Program Repair Guided by Datalog-Defined Static Analysis

Yu Liu
National University of Singapore
Singapore
liuyu@comp.nus.edu.sg

Sergey Mechtaev*
University College London
United Kingdom
s.mechtaev@ucl.ac.uk

Pavle Subotić
Microsoft
Serbia
pavlesubotic@microsoft.com

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Automated program repair relying on static analysis complements test-driven repair, since it does not require failing tests to repair a bug, and it avoids test-overfitting by considering program properties. Due to the rich variety and complexity of program analyses, existing static program repair techniques are tied to specific analysers, and thus repair only narrow classes of defects. To develop a general-purpose static program repair framework that targets a wide range of properties and programming languages, we propose to integrate program repair with Datalog-based analysis. Datalog solvers are programmable fixed point engines which can be used to encode many program analysis problems in a modular fashion. The program under analysis is encoded as Datalog facts, while the fixed point equations of the program analysis are expressed as recursive Datalog rules. In this context, we view repairing the program as modifying the corresponding Datalog facts. This is accomplished by a novel technique, symbolic execution of Datalog, that evaluates Datalog queries over a symbolic database of facts, instead of a concrete set of facts. The result of symbolic query evaluation allows us to infer what changes to a given set of Datalog facts repair the program so that it meets the desired analysis goals. We developed a symbolic executor for Datalog called Symlog, on top of which we built a repair tool SymlogRepair. We show the versatility of our approach on several analysis problems — repairing null pointer exceptions in Java programs, repairing data leaks in Python notebooks, and repairing four types of security vulnerabilities in Solidity smart contracts.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; **Automated static analysis**; • **Theory of computation → Constraint and logic programming**; • **Information systems → Query languages**.

KEYWORDS

program repair, static analysis, Datalog, symbolic execution

ACM Reference Format:
Yu Liu, Sergey Mechtaev, Pavle Subotić, and Abhik Roychoudhury. 2023. Program Repair Guided by Datalog-Defined Static Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616363

1 INTRODUCTION

Program analysis determines if the behaviour of a given program satisfies a certain property. Program repair based on static program analysis is an attractive technique, since, in contrast to test-driven program repair [23], it does not require executing the program, thus significantly simplifying deployment, and it avoids test-overfitting by taking program properties into account. A plethora of static analysis techniques exist, each characterised by a unique set of underlying mechanisms, target programming languages, and target program properties. These characteristics significantly impact the implementation and optimisation of each analysis. Existing program repair methods based on static analysis do not accommodate this diversity, and are inherently tied to specific analysers. This restricts their capacity to rectify a wide class of defects.

The goal of this work is to design a modular static program repair system that fixes bugs violating a wide range of program properties across multiple programming languages. The modularity implies that the conceptual components, the modeling of programming language semantics, analysis algorithms, search space construction, and patch generation, are independent and reusable. Such a design radically reduces the complexity of creating a static program repair system capable of efficiently addressing a wide spectrum of defects.

Our proposed architecture leverages the modular static analysis framework based on Datalog. In this setup, Datalog acts as a domain specific language (DSL) for defining program analyses. A program is encoded into a database i.e., a set of input relations, and the program analysis constraints are defined as a Datalog query. A Datalog solver acts as a programmable fixed point engine that computes a least fixed point solution to the program analysis constraints. Datalog enables both a succinct and modular encoding of the program analysis, while providing high efficiency and interoperability [34].

To introduce program repair to this setup, we propose a novel approach called symbolic execution of Datalog (SEDL). We define the repair search space by injecting symbols into the input relations

[*] Corresponding author

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616363

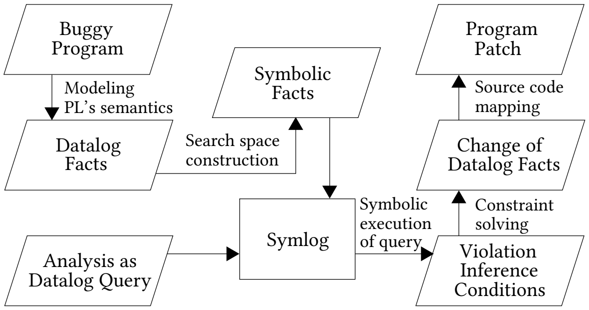


Figure 1: SymlogRepair, a program repair system using symbolic execution of Datalog (SEDL) implemented in Symlog.

representing the buggy program. These symbols denote unknown constants, unknown predicates, and unknown truthfulness of facts in the database. The resulting symbolic database represents a set of changes to the original database. SEDL executes the Datalog analysis query on the symbolic database, and the outcome of this execution summarises how values from various subdomains, when substituted in place of symbols, impact the inference of various output facts. These subdomains, expressed as logical constraints over the symbols, capture the dependencies between the input and output of the query; in analogy with path conditions in conventional symbolic execution, we call them inference conditions. Given an inference condition for an output fact, any satisfying assignment of symbols enables the inference of this fact, and any falsifying assignment disables its inference. To repair the program, we use an SMT solver to find a valuation of symbols that produces the desired query output, such as the absence of property violations.

We implemented SEDL in a tool called Symlog, which utilises the state-of-the-art Datalog engine Soufflé [20]. Symlog transforms a given symbolic database and query into a meta-program, which, when executed using a conventional Datalog evaluator, produces the result of symbolically executing the original query on the original database. To mitigate state explosion, we implemented two optimisation techniques for the meta-program. First, we group possible valuations of symbols into equivalent classes to eliminate redundant exploration of identical proof trees during symbolic Datalog evaluation. Second, we apply delta-debugging [47] to efficiently identify dependencies between input and output facts. The key advantage of the meta-programming approach is that it allows us to leverage the optimisations provided by existing Datalog engines [37] without modifying the engines. Lastly, we implemented SymlogRepair that generates candidate patches by solving Symlog's inference conditions for program analysis outputs with Z3 [11] to avoid property violations, and then ranks the candidates based on minimality. The architecture of SymlogRepair is shown in Figure 1.

To demonstrate the modularity of the proposed architecture, we specialised SymlogRepair for six types of defects across three programming languages. First, we realised SymlogRepair[NPE, Java], which uses SEDL to repair null pointer exception (NPE) bugs in Java programs [13]. We use Doop [8] to model the semantics of Java, and Digger [35] to define the analysed property. Second, we realised SymlogRepair[Leak, Python], the first approach for repairing data leaks in Python notebooks [38]. We use the rules by Yang *et al.* [45] to model the semantics of Python, and to define the analysed property. Finally, we realised SymlogRepair[Securify2, Solidity] for repairing four classes of security vulnerabilities in Solidity smart contracts: access control, unhandled exception, reentrancy, and locked ether. We use Securify2 [2, 41] to model the semantics of smart contracts, and to define the analysed properties.

In our evaluation, SymlogRepair[NPE, Java] correctly repaired 8 out of 10 NPE bugs detected by Digger, outperforming NPEX [25] by 1 correct patch, and AlphaRepair [44] and InCoder [16] by 6 and 8 correct patches respectfully. SymlogRepair[Leak, Python] correctly repaired 6 out of 11 notebook preprocessing leakage bugs detected by the analyser, while AlphaRepair and InCoder failed to repair any. SymlogRepair[Securify2, Solidity] correctly repaired 63 out of 64 vulnerabilities in smart contracts, outperforming Elysium [15] by 47 correct patches. Our optimisations played a key role in enabling the generation of patches, as our tool without optimisations repair fewer bugs due to running out of memory.

The contributions of our work are summarised as follows:

- A static program repair architecture based on Datalog, where repair is formulated as the problem of modifying a given database representing the buggy program to make the query representing the analysis detect zero property violations;
- Symbolic execution of Datalog (SEDL), the enabling component of our architecture, which identifies how changes to the database impact the result of a query;
- An efficient implementation of SEDL in a tool called Symlog[1]; and its application to program repair, SymlogRepair.
- SymlogRepair instances: SymlogRepair[NPE, Java] for fixing NPE bugs in Java programs, SymlogRepair[Leak, Python] for repairing data leak defects in Python notebooks, and SymlogRepair[Securify2, Solidity] for repairing four classes of security vulnerabilities in Solidity smart contracts, as well as an evaluation of these instances on realistic bugs.

2 OVERVIEW

In this section, we give an overview of our approach. We first explain how Datalog is used to define a program analysis. Then, we demonstrate how SEDL, the key technical novelty of this work, symbolically executes this analysis. Finally, we show SymlogRepair fixes a null pointer exception (NPE) defect.

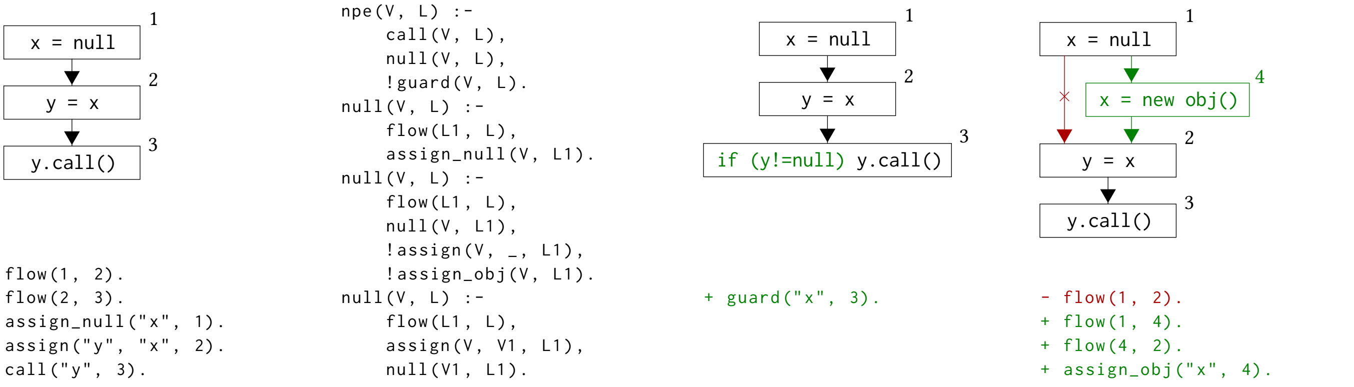
Since the definitions of practical analyses require hundreds of lines of code, and the semantics of real-world programming languages is extremely complex, for the illustrative purpose, we use simplified language and analysis. The language contains elementary statements (variable initialisations, assignments and method calls), and call statement guards that check if a variable is not null. The formal grammar of this language is given in Figure 4.

Consider the following program written in our simple language:

```
x = null
y = x
y.call()
```

This program triggers an NPE as y will be null when `y.call()` is performed. The remainder of this section will describe how this bug can be detected and fixed with Datalog.

[1] Implementation of Symlog and SymlogRepair: https://github.com/symlog/symlog



```
flow(1, 2).
flow(2, 3).
assign_null("x", 1).
assign("y", "x", 2).
call("y", 3).
```

(a) A program's CFG and its EDB representation.

```
npe(V, L) :-
    call(V, L),
    null(V, L),
    !guard(V, L).
null(V, L) :-
    flow(L1, L),
    assign_null(V, L1).
null(V, L) :-
    flow(L1, L),
    null(V, L1),
    !assign(V, _, L1),
    !assign_obj(V, L1).
null(V, L) :-
    flow(L1, L),
    assign(V, V1, L1),
    null(V1, L1).
```

(b) NPE analysis expressed as a Datalog query.

```
+ guard("x", 3).
```

(c) Repair with guard and the corresponding EDB change.

```
- flow(1, 2).
+ flow(1, 4).
+ flow(4, 2).
+ assign_obj("x", 4).
```

(d) Repair with assignment and the corresponding EDB change.

$$\dfrac{\dfrac{\dfrac{\texttt{flow(1,2)} \qquad \texttt{assign\_null("x",1)}}{\texttt{null("x",2)}} \qquad \texttt{flow(2,3)} \qquad \texttt{assign("y","x",2)}}{\texttt{null("y",3)}} \qquad \texttt{call("y",3)} \qquad \texttt{!guard("y",3)}}{\texttt{npe("y",3)}}$$

(e) A proof tree of the output of NPE analysis query demonstrating the presence of an NPE bug.

Figure 2: A program with a null pointer exception (NPE) bug, an analysis that detects this bug, its proof tree, and two repairs.

$$\dfrac{\dfrac{\xi_3\,\texttt{flow}(\alpha_1,\alpha_2) \qquad \texttt{assign\_null("x",1)}}{\texttt{null}(\texttt{"x"},\alpha_2)}\ \xi_1 \land \alpha_1 = 1 \qquad \xi_4\,\texttt{flow}(\alpha_2,\alpha_3) \qquad \texttt{!assign}(\texttt{"x"},\_,\alpha_2) \qquad \xi_6\,\texttt{assign\_obj}(\alpha_5,\alpha_6)}{\texttt{null}(\texttt{"x"},\alpha_2)}\ \xi_4 \land \alpha_3 = 3 \land \neg(\xi_6 \land \alpha_5 = \text{“}x\text{”} \land \alpha_6 = \alpha_2)$$

$$\dfrac{\dfrac{\texttt{null}(\texttt{"x"},\alpha_2) \qquad \xi_2\,\texttt{flow(2, 3)} \qquad \texttt{assign("y","x",2)}}{\texttt{null("y",3)}}\ \xi_2 \qquad \texttt{call("y",3)} \qquad \xi_5\,\texttt{guard}(\alpha_4,\ 3)}{\texttt{npe("y",3)}}\ \neg(\xi_5 \land \alpha_4 = \text{“}y\text{”})$$

Figure 3: A symbolic proof tree (in two parts) of an NPE after inserting a statement between the nodes 1 and 2.

$$\begin{array}{rcl}
\langle\text{stmt}\rangle & ::= & \langle\text{var}\rangle = \texttt{new obj()} \ \mid\ \langle\text{var}\rangle = \texttt{null} \ \mid \\
 & & \langle\text{var}\rangle = \langle\text{var}\rangle \ \mid\ \langle\text{call}\rangle \\
\langle\text{guard}\rangle & ::= & \langle\text{var}\rangle \texttt{ != null} \\
\langle\text{call}\rangle & ::= & \langle\text{var}\rangle\texttt{.call()} \ \mid\ \texttt{if (}\langle\text{guard}\rangle\texttt{)}\ \langle\text{var}\rangle\texttt{.call()}
\end{array}$$

Figure 4: Example programming language.

2.1 Program Analysis with Datalog

Datalog is a query language based on logic programming. A Datalog query is a set of Horn clauses comprising of a set of body predicates (left part) and a head predicate (right part). The symbol `:-` shows that the left part is logically implied from the right part. A Datalog query is executed against a database of facts referred to as the extensional database (EDB) and produces a set of derived facts, referred to as the intensional database (IDB). Since Datalog implements the semantics of fix point computation, and many program analysis algorithms are instances of fix point computation, Datalog is often used as a language for defining program analyses.

To analyse a program with Datalog, the program is represented as an EDB. For instance, we can construct a program's control-flow graph (CFG) as a set of facts as shown in Figure 2a. In this EDB, the predicate `flow` encodes arcs of the CFG, *e.g.* `flow(1,2)` states that there is an arc between the nodes 1 and 2. The fact `assign_null ("x",1)` states that the variable `x` is assigned to null in the node 1; `assign("y","x",2)` states that the value of `x` is assigned to `y` in the node 2, and `call("y",3)` states that a method of the object stored in `y` is called in the node 3.

Figure 2b shows an example analysis for detecting NPEs. This analysis defines two predicates. The predicate `npe(V,L)` states that there may be a dereferencing of null stored in the variable `V` in the node `L`. The predicate `null(V,L)` states that the variable `V` may store null at the entry of the node `L`. The predicates are defined using four rules. The first rule states an NPE happens when a method of a null object is called without a guard. The second rule states that a variable may be null at the entry of a node if it is assigned to null in a parent node. The third rule states that a variable may be null at the entry of a node if it is not assigned at the entry of a parent node, and may be null at the entry of that parent node. The fourth rule state that a variable may be null at the entry of a node if it is assigned to a variable that may be null in a parent node.

When a query is executed on an EDB, a Datalog engine computes all facts that can be inferred from the database using the rules of the query. For example, for the above program and analysis, Datalog infers the fact `npe("y",3)` stating that there may be an NPE in the node 3. In Figure 2e, a proof tree visualises how this fact is derived.

2.2 Symbolic Execution of Datalog

The key technical novelty of this work is SEDL that determines how a change to the EDB affects the output of a given query. A set of changes to the database is encoded using symbols, and then the query is executed symbolically. The result of symbolic execution of a Datalog query compactly summarises the relations between the input symbols and the output of the query.

Consider the following symbolic EDB obtained from the EDB in Figure 2a by injecting symbolic constants α_i representing unknown constants and symbolic signs ξ_i representing unknown truthfulness (if the associated facts are true of false):

```
ξ1 flow(1, 2).              assign("y", "x", 2).
ξ2 flow(2, 3).              call("y", 3).
ξ3 flow(α1, α2).            ξ5 guard(α4, 3).
ξ4 flow(α2, α3).            ξ6 assign_obj(α5, α6).
assign_null("x", 1).
```

Any valuation of these symbols corresponds to a concrete EDB. For example, the original EDB corresponds to the following valuation:

$$\{\, \xi_1 \mapsto T, \xi_2 \mapsto T, \xi_3 \mapsto F, \xi_4 \mapsto F, \xi_5 \mapsto F, \xi_6 \mapsto F, \dots \}.$$

Executing the analysis in Figure 2b on this symbolic EDB with SEDL yields the fact `npe("y",3)` and the inference condition ϕ for this fact defined as a finite set of disjuncts:

$$\begin{aligned}\phi \triangleq\ & \xi_1 \wedge \xi_2 \wedge \neg(\xi_5 \wedge \alpha_4 = \text{“}y\text{”}) \\ & \vee (\xi_2 \wedge \xi_3 \wedge \alpha_1 = 1 \wedge \xi_4 \wedge \alpha_3 = 2 \\ & \quad \wedge \neg(\xi_6 \wedge \alpha_5 = \text{“}x\text{”} \wedge \alpha_6 = \alpha_2) \wedge \neg(\xi_5 \wedge \alpha_4 = \text{“}y\text{”})) \\ & \vee \dots\end{aligned}$$

Each disjunct represents a subset of programs encoded in the symbolic EDB where the defect `npe("y",3)` is detected using the same proof. The first disjunct corresponds to the original program. The second disjunct corresponds to buggy programs with a statement inserted between the nodes 1 and 2, for example,

```
x = null
y = new obj()
y = x
y.call()
```

A symbolic proof tree corresponding to the second disjunct is given in Figure 3. In this proof tree, the labels at each production capture the conditions under which the rule can be applied. The disjunct is computed as a conjunction of all these labels.

2.3 Fixing an NPE with SymlogRepair

We now demonstrate how to apply SEDL for program repair. Given a buggy program and a Datalog-defined analyser that detects the bug, SymlogRepair defines the patch search space by injecting symbols into an EDB representation of the program. The method of search space construction is determined by a domain expert based on the analysed property and the modelling of the programming language semantics in the EDB. Section 5 shows how to generate search spaces for realistic analyses. Here we assume that the search space is defined by the symbolic EDB in Section 2.2.

Since not all assignments of symbols may correspond to syntactically valid programs, we restrict them with structural constraints ψ. For the example above, the structural constraints capture that the `flow` relation does not form branches or cycles, and that each node contains a single statement.

To generate a patch for the program, we construct and solve a repair condition. A repair condition is a formula over symbols injected into the EDB representation of the program, such that any satisfying assignment of this formula corresponds to a program for which the analysis does not infer property violations. We obtain this condition by symbolically executing the analysis using our implementation of SEDL, Symlog. For the example above, the repair condition is $\neg\phi \wedge \psi$, since it ensures the fact `npe("y",3)` will not be inferred, and the program is syntactically valid.

Since repair constraints accept multiple solutions, we only consider minimal solutions, such that all other satisfying assignments change a superset of values corresponding to the original program. A minimal solution for the above example is

$$\{\, \xi_1 \mapsto T, \xi_2 \mapsto T, \xi_3 \mapsto F, \xi_4 \mapsto F, \xi_5 \mapsto T, \alpha_4 \mapsto \text{“}y\text{”}, \dots \}.$$

It corresponds to the CFG and EDB presented in Figure 2c.

Another minimal solution shown in Figure 2d is

$$\begin{gathered}\{\, \xi_1 \mapsto F, \xi_2 \mapsto T, \xi_3 \mapsto T, \xi_4 \mapsto T, \xi_5 \mapsto F, \xi_6 \mapsto T, \\ \alpha_1 \mapsto 1, \alpha_2 \mapsto 4, \alpha_3 \mapsto 2, \alpha_5 \mapsto \text{“}x\text{”}, \alpha_6 \mapsto 4, \dots \}.\end{gathered}$$

The second solution shows the ability of our approach to “invent” new values that do not exist in the original database.

3 SYMBOLIC EXECUTION OF DATALOG

In this section, we introduce relevant background on Datalog, and define our key technical novelty, symbolic execution of Datalog.

3.1 Background

Datalog is a query language based on Horn clauses in the form $\mathtt{L}_0$`:-`$\mathtt{L}_1, \dots, \mathtt{L}_n$, where each $\mathtt{L}_i$ is a *literal* $\mathtt{p}(\mathtt{t}_1, \dots, \mathtt{t}_n)$ such that `p` is a *predicate* and $\mathtt{t}_i$ are *terms*. A term is either a constant or a variable. *Facts* are ground literals, *i.e.* literals without variables. A *ground substitution* $\theta \triangleq \{x_1/c_1, \dots, x_n/c_n\}$ is a mapping from variables to constants. We denote an application of the substitution θ to the term `t` as $\mathtt{t}\theta$, and to the literal `L` as $\mathtt{L}\theta$. EDB, *extensional database*, is a set of input facts. IDB, *intensional database*, is a set of output facts determined by the query. The *Herbrand Base* HB is the set of all expressible facts. EHB denotes the set of all HB facts where the predicate appears in EDB; IHB denotes the set of all HB facts where the predicate appears in IDB, but not EDB. The *semantics* of a Datalog program P can be represented as a function $\mathfrak{M}_P : 2^{\mathrm{EHB}} \to 2^{\mathrm{IHB}}$ that for each existential database computes all IDB-facts that logically follow from that database. Datalog allows specifying a goal to select a subset of its outputs that are subsumed by the goal. For example, the fact `f(1,1)` is subsumed by the goal `f(X,X)`, but `f(1,2)` is not.

Consider the Datalog query P in Figure 2b that specifies an NPE analysis for a simple programming language defined in Figure 4. Let the CFG of an analysed program be represented via an EDB consisting of nodes, arcs, such as`flow(1,2)`, and information about statements such as the fact `assign("y","x",2)` stating that the value of `x` is assigned to `y` in the node 2. The semantics of such program, $\mathfrak{M}_P$, is a function that for an CFG represented as a set of facts returns the set of IDB facts `npe` and `null` as follows:

$$\mathfrak{M}_P\left(\left\{\begin{array}{c}\texttt{flow}(1,2),\\ \texttt{flow}(2,3),\\ \texttt{assign\_null}(\text{“}x\text{”},1),\\ \texttt{assign}(\text{“}y\text{”},\text{“}x\text{”},2),\\ \texttt{call}(\text{“}y\text{”},3)\end{array}\right\}\right) = \left\{\begin{array}{c}\texttt{npe}(\text{“}y\text{”},3)\\ \texttt{null}(\text{“}x\text{”},2)\\ \texttt{null}(\text{“}y\text{”},3)\end{array}\right\}$$

To define $\mathfrak{M}_P$, we use the proof-theoretic interpretation of Datalog in which the meaning of a program is defined as the set of all facts that can be inferred from the program. For a given rule $\mathtt{L}_0$ `:-` $\mathtt{L}_1, ..., \mathtt{L}_n$ and a set of ground facts $\mathtt{F}_1, ..., \mathtt{F}_n$, the output fact $\mathtt{F}_0$ can be inferred in one step from $\mathtt{F}_1, ..., \mathtt{F}_n$ if there is a substitution θ such that for any $i \in 0..n$, $\mathtt{F}_i = \mathtt{L}_i\theta$. This procedure is referred to in the Datalog literature as the *elementary production principle* (EPP). A ground fact F can be inferred from a program P if either $F \in P$ or F can be inferred by applying EPP a finite number of times. The sequence of applications of EPP used to infer a fact F from P forms a proof of F from P.

A proof can be represented using a *proof tree*. For a given Datalog query Q, an EDB E, and a ground fact F, a proof tree of F from E is the tree $(F, R, \theta, \{t_1, ..., t_n\})$ such that $n = 0$ iff $F \in E$, otherwise $R \triangleq \mathtt{L}_0$ `:-` $\mathtt{L}_1, ..., \mathtt{L}_n$ is a rule from Q, $F = \mathtt{L}_0\theta$, and t_i are proof trees of $\mathtt{L}_i\theta$. For the NPE analysis described above, the proof tree of the fact `npe("y",3)` is visualised in Figure 2e.

3.2 Semantics of SEDL

The goal of *symbolic execution of Datalog* (SEDL) is to identify how varying values in a database impacts the result of executing a query on this database. To achieve this, SEDL executes a Datalog query on an abstracted database which represents a set of concrete databases. The result of symbolic execution is a set of pair of output facts and logical constraints that enable the inference that facts. Through this process, SEDL succinctly summarises the outcome of executing the query on all of the encoded concrete databases.

By analogy with symbolic execution of conventional programs, we abstract a given database by injecting *symbols* that represent unknown information. SEDL uses three categories of symbols: symbolic constants, symbolic predicates and symbolic signs. We denote *symbolic constants* as α, β, γ that range over finite domains of numbers, strings, etc. We use them to represent facts with unknown constants, e.g. `flow(`α`,`β`)`. We denote *symbolic predicates* as ρ that range over all predicates. We use them to represent facts with unknown predicates, e.g. ρ`(1,2)`. We denote *symbolic signs* as ξ that range over booleans. We associate a symbolic sign with a fact to control if the fact is positive or negative. Collectively, we refer to such symbols as $\Sigma \triangleq \{\sigma_1, ..., \sigma_n\}$ over domains $D_1, ..., D_n$. All these symbols can be used simultaneously to represent facts with unknown predicates, constants and truthfulness.

We consider the set of *constraints* Φ over the symbols Σ that include the standard logical connectives and equalities between symbols and constants from the corresponding domains. For example, the following is a valid constraint: $(\xi_1 \vee \neg\xi_2) \wedge (\alpha = \gamma) \wedge (\beta \neq 5)$. We denote the evaluation of the formula ϕ value under the interpretation $\{v_i \in D_i\}_{i\in[1..n]}$ as $\phi[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n]$.

We call the set of all facts expressible using given predicates, constants, and symbols the *symbolic Herbrand base*, denoted as SHB. We similarly define SEHB and SIHB. Given a symbolic fact $f \in$ SHB, we define its *concretisation* with the values $\{v_i \in D_i\}_{i\in[1..n]}$, denoted as $f[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n]$, as the fact obtained by replacing all symbols with their concrete counterparts. Given a symbolic EDB $\mathcal{E} \subset$ SEHB, we define its concretisation with the values $\{v_i \in D_i\}_{i\in[1..n]}$, denoted as $\mathcal{E}[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n]$, as the EDB obtained from $\mathcal{E}$ by concretising all symbolic facts, and removing all facts corresponding to symbolic signs with the negative interpretation.

We define the *semantics of symbolic execution* of a Datalog program P as a function $\mathfrak{S}_P : 2^{\mathrm{SEHB}} \rightarrow 2^{\mathrm{SIHB}\times\Phi}$. Given a symbolic EDB $\mathcal{E}$, this function returns a set of pairs of symbolic facts and logical constraints. We refer to these pairs as the set of *symbolic outputs* and the corresponding *inference conditions*. The latter resemble path conditions in conventional symbolic execution in that they specify under which condition the output is generated. Specifically, for each concretisation of $\mathcal{E}$ with $\{v_i \in D_i\}_{i\in[1..n]}$, the output of the query is identical to the set of all concretisations of the outputs of SEDL on $\mathcal{E}$ corresponding to the positive inference conditions:

$$\begin{aligned}\forall v_1 \in D_1, ..., v_n \in D_n.\ & \mathfrak{M}_P(\mathcal{E}[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n]) \\ &= \{\, f[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n] \\ &\quad \mid (f, \phi) \in \mathfrak{S}_P(\mathcal{E}) \wedge \phi[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n] \,\}\end{aligned}$$

The inference of a symbolic fact from a symbolic EDB can be visualised using a *symbolic proof tree*, a proof tree in which productions are annotated with logical constraints that enable that productions. Formally, for a given Datalog query Q, a symbolic EDB $\mathcal{E}$, and a symbolic fact F, a symbolic proof tree of F from $\mathcal{E}$ is the tree $(F, R, \phi, \{\theta_0, ..., \theta_n\}, \{t_1, ..., t_n\})$ such that $n = 0$ iff $F \in \mathcal{E}$, otherwise $R \triangleq \mathtt{L}_0$ `:-` $\mathtt{L}_1, ..., \mathtt{L}_n$ is a rule from Q, $F = \mathtt{L}_0\theta_0$, t_i are symbolic proof trees of $\mathtt{L}_i\theta_i$, and the substitutions $\{\theta_0, ..., \theta_n\}$ are identical under each satisfying assignment of ϕ. Apart from that, we are only interested in feasible symbolic proof trees, that is symbolic proof trees where all logical annotations are consistent with each other, since in this case the symbolic proof tree corresponds to at least one concrete proof tree for a concrete database.

To illustrate SEDL, consider the Datalog query in Figure 2b, and the symbolic database in Section 2.2. A symbolic proof tree of the fact `npe("y",3)` inferred by SEDL from this database is given in Figure 3. The inference condition for this fact is formed as the conjunction of all constraints in the symbolic proof tree:

$$\begin{aligned}&\xi_2 \wedge \xi_3 \wedge \alpha_1 = 1 \wedge \xi_4 \wedge \alpha_3 = 2 \\ &\wedge \neg(\xi_6 \wedge \alpha_5 = \text{“}x\text{”} \wedge \alpha_6 = \alpha_2) \wedge \neg(\xi_5 \wedge \alpha_4 = \text{“}y\text{”})\end{aligned}$$

3.3 Encoding a Neighbourhood of a Database

Although SEDL can execute an abstracted database that is fully symbolic, practical applications such as program repair described in Section 5 benefit from injecting symbols into an existing concrete database, thus considering a neighbourhood of this concrete database. Let E be an EDB; its *k-neighbourhood* $\mathcal{N}(E, k)$ represents the set of databases differed from the original one in at most k facts:

$$\mathcal{N}(E, k) \triangleq \{\, E' \mid E' \subseteq \mathrm{EHB} \wedge |E' \triangle E| < k \,\}$$

where $\triangle$ is symmetric difference and k is a user-defined parameter.

To represent a neighbourhood of concrete database using symbols, we augment the symbolic EDB with structural constraints. For example, the symbolic encoding of k-neighbourhood $\mathcal{N}(E,k)$ for an EDB E can be defined through a pair of a symbolic EDB $\mathcal{E}$ and constraints ψ such that

$$\mathcal{N}(E,k) = \{\, \mathcal{E}[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n] \\ \mid \psi[\sigma_1 \mapsto v_1, ..., \sigma_n \mapsto v_n] \wedge v_i \in D_i \,\}$$

To illustrate such encoding, consider the EDB

```
node(1).      flow(1, 2).
node(2).
```

Its 1-neighbourhood can be encoded as the symbolic EDB

```
node(1). assoc. with ξ1      flow(1, 2). assoc. with ξ4
node(2). assoc. with ξ2      flow(β, γ). assoc. with ξ5
node(α). assoc. with ξ3
```

and the structural constraints

$$\xi_3 \wedge \xi_5 \wedge \mathrm{atLeast}(2, \xi_1, \xi_2, \xi_4) \\ \vee\, \xi_1 \wedge \xi_2 \wedge \xi_4 \wedge \mathrm{atMost}(1, \xi_3, \xi_5)$$

where atLeast and atMost are cardinality constraints. In the context of program repair driven by Datalog analysis, Section 5 shows how to encode an analysis specific neighbourhood consisting of only syntactically valid programs.

4 REALISATION OF SEDL IN SYMLOG

Symlog employs a meta-programming approach to implementing SEDL. Specifically, a given Datalog query and a symbolic EDB are transformed into a *meta-query* and a *meta-EDB* in such a way that executing the meta-query on the meta-EDB with the standard Datalog semantics yields the output of symbolic execution of the original query on the original symbolic database. We collectively refer to the meta-query and the meta-EDB as the *meta-program*. The advantage of this approach is that it enables us to reuse existing efficient implementations of Datalog, and makes Symlog independent of the Datalog evaluation strategy.

4.1 Naïve Encoding of SEDL

We first describe a naïve implementation of SEDL. It explicitly enumerates all values from the domains of symbols inside the meta-program, and computes how each of them relates to the output of the query. To achieve this, we augment all predicates used in the query with auxiliary variables that store the assignment of symbolic constants to concrete values, which we refer to as *symbolic bindings*. To ensure that assignments of symbolic constants are consistent within a derivation of each output fact, as per the definition of symbolic proof trees in Section 3.2, we propagate the values of symbolic bindings across each rule.

Assume that a given symbolic EDB contains the symbolic constants $\alpha_1, ..., \alpha_n$. We introduce auxiliary Datalog variables $C_1, ..., C_n$ for symbolic bindings. For each rule, the meta-query adds $C_1, ..., C_n$ to the head and each literal of the body. For instance, we transform the first rule defining `null` in Figure 2b into

```
null(V, L, C1, ..., Cn) :-
    flow(L1, L, C1, ..., Cn),
    assign_null(V, L1, C1, ..., Cn).
```

To transform the EDB into a meta-EDB, the naïve approach enumerates all instantiations of symbolic constants with the values from the corresponding domains. For example, given the EDB fact `flow(`α_1`, 2)`, we transform it into the rule

```
flow(C1, 2, C1, ..., Cn) :-
  domain_alpha_1(C1), ...,
  domain_alpha_n(Cn).
```

where each predicate `domain_alpha_`i, called *symbolic domain predicate*, is true for all values from the domain of α_i. After the meta-program is executed, the resulting values of symbolic bindings capture the assignment of symbolic constants that enable the generation of corresponding output facts.

To support symbolic predicates, the naïve approach introduces auxiliary integer variables P_i called *predicate selectors* for each symbolic predicate to identify which of the concrete predicates from the corresponding domain is enabled. For each rule, the meta-query adds these variables to the head and each literal of the body. In the meta-EDB, we transform each fact with a symbolic predicate, say, ρ`(1,2)` where the domain of ρ is $\{\mathrm{p1}, \mathrm{p2}\}$, into the rules

```
p1(1, 2, P) :- P = 1.
p2(1, 2, P) :- P = 2.
```

After the meta-program is executed, the resulting values of predicate selectors indicate if output facts can be inferred by relying on various instantiations of the symbolic predicates.

To support symbolic signs, the naïve approach introduces auxiliary boolean variables S_i called *sign selectors* for each fact with a symbolic sign. To generate a meta-EDB, we transform each fact with a symbolic sign, say, F defined as ξ `p(1, 2)`, into the fact `p(1, 2, True)`, and each other fact with the same or a different predicate, say, `q(3, 4)`, into `q(3, 4, False)`. To generate a meta-query, we transform each rule by adding sign selectors to the head and each literal of the body, and also additional constraints that state that the head depends on F if at least one of the literals of the body depends on F. For example, given the rule `t(X, Y):- v(X), r(Y)`, we transform it into the rule

```
t(X, Y, S) :- v(X, S1), r(Y, S2), S = S1 ∨ S2.
```

After the meta-program is executed, the resulting values of sign selectors indicate if output facts can be inferred with or without relying on the fact with a symbolic sign.

By computing the values of auxiliary variables (symbolic bindings, predicate selectors and sign selectors), it is possible to reconstruct inference conditions for each output fact. However, since the domains of symbolic constants and the number of facts with symbolic signs can be huge, this approach does not scale to realistic databases, which motivated us to design optimisations to avoid explicit enumeration for these two categories of symbols.

4.2 Optimisation for Symbolic Constants

To address the search space explosion of the naïve approach, instead of enumerating all possible values of symbolic constants in the definition of symbolic domain predicates, we explicitly maintain abstract symbols during evaluation. For each symbolic constant α with the domain D defined by the symbolic domain predicate in the meta-program described in Section 4.1, we optimise the meta-program by compressing D in a way that does not change the output of the query. We separately handle two categories of constants in

$$
\begin{aligned}
&p(..., c_i, ...) \in \mathcal{E} \Rightarrow \mathrm{depend}(p, i, c_i)\\
&L :- ..., p(..., c_i, ...), ... \in Q \Rightarrow \mathrm{depend}(p, i, c_i)\\
&L :- ..., p_1(..., X_i, ...), ..., p_2(..., X_j, ...), ... \in Q, X_i \equiv X_j \Rightarrow\\
&\qquad \forall c.\, \mathrm{depend}(p_1, i, c) \Leftrightarrow \mathrm{depend}(p_2, j, c)
\end{aligned}
$$

Figure 5: Constraints defining the relation depend**: Q is the query, $\mathcal{E}$ is the symbolic EDB, c_i/X_i is a constant (symbolic or concrete)/a variable appearing as the i-th predicate parameter, $X_i \equiv X_j$ denotes that X_i and X_j are identical variables.**

the domain: those constants that already exist somewhere else in the program, and previously unseen constants.

To handle constants that already exist somewhere in the program, we consider all such constants c from D, which we refer to as *unifiable constants* of α, appearing either in the EDB or the query that there exists a symbolic proof tree whose production rule annotations contain a dependency between α and c, *e.g.* a constraint $\alpha = c$. If such dependency does not exist in any of the symbolic trees, we prune these constants from the symbolic domain predicate, since the lack of interactions with existing constants implies that these elements of the domain can be handled in the same way as previously unseen constants. However, identifying the exact set of unifiable constants is impractical, because that would require computing all symbolic proof trees in advance. Instead, we compute an overapproximation of this set by using the relation $\mathrm{depend}(p, i, c)$ that states that the i-th parameter of the predicate p may depend on the symbolic or concrete constant c. This relation can be computed using constraints defined in Figure 5 via a fixed point computation algorithm. For the symbolic constant α, we select all pairs $\{(p_j, i_j)\}$ such that $\mathrm{depend}(p_j, i_j, \alpha)$ for all j. Then, an overapproximation of the set of unifiable constants for α can be defined as

$$\mathrm{unifiable}(\alpha) \triangleq D \cap \bigcup_j \{\, c \mid \mathrm{depend}(p_j, i_j, c) \,\}.$$

To handle the elements of D that do not exist anywhere else in the program, which we refer to $D_{\mathrm{new}} \triangleq D \setminus \mathrm{unifiable}(\alpha)$, we introduce extra constants that serve as abstract representation of D_{new}. We also consider interactions between multiple symbolic constants. For example, assume that the symbolic EDB contains symbolic constants α and β. We introduce extra constants n_1 and n_2 to encode all partitionings of α and β into equivalence classes. Given the EDB fact `flow(α, 2)`, we transform it into the rules

```
flow(C1, 2, C1, C2) :- domain(C1, C2).
domain(X, Y) :-
  domain_alpha_unifiable(X),
  domain_beta_unifiable(Y).
domain(n1, Y) :- domain_beta_unifiable(Y).
domain(X, n1) :- domain_alpha_unifiable(X).
domain(n1, n1).
domain(n1, n2).
```

where `domain_alpha_unifiable` is the set of unifiable constants for α (the same for β), and `domain` is the optimised symbolic domain predicate. The first rule defining this predicate states that symbolic constants may take any of the values of corresponding unifiable constants. The second and the third rules state that one of them may take a new, previously unseen value. The last two rules state that both symbolic constants can take previously unseen values, either equal (rule four) or different (rule five). After such meta-program is executed, n_1 and n_2 can be replaced with α and β by adding constraints imposed by the corresponding equivalence classes. For example, the meta-program output fact `null("v", n1, n1, n2)` would correspond to `null("v", α)` under the condition

$$\alpha \notin \mathrm{unifiable}(\alpha) \wedge \beta \notin \mathrm{unifiable}(\beta) \wedge \alpha \neq \beta$$

4.3 Optimisation for Symbolic Signs

Supporting symbolic signs is challenging because there are $2^{|\mathrm{EDB}|}$ ways to negate facts in the database. To avoid a search space explosion, we generate inference conditions for only k selected outputs under the assumption that at most n of the symbolic facts are negative. This assumption holds in our application to program repair, since we only explore a neighbourhood of a given database, and program repair constraints only depend on the inference of a small number of output facts. For brevity, we only explain how to support symbolic signs for $k = n = 1$.

For a given fact with a symbolic sign I, the encoding in Section 4.1 enables us to determine if an output fact O depends on that fact (if it can be inferred with and without the presence that fact in the database). We say that I is a hard dependency of O, if O is only inferred when the sign of I is true. We formulate the problem of computing the inference condition of O in terms of symbolic signs as the problem of finding all hard dependencies of O among all inputs with symbolic signs, since removing any hard dependency from the database disables the generation of the output fact.

To identify all hard dependencies of a given output fact, we rely on Zeller's delta debugging algorithm [47] that operates by iteratively splitting the space of solutions and investigating each half individually. To apply delta-debugging, we define the function $\mathrm{interesting}_O$ that for a given set of input facts returns true if this set contains all hard dependencies of O. We also extend the encoding of symbolic signs in Section 4.1 so that sign selectors are applied not to individual facts, but to sets of facts, to identify if an output fact can be inferred without using any element from that set. Such an extended encoding enables us to implement the function $\mathrm{interesting}_O$, and thus apply the delta-debugging algorithm.

5 SYMLOGREPAIR AND ITS INSTANCES

In this section, we describe how SymlogRepair uses Symlog and an SMT solver to generate patches, and describe SymlogRepair instances: SymlogRepair[NPE, Java], SymlogRepair[Leak, Python], and SymlogRepair[Securify2, Solidity].

5.1 Architecture of SymlogRepair

The architecture of SymlogRepair is shown in Figure 1. It accepts two inputs: the source code of the buggy program, and an analysis defined in Datalog. It then executes the following steps:

(1) converts the program into a Datalog EDB, injects symbols into this EDB, and produces structural constraints ψ over the symbols that captures a set of syntactically valid programs;
(2) executes the analysis symbolically obtaining inference conditions ϕ for the analysis outputs manifesting violations.;
(3) constructs the repair condition $\neg\phi \wedge \psi$;

(4) solves the repair condition to produce minimal repairs, that is, repairs that minimally modify original values in the EDB;
(5) maps the obtained valuation of symbols back into the source code to generate a patch.

Our approach generates minimal repairs, because such repairs are less likely to break unspecified functionality of the program [30]. The first step, generating a repair search space, is analysis and programming language specific. Below, we describe how it is realised for the three analysis problems we consider in this work.

5.2 SymlogRepair for Java NPE

Digger's NPE analysis is built on top of a points-to analysis, which approximates the heap memory configuration at program points. Among other uses, this information can be used to detect if a pointer variable points to NULL when it is dereferenced.

The search space for fixing NPEs is defined through generic templates that check if dereferenced program variables equal NULL, and if so, perform various actions, such as not executing a fragment of code, returning a default value, such as the call to the default constructor of a given type, or perform an early exit. We select facts related to the above templates to symbolise and add symbolic signs to the facts that represent the function with the detected bug.

5.3 SymlogRepair for Python Notebook

Data leak analysis assesses the dependence between training and test data, which can lead to artificially optimistic results. Preprocessing data leakage happens when training and test data are transformed together by a function that may impute results based on both data sets, like normalization using both datasets' distributions.

To repair the bug, preprocessing should be removed from source data and applied separately to training and test data. For example, in Figure 6, the correct version uses `fit_transform` for training data and `transform` for testing data. We define the search space of repair by a generic template that moves preprocessing from source to training data and add corresponding preprocessing for test data. Symbolic constants and symbolic signs are added for preprocessing APIs and control flow graph components, respectively.

5.4 SymlogRepair for Smart Contracts

We address four smart contract vulnerabilities detected by Securify2 [2, 41]: access control, reentrancy, unhandled exceptions, and locked Ether.

There are four types of access control bugs. Transaction origin bugs emerge from using the outdated `tx.origin` for caller verification. The fix is to replace it with `msg.sender`. We symbolise `tx.origin` related facts. Suicidal bugs result from unchecked `selfdestruct()` calls. The fix is to ensure the caller is the contract's owner. We symbolise the facts of owner variable and control flow graph. Leaking and delegate call bugs are addressed similarly as the suicidal bug.

Reentrancy bugs occur when external contract can make new calls to the calling contract before the first invocation is finished. The repair is finalising all internal state changes before the call is executed. We symbolise control flow facts for such bugs. Unhandled exception bugs are caused by unchecked low-level call returns. The fix is to add a check for the returned value of the low level call. Locked ether bugs result from contracts with payable functions but no withdrawal functions. One can fix the bugs by removing the payable attribute, adding a withdraw function or adding a function with `selfdestruct`. We symbolise facts for `selfdestruct`.

Table 1: NPE10 dataset of NPE bugs from Java projects.

Program	Bugs	kLoC	Project description
jfreechart	1	132	A 2D chart library for Java
spoon	1	155	A library for Java code transformation
jackson-databind	1	142	A data-binding package for Jackson
jeveassets	1	101	An asset manager for Eve-Online
fastjson	1	186	A JSON processing library
karaf	1	128	A modulith runtime
acs-aem-commons	2	111	Components of AEM consulting practice
camel	1	1156	An integration framework
thirdeye	1	126	A tool for time series analysis

6 EVALUATION

In this section, we evaluate SymlogRepair by investigating the following research questions (RQs).

RQ-I: Ability to Repair a Diverse Class of Bugs. Can SymlogRepair fix a diverse bugs and how it compares to existing tools that target the same bugs?

RQ-II: Impact of Optimisations. How much do our optimisations improve the result compared to the naïve implementation of SEDL in Symlog?

6.1 Experimental Setup

Our experiments were performed on an Intel® Core™ Intel(R) Xeon(R) Gold 6258R CPU at 2.70GHz with 256GB of physical RAM running Ubuntu 20.04.1 LTS. We set timeout to 1 hour.

Datasets. We constructed three datasets. The first is **NPE10**, a dataset consisting of 10 NPE bugs in Java projects: five of them are from NPEX dataset [25]; two from BugSwarm [39], and three bugs that we systematically mined on GitHub; the criterion of including the bugs into NPE10 is that they are detected by Digger [35], the underlying component of SymlogRepair[NPE, Java]. The summary of this dataset is given in Table 1. The second is **PL11**, a dataset consisting of 11 preprocessing leakage bugs in Python notebooks collected by Yang *et al.* [45], which excludes bugs not detectable by the used analyser and false positives. Finally, **SC63** is a dataset composed of 63 bugs in Solidity smart contracts. It includes 22 bugs from Smartbugs [14] and 41 bugs from ScrawlID [46]. SC63 excludes bugs from Smartbugs and from ScrawlID that Securify2 [41] cannot detect. As ScrawlID does not provide ground truth, we manually annotated the selected bugs with correct repairs.

Tools. When repairing Java NPE bugs, we considered two configurations of SymlogRepair[NPE, Java]: (1) **SRNJN** using the naïve algorithm defined in Section 4.1 and (2) **SRNJO** using the optimisations described in Section 4.2 and Section 4.3. We considered three baselines for NPE bugs: NPEX [25] that is specifically designed for NPE bugs, and a general machine learning based tool AlphaRepair [44], and InCoder [16], a code model that demonstrated state-of-the-art results for program repair [19]. When repairing Python notebook data leaks, we considered two configurations

Table 2: Patches generated by SymlogRepair[NPE, Java], NPEX, AlphaRepair and InCoder for NPE10: ● indicates correct patch, ◐ — property-overfitting patch, ○ — no patch found, OOM — out of memory, "Symbols" show the num. of symbolic constants + the num. of symbolic signs. The optimised SymlogRepair fixed more bugs than NPEX, AlphaRepair and InCoder, and outperformed the non-optimised version by avoiding OOMs.

Project	Version	Symbols	Patch					Time		Memory (GB)	
			NPEX	AlphaRepair	InCoder	SRNJN	SRNJO	SRNJN	SRNJO	SRNJN	SRNJO
jfreechart	2182	5+8303	●	●	○	●	●	3m 53s	2m 54s	6.245	1.322
spoon	b3f568d	5+8024	○	●	○	●	●	3m 6s	2m 7s	6.547	1.330
jackson-databind	974ccdd	8+531	●	◐	○	●	●	2m 44s	2m 36s	3.587	2.643
jeveassets	f35ccd9	5+2198	●	○	○	●	●	2m 16s	2m 11s	10.099	10.098
camel	597883f	5+40768	●	○	○	●	●	11m 29s	8m 30s	23.688	6.173
thirdeye	e286991	5+54	●	○	○	●	●	1m 20s	1m 21s	2.203	2.197
fastjson	7c05c6f	8+5739	○	○	○	○	◐	-	2m 43s	OOM	2.643
karaf	5965290	8+573	◐	◐	○	◐	◐	10m 32s	4m 28s	21.881	22.047
acs-aem-commons	374231978	13+14277	●	○	○	○	●	-	5m 47s	OOM	6.344
acs-aem-commons	374231969	13+14206	●	○	○	○	●	-	5m 46s	OOM	6.408
Overall		7.5+5737.3	7+1	2+2	0+0	6+1	8+2	>3m 50s	3m 50s	>9.788	6.121

SymlogRepair[Leak, Python]: (1) **SRPLN** using the naïve algorithm and (2) **SRPLO** using the optimisations. We also selected AlphaRepair and InCoder as baselines for Python bugs. For repairing bugs in Solidity smart contracts, we considered two configurations of SymlogRepair[Securify2, Solidity]: (1) **SRSSN** using the naïve algorithm and (2) **SRSSO** using the optimisations. As the baseline, we selected the state-of-the-art smart contract repair tool Elysium [15].

Correctness criteria. Patches that satisfy the analysis property we classify as *plausible* (contrary to test-driven repair, where plausible patches refer to patches that pass given tests). A patch is *correct* if it is semantically equivalent to the developer patch. We manually investigated generated patches to check their correctness. In the manual analysis, we conservatively assumed that the correct patch must be syntactically identical to the developer patch subject to trivial refactorings. Those patches that are plausible, but not correct, in analogy with test-overfitting [36], we call *property-overfitting*.

6.2 Ability to Repair a Diverse Class of Bugs

To investigate the ability to handle a diverse class of bugs, we evaluated SymlogRepair[NPE, Java] on NPE10 dataset of Java NPE bugs, SymlogRepair[Leak, Python] on PL11 dataset of Python data leaks, and SymlogRepair[Securify2, Solidity] on SC63 dataset of Solidity smart contracts bugs.

Table 2 summarises the results of our experiments on NPE10. In this table, the columns "Patch" show the number of correct and property-overfitting patches for each configuration. SRNJO is the default (optimised) configuration of SymlogRepair[NPE, Java]. SRNJO generated patches for all 10 bugs, and 8 of them were repaired correctly, and for 2 the tool generated overfitting patches. We compared SRNJO with NPEX, AlphaRepair and InCoder. For AlphaRepair and InCoder we used the Datalog analyser for validating candidate patches. The experiments demonstrated that SRNJO correctly repaired one more bugs than NPEX, 6 more bugs compared with AlphaRepair, and 8 more than InCoder.

Table 3 summarises the results of our experiments on PL11. As in the previous experiment, the columns "Patch" show the number of correct and property-overfitting patches generated by each configuration. For PL11, developer patches are not available, therefore we classify generated patches into correct or plausible via a manual inspection and cross-checking. SRPLO is the default (optimised) configuration of the SymlogRepair[Leak, Python]. SRPLO repaired 6 out of 11 bugs correctly. Among the remaining 5 bugs, for 4 bugs it generated patches that overfit the property, and one was not repaired due to timeout. AlphaRepair and InCoder failed to generate any patches that satisfy the analysis property.

Table 4 summarises the results of our experiments on SC63. The columns "Fixed" show the numbers of fixed bugs by the state-of-the-art smart contract repair tool Elysium [15] and the default configuration of SymlogRepair[Securify2, Solidity], SRSCO. "Time" shows the average time taken by SymlogRepair for generating patches. The experiments demonstrated that our tool generated 47 more correct patches than Elysium. The non-optimised version, SRSSN, failed to generate any patches because of OOM errors.

RQ-I: SymlogRepair demonstrated its ability to statically repair a diverse class of realistic bugs across multiple programming languages, where it demonstrated results close to or better than the state-of-the-art.

6.3 Impact of Optimisations

Symbolic execution of Datalog requires additional memory, because it has to explore a large space of possible relations, which poses a scalability challenge. To improve scalability, we introduced optimisations. To investigate the impact of these optimisations, we compared the performance of the naïve (cf. Section 4.1) and optimised (cf. Section 4.2) configurations of SymlogRepair in terms of time and memory usage.

Table 2 and Table 3 provide detailed statistics of executing SRNJN, SRNJO, SRPLN and SRPLO on the corresponding datasets. The columns "Symbols" show how many symbols (symbolic constants + symbolic signs) were automatically injected into the EDB to generate a repair search space. The columns "Time" show total elapsed time, and "Memory" show the peak memory usage or OOM if an out of memory error occurred. The results in Table 2 show that 80% of the time our optimisation exhibits speedups of up to 2.5× (ignoring unfinished benchmarks). In terms of memory usage, our

Table 3: Patches generated by SymlogRepair[Leak, Python] for PL11. The notation is identical to Table 2. The optimisations enabled SymlogRepair to repair multiple bugs by avoiding OOMs.

Notebook	Symbols	Patch				Time		Memory (MB)	
		AlphaRepair	InCoder	SRPLN	SRPLO	SRPLN	SRPLO	SRPLN	SRPLO
nb_2528	6+22584	○	○	○	●	-	23m 50s	OOM	91.049
nb_2816	6+11975	○	○	○	●	-	30m 46s	OOM	81.211
nb_1417	6+17659	○	○	○	●	-	10m 58s	OOM	70.569
nb_949	6+95652	○	○	○	●	-	47m 24s	OOM	393.623
nb_2760	6+14941	○	○	○	◐	-	18m 51s	OOM	121.021
nb_3144	6+247561	○	○	○	○	-	Timeout	OOM	181.112
nb_318	6+49748	○	○	○	◐	-	27m 59s	OOM	191.175
nb_296	6+493152	○	○	○	◐	-	25m 35s	OOM	494.718
nb_2646	6+5196	○	○	○	◐	-	35m 20s	OOM	60.447
nb_195	6+79617	○	○	○	●	-	12m 44s	OOM	429.864
nb_3594	6+21238	○	○	○	●	-	13m 36s	OOM	80.828
Overall	6+96610.5	0+0	0+0	0+0	6+4	-	>24m 7s	-	199.328

Table 4: Patches generated by Elysium and SymlogRepair[Securify2, Solidity] for SC63. The notation is identical to Table 2.

Bug Type	Bugs	Fixed (Correct+Overfitting)		Time
		Elysium	SRSSO	
Access control	5	3+0	4+1	3m 39s
Unhandled exception	13	4+0	13+0	4.32s
Reentrancy	8	8+0	8+0	1m 47s
Locked ether	41	0+0	41+0	1m 34s
Overall	63	15+0	62+1	< 4m

optimisation exhibits a reduction 80% of the time of up to a 5× (ignoring cases that ran out of memory). In the case of results in Table 3, the naïve version runs out of memory on all benchmarks. On the other hand, the optimised version was able to repair all but a single benchmark. Programs in Table 2 are significantly larger, but their repair search space, as shown in the "Symbols" column, is smaller due to localisation described in Section 5.2, which explains the difference in time and memory usage across these benchmarks. In the experiments on SC63, the non-optimised version of SymlogRepair failed to generate any patches because of OOM errors, while the optimised version fixed 62 bugs correctly, while using less than 16GB of memory on average, and less than 4 minutes on average.

RQ-II Our optimisations play a crucial role in ensuring the scalability of SymlogRepair, as they on average reduce the repair time and memory usage while enabling the generation of more patches by preventing OOM exceptions.

7 DISCUSSION

The use of program properties defined in Datalog enables our approach to avoid test-overfitting intrinsic to test-driven program repair [22, 36], but our approach may overfit the property. First, static analysis properties are designed to avoid undesirable behaviour but not to specify the intended behaviour. Second, static analysis is subject to false positives. Third, some analysis may not be able to recognise a patch as correct due to the use of over-approximations.

```
from sklearn.preprocessing import MinMaxScaler

dataset = load_data()
scaler = MinMaxScaler(feature_range=(0, 1))
- scale_data=scaler.fit_transform(dataset)

- train_data, test_data = split_data(scale_data)
+ train_data, test_data = split_data(dataset)

x_train, y_train = split_train_data(train_data)
x_test, y_test = split_test_data(test_data)
model = LSTM_model()

+ x_train_new = scaler.fit_transform(x_train)
+ x_test_new = scaler.transform(x_test)

- model.fit(x_train, y_train)
+ model.fit(x_train_new, y_train)
- predictions = model.predict(x_test)
+ predictions = model.predict(x_test_new)
```

Figure 6: Patch for preprocessing leakage bug in nb_2528.

Thus, patches generated by our approach still need to be reviewed by developers and/or prioritised by AI. We will investigate the problem of prioritising statically generated patches in future work.

Our method of symbolically executing standard Datalog, however many practical applications of Datalog rely on extensions such as stratified negation and extra-logical operations. In future work, we plan to investigate how to support various Datalog extensions.

To support a new class of bugs in SymlogRepair, it is necessary to provide a Datalog program for detecting this class of bugs, and define symbolic search space for fixing the bugs. In this work, we used existing research analysers for Java, Python and Solidity. Commercial Datalog-based analysers like Semmle [1] provide analyses for more programming languages.

We manually wrote three ground truth patches from NPE10, as well as all the patches from PL11 and SC63. Due to the nuances of domain-specific knowledge, the provided solutions may not capture all possible intricacies or be optimal.

In this work, we only investigate the application of SEDL to program repair. However, we believe SEDL can be applied to many domains, so as conventional symbolic execution. In future work, we will investigate other applications of SEDL: data integration [26], repairing networks [28], and database testing.

8 RELATED WORK

Our work contributes to automated program repair [17]; it is relevant to static program repair and repair based on symbolic execution. SEDL is relevant to Datalog extensions and Datalog debugging.

Static Program Repair. Program repair systems based on static analysis fix specific types of bugs, such as memory leaks, null pointer exceptions (NPEs), data races, or locking errors. MemFix [24] addresses the memory leak problem by reducing it to an exact cover problem and using a SAT solver to find a solution. Saver [18] uses object flow graphs to patch memory errors in C programs. NPEX [25] applies a custom symbolic execution technique to repair NPEs in Java programs, but relies on stack traces to detect bugs. Footpatch [42] uses separation logic to fix memory leaks and NPEs. Hippodrome [9] repairs data races in Java programs with the aid of RacerD's static analysis [7]. Crayons [10] synthesises patches for locking API misuses using graph coloring and ranking, and estimates the criticality of code in static analyser error traces. We did not compare SymlogRepair with techniques other than NPEX, because they handle different defect classes and use an ad-hoc integration of static analysis and repair. Rather than developing new Datalog-based analyzers for these defects, which is outside this work's scope, we showed our method's generality by integrating existing Datalog-based analyzers for various bugs across multiple languages.

Several work employs static analysers to discover repair patterns. Phoenix [5] mines repair patterns from examples of bug fixes reported by static analysers and abstracts patterns by clustering similar edit examples through the use of a domain-specific language. Avatar [4] infers fix patterns from the static analysis violations detected by FindBugs. Our approach does not learn repair patterns, but instead provides repair strategies through the use of Datalog, which enables it to address a wider range of bugs.

Program Repair with Symbolic Execution. Our work is relevant to semantic program repair such as SemFix [32], Angelix [31] and symbolic execution with existential second-order constraints (SE-ESOC) [29]. These approaches also abstract a given program by injecting symbols, and execute the program symbolically to infer repair constraints. The key difference of our work is that instead of inferring constraints from test, we infer them from a Datalog analyser, thus taking the analysed property into account.

Program Repair Benchmarks. We constructed datasets specifically for the defect classes handled by the Datalog analysers, instead of relying on Defects4J [21] and QuixBugs [27] often used for evaluating test-driven repair. The new datasets better reflect the intended usage scenario of our approach, *i.e.* repairing bugs detected by static analysis, as opposite to repairing bugs found by testing. Bugs found by static analysis to universal program properties, *e.g.* the lack of crashes and security vulnerabilities, rather than project-specific requirements, and such bugs are not accompanied with tests.

ML-based Program Repair. ML-based program repair recently demonstrated promising results [19, 43]. We used two state-of-the-art tools, AlphaRepair [44] and InCoder [16], as baselines in our evaluation. In future work, we will investigate how to augment ML-based repair with semantic information from Symlog to improve their effectiveness in repairing static analysis bugs.

Datalog Input Repair. Database input repair has been studied in the context of integrity constraints violations that are limited to fragments of first order logic [3]. Theoretical solutions for positive Datalog using abduction are proposed [33]. However, traditional database methods often don't scale well for larger static analysis rulesets [51]. Zhao et al.'s technique [51] uses proof annotations in Datalog for enhanced scalability, but it mainly restricts to non-symbolic domains and doesn't execute repairs. Elastic incrementalisation extends this by incorporating provenance annotations for input adjustments in incremental Datalog [49, 50]. Xin et al.'s method [48] uses counter-example abstract refinement to choose the optimal EDB subset modeling an abstraction in program analysis. SynNet [12] generates candidate inputs for rules in network synthesis. To our knowledge, our work is the first repair technique that scales to large rulesets and inputs in Datalog-based program analyses.

Symbolic Datalog. Standard Datalog is limited to the domain of powersets. Recently, Datalog engines have integrated symbolic reasoning. Formulog [6] is a Datalog engine that uses SMT solving in the presence of symbolic variables in the ruleset. Modus [40] is a non-recursive Datalog engine that allows non-grounded variables to solve the container package dependency problem. Unlike these approaches, Symlog executes symbolic variables by encoding them in the standard Datalog setup thus existing optimised Datalog engines can be used. Based on the inferred constraints SimlogRepair can provide a patch that transforms the EDB to a correct version based on inferred constraints and an SMT solver. Thus, our technique can seamlessly plug into standard Datalog-based static analysis setups.

9 CONCLUSION

We introduced a new static program repair architecture that, by relying on Datalog, is able to repair a wide range of defects for various programming languages. Its core enabling component is symbolic execution of Datalog, which computes dependencies between the input and the output of a query, and repairs the database based on the desired output specification. We implemented this technique in an efficient tool Symlog, and applied to program repair based on Datalog-defined program analysis in the tool SymlogRepair. Our experiments demonstrated that SymlogRepair scales to real-world programs, and repairs a wide range of defects, including NPE bugs in Java programs, and data leaks in Python notebooks.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback which helped improve this paper. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

REFERENCES

[1] 2023. CodeQL (formely Semmle). https://codeql.github.com/. Accessed: 2023-06-29.

[2] 2023. Securify 2.0 security scanner for Ethereum smart contracts. https://github.com/eth-sri/securify2. Accessed: 2023-06-29.

[3] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 1999. Consistent Query Answers in Inconsistent Databases. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Philadelphia, Pennsylvania, USA) *(PODS '99)*. Association for Computing Machinery, New York, NY, USA, 68–79. https://doi.org/10.1145/303976.303983

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[5] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 613–624.

[6] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-Based Static Analysis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 141 (nov 2020), 31 pages. https://doi.org/10.1145/3428209

[7] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–28.

[8] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.

[9] Andreea Costea, Abhishek Tiwari, Sigmund Chianasta, Abhik Roychoudhury, Ilya Sergey, et al. 2021. HIPPODROME: Data Race Repair using Static Analysis Summaries. *arXiv preprint arXiv:2108.02490* (2021).

[10] Alfredo Cruz, Mahsa Varshosaz, Claire Le Goues, and Andrzej Wasowski. 2022. Patching Locking Bugs Statically with Crayons. *ACM Transactions on Software Engineering and Methodology* (2022).

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[12] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2017. Network-Wide Configuration Synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 261–281. https://doi.org/10.1007/978-3-319-63390-9_14

[13] David Evans. 1996. Static Detection of Dynamic Memory Errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 44–53. https://doi.org/10.1145/231379.231389

[14] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1349–1352.

[15] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2022. Elysium: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. 115–128.

[16] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[17] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[18] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: scalable, precise, and safe memory-error repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 271–283.

[19] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).

[20] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[21] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[22] Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. 163–163.

[23] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[24] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. Memfix: static analysis-based repair of memory deallocation errors for c. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 95–106.

[25] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2022. NPEX: Repairing Java Null Pointer Exceptions without Tests. (2022).

[26] Maurizio Lenzerini. 2002. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 233–246.

[27] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[28] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 97–108.

[29] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 389–399.

[30] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 448–458.

[31] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[32] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[33] Babak Salimi and Leopoldo E. Bertossi. 2016. Causes for Query Answers from Databases, Datalog Abduction and View-Updates: The Presence of Integrity Constraints. In *The Florida AI Research Society*.

[34] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. 196–206.

[35] Jixiang Shen, Xi Wu, Neville Grech, Bernhard Scholz, and Yannis Smaragdakis. 2020. Explaining bug provenance with trace witnesses. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP@PLDI 2020, London, UK, June 15, 2020*, Paddy Krishnan and Christoph Reichenbach (Eds.). ACM, 14–19. https://doi.org/10.1145/3394451.3397206

[36] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 532–543.

[37] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan D. Fekete, and Bernhard Scholz. 2018. Automatic Index Selection for Large-Scale Datalog Computation. *Proc. VLDB Endow.* 12, 2 (2018), 141–153. https://doi.org/10.14778/3282495.3282500

[38] Pavle Subotic, Lazar Milikic, and Milan Stojic. 2022. A Static Analysis Framework for Data Science Notebooks. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 13–22. https://doi.org/10.1109/ICSE-SEIP55303.2022.9794067

[39] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 339–349.

[40] Chris Tomy, Tingmao Wang, Earl T. Barr, and Sergey Mechtaev. 2022. Modus: a Datalog dialect for building container images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 595–606. https://doi.org/10.1145/3540250.3549133

[41] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[42] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*. 151–162.

[43] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery.*

[44] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 959–971.

[45] Chenyang Yang, Rachel A Brower-Sinning, Grace A Lewis, and Christian Kästner. 2022. Data leakage in notebooks: Static detection and better processes. In *Proceedings of the 2022 37th IEEE/ACM International Conference on Automated Software Engineering.*

[46] Chavhan Sujeet Yashavant, Saurabh Kumar, and Amey Karkare. 2022. Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. *arXiv preprint arXiv:2202.11409* (2022).

[47] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[48] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. 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, 239–248. https://doi.org/10.1145/2594291.2594327

[49] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, Niccolò Veltri, Nick Benton, and Silvia Ghilezan (Eds.). ACM, 20:1–20:16. https://doi.org/10.1145/3479394.3479415

[50] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2023. Automatic Rollback Suggestions for Incremental Datalog Evaluation. In *Practical Aspects of Declarative Languages - 25th International Symposium, PADL 2023, Boston, MA, USA, January 16-17, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13880)*, Michael Hanus and Daniela Inclezan (Eds.). Springer, 295–312. https://doi.org/10.1007/978-3-031-24841-2_19

[51] David Zhao, Pavle Subotic, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 7:1–7:35. https://doi.org/10.1145/3379446