of large intermediate results during recursive evaluation since limits would be easily hit at early iterations of the semi-naive evaluation.

In summary, previous work has explored non-deterministic choice and techniques for limiting the size of computed relations, but these approaches either do not integrate seamlessly with recursive Datalog computations or introduce performance overheads that negate their benefits. Our approach leverages the choice construct in a novel way to bound the evaluation of predicates adaptively, providing a simple and effective solution to the scalability problem in Datalog-based program analyses.

7 Conclusions

We presented *Choice-Bound*: a technique that offers a powerful mechanism for tuning scalability vs. completeness in declarative computations. *Choice-Bound* leverages an efficient, simple mechanism of non-deteministic choice in modern Datalog engines. The idea is to implement multiplicity dependencies over existing relations: enforce that the same combination of pre-selected variables/columns can arise only up to a certain number of times. A large design space arises from this simple principle, offering expressiveness and flexibility.

Applied to program analysis algorithms, which can unpredictably fail to scale, *Choice-Bound* has significant value: previously unscalable analyses can now become entirely realistic, at the expense of a small loss in completeness. (More non-determinism is also introduced, but non-determinism, from several sources, is inevitable in all analyses we have encountered.) We apply *Choice-Bound* to pre-existing, large Datalog program analysis frameworks, such as Doop and its "ideal" analysis, 2-object-sensitive+heap. In subject programs also examined in past literature, for which the default analysis had difficulties scaling, *Choice-Bound* achieves speedups typically well over 10x. Analyses that would not terminate in over 24 or 48 hours now run in sub-hour time. The result is a powerful tool in the hands of an analysis designer, permitting customization and obtaining results even for highly-complex analyses and large inputs.

Acknowledgments

We gratefully acknowledge funding by ERC Advanced Grant PINDESYM (101095951).

Data-Availability Statement

Choice-Bound has been implemented in the Doop open-source repository. An artifact[3] is also available, which includes a copy of Doop, a pre-compiled binary of the symvalic analysis and the required datasets for reproducing the results of the experimental evaluation.

References

[1] Lars O. Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph. D. Dissertation. DIKU, University of Copenhagen.

[2] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 794–807. https://doi.org/10.1145/3385412.3386026

[3] Anastasios Antoniadis, Ilias Tsatiris, Neville Grech, and Yannis Smaragdakis. 2025. *Artifact: Universal Scalability in Declarative Program Analysis (with Choice-Based Combination Pruning).* https://doi.org/10.5281/zenodo.15723754

[4] H. L. Berghel. 1985. Simplified integration of Prolog with RDBMS. *SIGMIS Database* 16, 3 (April 1985), 3–12. https://doi.org/10.1145/2147769.2147770

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the 21st Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[6] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[7] Madhurima Chakraborty, Aakash Gnanakumar, Manu Sridharan, and Anders Møller. 2024. Indirection-Bounded Call Graph Analysis. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:22. https://doi.org/10.4230/LIPIcs.ECOOP.2024.10

[8] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation* 2, 4 (1992), 511–547.

[9] Fosca Giannotti, Dino Pedreschi, Domenico Saccà, and Carlo Zaniolo. 1991. *Non-determinism in deductive databases*. Springer Berlin Heidelberg, 129–146. https://doi.org/10.1007/3-540-55015-1_7

[10] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1176–1186. https://doi.org/10.1109/ICSE.2019.00120

[11] Neville Grech, George Kastrinis, and Yannis Smaragdakis. 2018. Efficient Reflection String Analysis via Graph Coloring. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 26:1–26:25. https://doi.org/10.4230/LIPIcs.ECOOP.2018.26

[12] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133926

[13] Sergio Greco and Carlo Zaniolo. 1998. Greedy Algorithms in Datalog with Choice and Negation. In *IJCSLP*. https://api.semanticscholar.org/CorpusID:15290518

[14] Nevin Heintze and Olivier Tardieu. 2001. Demand-Driven Pointer Analysis. In *Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

[15] Xiaowen Hu, Joshua Karp, David Zhao, Abdul Zreika, Xi Wu, and Bernhard Scholz. 2021. The Choice Construct in the Soufflé Language. In *Programming Languages and Systems*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 163–181.

[16] ISO/IEC. 2003. Information Technology—Database Languages—SQL—Part 2: Foundation (SQL/Foundation). International Standard ISO/IEC 9075-2:2003.

[17] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (oct 2018), 29 pages. https://doi.org/10.1145/3276510

[18] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.* 6, POPL, Article 58 (jan 2022), 29 pages. https://doi.org/10.1145/3498720

[19] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (oct 2017), 28 pages. https://doi.org/10.1145/3133924

[20] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

[21] S. Kunifuji and H. Yokota. 1982. Prolog and Relational Databases for 5th Generation Computer Systems. In *Proceedings of the Workrhop on Logical Bases for Data Bases*.

[22] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proc. of the 24th Symp. on Principles of Database Systems* (Baltimore, Maryland, USA) *(PODS '05)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1065167.1065169

[23] 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[24] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 10 (may 2020), 40 pages. https://doi.org/10.1145/3381915

[25] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[26] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*. Springer, 139–160. https://doi.org/10.1007/11575467_11

[27] 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. *Proc. ACM Program. Lang.* 7, PLDI (2023), 539–564. https://doi.org/10.1145/3591242

[28] Magnus Madsen and Ondrej Lhoták. 2018. Safe and sound program analysis with Flix. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 38–48. https://doi.org/10.1145/3213846.3213847

[29] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.

[30] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

[31] Chris Martens, Robert J. Simmons, and Michael Arntzenius. 2024. Finite-Choice Logic Programming. arXiv:2405.19040 [cs.PL] https://arxiv.org/abs/2405.19040

[32] Anders Alnor Mathiasen and Andreas Pavlogiannis. 2021. The fine-grained and parallel complexity of andersen's pointer analysis. *Proc. ACM Program. Lang.* 5, POPL, Article 34 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434315

[33] Ana 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.

[34] Mayur Naik. 2011. Chord: A Versatile Platform for Program Analysis. In *2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. Tutorial.

[35] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[36] 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

[37] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*. ACM, 264–274. https://doi.org/10.1145/2259016.2259050

[38] Yeh-Heng Shen. 1990. IDLOG: extending the expressive power of deductive database languages. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (Atlantic City, New Jersey, USA) *(SIGMOD '90)*. Association for Computing Machinery, New York, NY, USA, 54–63. https://doi.org/10.1145/93597.93621

[39] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 245–251.

[40] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 17–30.

[41] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic Value-Flow Static Analysis: Deep, Precise, Complete Modeling of Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 163 (oct 2021), 30 pages. https://doi.org/10.1145/3485540

[42] 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 485–495.

https://doi.org/10.1145/2594291.2594320
[43] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22
[44] Manu Sridharan and Rastislav Bodik. 2006. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 387–400.
[45] Manu Sridharan and Stephen J. Fink. 2009. The Complexity of Andersen's Analysis in Practice. In *Proc. of the 16th International Symp. on Static Analysis* (Los Angeles, CA, USA) *(SAS '09)*. Springer, 205–221. https://doi.org/10.1007/978-3-642-03237-0_15
[46] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proc. of the 20th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817
[47] Yulei Sui and Jingling Xue. 2016. On-demand Strong Update Analysis via Value-flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296
[48] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29. https://doi.org/10.1145/3276509
[49] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. 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, 1–15. https://doi.org/10.1145/3453483.3454026
[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. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (oct 2021), 27 pages. https://doi.org/10.1145/3485524
[51] Shalom Tsur and Carlo Zaniolo. 1986. LDL: A Logic-Based Data Language. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 33–41.
[52] Akshay Utture and Jens Palsberg. 2022. Fast and precise application code analysis using a partial library. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 934–945. https://doi.org/10.1145/3510003.3510046
[53] Raf Venken. 1984. *lhe Interaction between Prolog and Relational Databases*. Technical Report. Internal Report BIM-prolog IR6.
[54] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744
[55] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*. Springer, 97–118. https://doi.org/10.1007/11575467_8
[56] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.* 8, PLDI, Article 187 (June 2024), 24 pages. https://doi.org/10.1145/3656417
[57] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. ACM, 155–165. https://doi.org/10.1145/2001420.2001440
[58] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 239–248. https://doi.org/10.1145/2594291.2594327
[59] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464

Received 2025-03-23; accepted 2025-08-12