SEAL: Integrating Program Analysis and Repository Mining

FLORIAN SATTLER and SEBASTIAN BÖHM, Saarland University, Germany
PHILIPP DOMINIK SCHUBERT, Paderborn University, Germany
NORBERT SIEGMUND, Leipzig University, Germany
SVEN APEL, Saarland University, Germany

Software projects are complex technical and organizational systems involving large numbers of artifacts and developers. To understand and tame software complexity, a wide variety of *program analysis* techniques have been developed for bug detection, program comprehension, verification, and more. At the same time, *repository mining* techniques aim at obtaining insights into the inner socio-technical workings of software projects at a larger scale. While both program analysis and repository mining have been successful on their own, they are largely isolated, which leaves considerable potential for synergies untapped. We present SEAL, the first *integrated* approach that combines low-level program analysis with high-level repository information. SEAL maps repository information, mined from the development history of a project, onto a low-level intermediate program representation, making it available for state-of-the-art program analysis. SEAL's integrated approach allows us to efficiently address software engineering problems that span multiple levels of abstraction, from low-level data flow to high-level organizational information. To demonstrate its merits and practicality, we use SEAL to determine which code changes modify central parts of a given software project, how authors interact (indirectly) with each other through code, and we demonstrate that putting static analysis' results into a socio-technical context improves their expressiveness and interpretability.

CCS Concepts: • **Software and its engineering** → **Software configuration management and version control systems**; **Collaboration in software development**; Software verification and validation; *Automated static analysis;*

Additional Key Words and Phrases: Static program analysis, software repository mining, socio-technical software analytics

ACM Reference format:
Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. 2023. SEAL: Integrating Program Analysis and Repository Mining. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 121 (July 2023), 34 pages.
https://doi.org/10.1145/3585008

This work was partially supported by the Heinz Nixdorf Foundation and the German Research Foundation (DFG) within the Collaborative Research Center 901 "On-The-Fly Computing" (grant no. 160364472) and Collaborative Research Center TRR 248 "Perspicuous Computing" (grant no. 389792660), project "Pervolution" (grant no. AP 206/11-1, AP 206/11-2, SI 2171/2-1, and SI 2171/2-2), project "Green Configuration" (grant no. SI 2171/3-1), and project "Congruence" (grant no. AP 206/14-1).
Authors' addresses: F. Sattler, S. Böhm, and S. Apel, Saarland University, Campus E1 1, Saarbrücken, 66123, Germany; emails: sattlerf@cs.uni-saarland.de, boehmseb@cs.uni-saarland.de, apel@cs.uni-saarland.de; P. D. Schubert, Paderborn University, Warburger Str. 100, Paderborn, 33098, Germany; email: philipp.schubert@upb.de; N. Siegmund, Leipzig University, Augustusplatz 10, Leipzig, 04109, Germany; email: norbert.siegmund@uni-leipzig.de.

1049-331X/2023/07-ART121 $15.00
https://doi.org/10.1145/3585008

1 INTRODUCTION

Software systems are among the most complex human-made systems today. To understand the inner workings and external qualities of complex software systems, researchers and developers have devised a variety of *program analysis* techniques to extract relevant information, including bug finders, program verifiers, and code metric tools.

The rise of open-source software has triggered the development of *repository mining* techniques that extract organizational information from software repositories. For example, researchers have investigated how code changes evolve on different platforms [59] or analyzed the characteristics of uncompilable code [21]. Other approaches analyze the socio-technical interaction around a software project to better understand how developers collaborate [7, 24–26].

Such *high-level* repository mining approaches analyze code typically at file, textual, or syntactical level and thus miss important information that is only encoded in the underlying program semantics. Information on the program semantics is often only accessible by *low-level* static program analysis techniques such as data-flow analyses. While high-level repository analyses are sufficient to get good first results, information that is hidden in the program's semantics, e.g., information on data-flow, is overlooked. This insufficiency precludes important use cases such as accounting bugs and security vulnerabilities directly to the commits/developers that introduced or know best how to fix them. For illustration, let us consider the example in Figure 1: Alice commits a change c_1, later Bob also changes the same file with commit c_2. With a purely textual or syntactical repository analysis, no connection between the two changes can be inferred, except that both change the same file. By mapping the change to a control-flow graph, we can leverage powerful program analysis techniques to detect previously hidden data dependencies between the changes and, in turn, infer that c_2 could be related to a bug in c_1.

Our central hypothesis is that a combination of low-level program analysis (in particular, data-flow analysis) with high-level repository information empowers us to answer practically- and scientifically-relevant questions that neither can answer alone. Many questions are difficult to answer solely based on data from a high-level repository analysis, for example, "Which developers are affected by a change?" or "Which portion of a code base is influenced by a particular change?" The underlying problem is that repository analyses often do not have precise data-flow information at their disposal, if at all, or try to include this information in an ad hoc manner, making it difficult to apply them to a wide range of practical settings. Conversely, state-of-the-art program analysis techniques and tools do not have organizational information (e.g., about versions or developers) at their disposal. This problem is hard to solve by just combining different tools, because the high-level information needs to map correctly to the program operational semantics and a fairly low-level program representation. We seek to *integrate* high-level repository information with low-level information in a principled way and make the *combined information* available for interpretation.

For this purpose, we have developed SEAL, a parameterizable approach that combines change information from a project's history with information on the program semantics computed by an inter-procedural, flow- and context-sensitive data-flow analysis. SEAL's integrated approach allows us to *precisely* analyze data-flow interactions between commits—a previously infeasible endeavor. More generally, SEAL establishes a mapping from a high-level information source (in our case, commit information) to an intermediate program representation that is suited for writing precise, low-level program analyses, conceptually decoupling the information source from the actual analysis. SEAL defines a general and parameterizable *relation* between data flows and high-level information and allows us, for the first time, to embed statements on the program semantics into the socio-technical context of a software project.

We implemented SEAL on top of LLVM [35] and PhASAR [51] in the form of a parameterizable framework that adds commit information to the compiler's intermediate representation (LLVM IR)

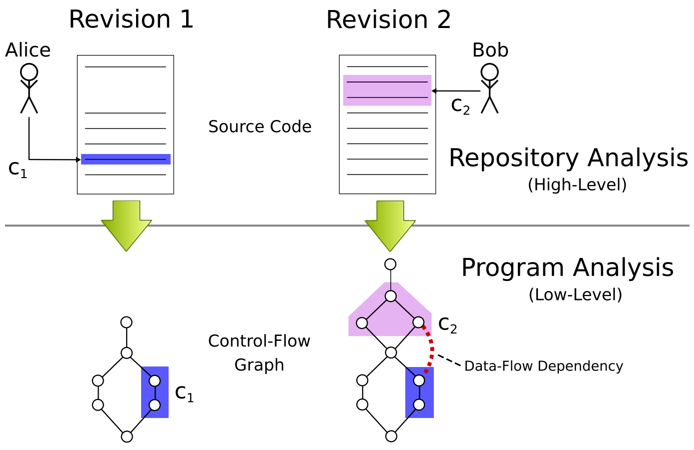


Fig. 1. Combining low-level program analysis with high-level repository analyses: Alice makes a change c_1 to a source code file creating revision 1. Later, Bob modifies the same file at a different location with c_2 creating revision 2. A typical repository analysis cannot find a connection between c_1 and c_2, except that both modified the same, possibly very large, file. If we map the changes of Alice and Bob onto the corresponding control-flow graph and run a data-flow analysis, then we discover a data-flow dependency (red) from c_2 to c_1.

and combines it with an inter-procedural data-flow analysis to determine which commits interact with each other at the level of data-flow. We designed SEAL to be modular and reusable to enable the commit information to be used by different static program analyses that target LLVM IR.

By means of a divers set of 13 open-source projects, we demonstrate the practicality of SEAL. Specifically, we apply it to four relevant software analysis problems and demonstrate how a combination of commit and data-flow information can be leveraged to solve these problems: (1) We use SEAL to detect potentially impactful changes that modify central code (i.e., code that interacts with lots of other code) in a software project. (2) We demonstrate the applicability of SEAL to socio-technical analyses by analyzing the socio-technical structure that arises from interactions among authors via commits. (3) We apply our approach to analyze which authors could be affected by a change. (4) Furthermore, we discuss how data generated by SEAL can be used to enrich existing data-flow analyses by making them change-aware and putting their results into a socio-technical context without paying the full costs for follow-up analyses.

In summary, we make the following contributions:

- A novel approach, called SEAL, that combines high-level repository mining and precise low-level program analysis.
- An open-source implementation of SEAL built on top of LLVM [35] and PhASAR [51] as well as a modified implementation of CLANG that allows one to inject repository change information during compilation, which is integrated with PhASAR, allowing for precise data-flow analyses.
- An evaluation consisting of four different studies, three that demonstrate the applicability of SEAL, evaluated on 13 open-source projects, and a showcase of how SEAL can be used to make existing data-flow analyses change-aware.

All results and a replication package are publicly available.[1] Our implementation and additional evaluation tools are open source and available on our project website.[2]

[1]Supplementary Website: https://se-sic.github.io/paper-SEAL/ and on Zenodo: https://doi.org/10.5281/zenodo.7595363.
[2]https://vara.readthedocs.io/en/vara-dev/.

```
int main() {                                                        ▷ 3e8882e
  int i = 20;                                                       ▷ ea8426c
  int j = 22;                                                       ▷ c4d9b1a
  return i + j;                                                     ▷ 0872f49
}                                                                   ▷ 3e8882e
```

(a) Running example C program. The column on the right-hand side shows the hashes of the commits that modified the line last.

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4, !Commit !3                         ▷ 3e8882e
  %i = alloca i32, align 4, !Commit !9                              ▷ ea8426c
  %j = alloca i32, align 4, !Commit !5                              ▷ c4d9b1a
  store i32 0, i32* %retval, align 4, !Commit !3                    ▷ 3e8882e
  store i32 20, i32* %i, align 4, !Commit !9                        ▷ ea8426c
  store i32 22, i32* %j, align 4, !Commit !5                        ▷ c4d9b1a
  %0 = load i32, i32* %i, align 4, !Commit !7                       ▷ 0872f49
  %1 = load i32, i32* %j, align 4, !Commit !7                       ▷ 0872f49
  %add = add nsw i32 %0, %1, !Commit !7                             ▷ 0872f49
  ret i32 %add, !Commit !7                                          ▷ 0872f49
}
```

(b) LLVM-IR representation of the running example of Figure (a). For simplicity, we depict the meta-data nodes that contain the commit information as triangles (▷) on the right-hand side.

Fig. 2. Running example: commit information at source-code level and its mapping to the IR.

2 SEAL AT A GLANCE

In this section, we provide an overview of SEAL and show how mapping repository information onto a compiler IR enables SEAL to combine high-level data with a specialized data-flow analysis. We define the concept of commit interactions, an abstraction that merges both kinds of information and enables us to reason about how data flows connect seemingly unrelated commits. Then, we describe in detail how SEAL can compute these commit interactions using the mapped repository information and a static taint analysis.

2.1 Code Annotation

In a preparation step, we map information from the version control system into a representation on which we later conduct the program analysis. As program representation, we use the compiler's *intermediate representation (IR)*, which is a common abstraction used in modern compilers and static analysis tools. It is important to note that SEAL and the definitions in Section 2.2 are based on, but not restricted to, a given IR. A key mechanism of our approach is that, during the construction of the IR, we add information to the specific IR instructions that relate them to the commit that introduced the corresponding code. The compiler determines the last change for each source-code line by accessing repository meta-data (e.g., git-blame[3]) and then annotates the commit hash to the respective instruction. Each commit itself is a snapshot of the repository that represents the changes compared to the previous commit.

Figure 2(a) lists a C program that serves as our running example. The hash of the commit that introduced each source-code line is shown on the right (▷). During compilation, SEAL adds this information to the IR, as shown in Figure 2(b). Each of the IR instructions is annotated by its corresponding commit hash (right side).

[3]Git-blame is a versioning system mechanism that annotates each line in a file with the commit that last modified it.

2.2 Commit Interactions

We devise a formal framework that serves as a basis for SEAL. In particular, the framework defines the abstract structure of commit interactions and commit-interaction paths, based on a given relationship between program elements, data flows, in our case.

A program $\mathcal{P}$ is a composition of instructions $i_1, \ldots, i_n \in \mathcal{P}$ stemming from a sequence of commits $c_1, \ldots, c_m \in C$, where C is the set of all commits from the repository.

Definition 1. Let base be a function that maps an IR instruction $i \in \mathcal{P}$ to the corresponding *base commit* $c \in C$, which gets attached during compilation.

The base commit represents the initial commit that introduced the source line in question to the code base, as illustrated on the right-hand side of Figure 2(b).

Definition 2. Function $\mathsf{instructions}$ computes the set of all instructions added by commit c (i.e., the set of instructions whose base commit is c).

$$\mathsf{instructions}(c) = \{\, i \mid \mathsf{base}(i) = c \;\wedge\; i \in \mathcal{P} \,\}$$

Applied to our example of Figure 2(b), $\mathsf{instructions}$ returns for commit `3e8882e` a set containing the two instructions **`alloca`** in Line 3 and **`store`** in Line 6.

Next, we define interactions between commits. We define the relation $\rightsquigarrow$ (relates to) to represent data-flow interactions. In general, our framework abstracts from any concrete relationship and does not require specific properties of the relation $\rightsquigarrow$. However, in a concrete instantiation, certain properties can be defined by the user to express desired analysis semantics. In our case, we require $\rightsquigarrow$ to be transitive and not symmetric, because the effects of data-flow interactions are directional, meaning, a write to a variable does only affect subsequent reads and not preceding ones.

Definition 3. For two commits $c_1, c_2 \in C$, $\mathsf{interactions}$ computes all tuples of instructions related by $\rightsquigarrow$ that belong to c_1 and c_2.

$$\mathsf{interactions}(c_1, c_2) = \Big\{ (i_1, i_2) \mid i_1 \rightsquigarrow i_2 \;\wedge\; i_1 \in \mathsf{instructions}(c_1) \\ \wedge\; i_2 \in \mathsf{instructions}(c_2) \Big\}$$

Function $\mathsf{interactions}$ applied to `3e8882e` and `0872f49`, for our running example in Figure 2(b), would return an empty set, since there are no data flows between instructions added by `3e8882e` and instructions added by `0872f49`. Applied to `ea8426c` and `0872f49`, we obtain: $\{($**`alloca`**$:4,$ **`load`**$:9), ($**`store`**$:7,$ **`load`**$:9), \ldots\}$, where **`inst`**$:n$ denotes instruction **`inst`** at line number n from Figure 2(b).

Once we created a fixed mapping from commit information to IR and determined the interactions for our program, we can investigate interactions between commits derived from interactions between instructions.

Definition 4. The constructor CI for a commit interaction is a function that maps a pair of instructions that interact with respect to $\rightsquigarrow$ to a pair of commits.

$$\mathrm{CI}(i_1, i_2) = \big(\mathsf{base}(i_1), \mathsf{base}(i_2)\big) \quad \text{with} \quad \mathsf{base}(i_1) \neq \mathsf{base}(i_2) \;\wedge\; i_1 \rightsquigarrow i_2$$

Intuitively, two commits interact when code added by the first commit interacts with code from the second commit. In our example, we obtain the commit interaction (`ea8426c`, `0872f49`) for the pair (**`alloca`**$:4,$ **`load`**$:9$) of instructions. That is, code changes from `ea8426c` interact with changes introduced by `0872f49`.

To further aggregate information from commit interactions and add a context-specific meaning, we group commit interactions for every instruction into a commit-interaction path.

Definition 5. The constructor CIP for a commit-interaction path takes an instruction $\mathtt{i} \in \mathrm{I}$ and outputs a tuple whose first element is the set of all commits that interact with the base commit of $\mathtt{i}$, except the base commit itself; the second element is the base commit for $\mathtt{i}$.

$$\mathrm{CIP}(\mathtt{i}) = \Big(\big\{\, \mathsf{base}(\mathtt{i}') \mid \mathtt{i}' \rightsquigarrow \mathtt{i} \;\wedge\; \mathtt{i}' \in \mathcal{P} \;\wedge\; \mathsf{base}(\mathtt{i}') \neq \mathsf{base}(\mathtt{i}) \,\big\},\ \mathsf{base}(\mathtt{i})\Big)$$

A commit-interaction path aggregates over multiple commit interactions based on a target instruction $\mathtt{i}$. All commits, except the base commit of the target instruction $\mathtt{i}$, belonging to the interaction set of $\mathtt{i}$ are merged into a single set. For the **ret** instruction in Line 12 from Figure 2(b), we obtain the corresponding $\mathrm{CIP}(\textbf{ret} : 12) = (\{\mathtt{c4d9b1a}, \mathtt{ea8426c}\}, \mathtt{0872f49})$, which indicates that two other commits, c4d9b1a and ea8426c, interact with the base commit 0872f49 at instruction **ret**.

2.3 Computing Commit Interactions

For computing commit interactions, we have implemented a flow- and fully context-sensitive, alias-aware, inter-procedural taint analysis based on *Interprocedural Distributive Environments (IDE)* [50].

IDE is an algorithmic framework to implement data-flow analysis. To check whether a property of interest holds at a certain point in a program, IDE constructs a so-called *exploded super-graph (ESG)*. An ESG is constructed by replacing each node in the program's inter-procedural control-flow graph with a bipartite graph representation of the corresponding flow function. Flow functions for *identity* (id), *generating* (gen), and *removing* (kill) data-flow facts are distributive and can be represented as bipartite graphs, as Figure 3 shows. Thus, all gen/kill-problems such as uninitialized variables, available expressions, reaching definitions, and taint analysis can be expressed in IDE. If a node $(\mathtt{i}, \mathrm{d})$ in the ESG is reachable from a special tautological node Λ, then the data-flow fact d ($\in \mathrm{D}$, the data-flow fact domain) holds at instruction $\mathtt{i}$ ($\in \mathrm{I}$, the set of program instructions). In addition, ESG edges can be annotated with lambda functions to specify value computations that are solved over a separate value domain V. These so-called *edge functions* allow one to encode an additional value computation problem that is solved while performing a reachability check. The runtime of IDE is $\mathcal{O}(|\mathrm{N}| \cdot |\mathrm{D}|^3)$ [50], where $|\mathrm{N}|$ is the number of nodes in the inter-procedural control-flow graph and $|\mathrm{D}|$ is the size of the data-flow domain D. Thus, the analysis efficiency highly depends on the size of the underlying data-flow domain. The value domain V can even be infinite and does not affect the algorithm's complexity. Rather than encoding a linear-constant propagation using flow functions that operate on the data-flow fact domain $\mathrm{D} := \langle v, c\rangle$, with $v \in \mathcal{V}$ the set of program variables and $c \in \mathbb{Z}$ that comprises tuples of program variables and their constant integer values, a linear-constant propagation can be instead encoded much more efficiently using $\mathrm{D} := \mathcal{V}$ and $\mathrm{V} := \mathbb{Z}$. This enables the IDE framework to propagate only constant program variables as data-flow facts while computing their constant values on the separate edge function domain. The effect of a set of instructions can be summarized by composing flow (and edge) functions. The composition $h = g \circ f$ of two flow functions f and g, called *jump function*, can be obtained by combining their bipartite graph representations. h can be produced by merging the nodes of g with the corresponding nodes of the domain of f. Once a summary for a complete procedure p has been constructed, it can be (re)applied in each subsequent context where the procedure p is called. Figure 4 shows an excerpt of a program and its respective ESG for the taint analysis that SEAL uses to compute commit interactions.

Taint analysis is a parameterizable analysis that tracks values that have been *tainted* by one or more *sources* through the program and that reports potential leaks if a tainted value reaches a *sink*. *Sources* and *sinks* may comprise functions and instructions. The taint analysis $\mathcal{T}$ that we use in our experiments tracks data flows between the instructions of a given target program. It treats all

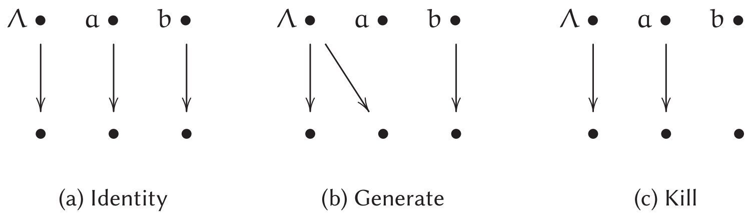
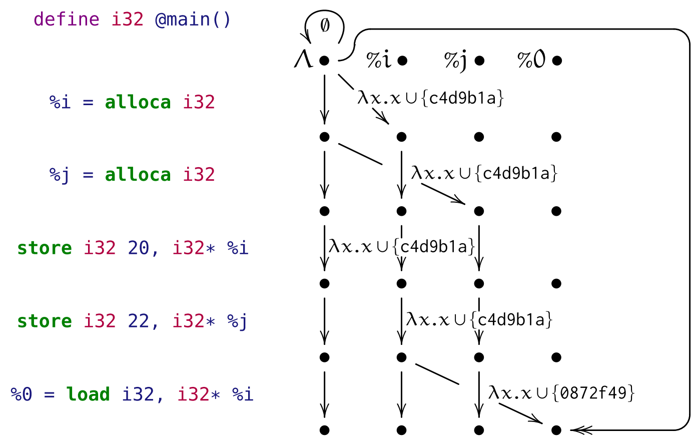


Fig. 3. Distributive flow functions and their bipartite graph representations.



```
define i32 @main()

%i = alloca i32

%j = alloca i32

store i32 20, i32* %i

store i32 22, i32* %j

%0 = load i32, i32* %i
```

Fig. 4. Excerpt of the exploded super-graph for analysis $\mathcal{T}$ conducted on the program shown in Figure 2(b). Identity edge functions ($\lambda x.x$) have been omitted. Solid arrows ($\longrightarrow$) indicate individual flow (and edge) functions. The two-headed arrow ($\twoheadrightarrow$) indicates a single *jump function* j that summarized the effects of the complete function on `%0`; the remaining jump functions have been omitted to avoid cluttering. The jump function j specifies the value computation problem $j = \{0872f49\} \cup \{c4d9b1a\} \cup \{c4d9b1a\} \cup \emptyset$ and evaluates to `%0` $\rightsquigarrow \{0872f49, c4d9b1a\}$.

variable declarations as *sources* and propagates these variables through the program. As we are interested in all instructions that interact with *tainted* variables, our set of *sinks* is empty. We then lift the data flows (i.e., the interactions of instructions with each other) to their respective commits, such that we can determine commit interactions (cf. Section 2.2).

We define the relevant intra-procedural (*normal*) flow and edge functions, which utilize our definitions from Section 2.2 to access commit information, formally in Figures 5 and 6. For the sake of brevity, we omit a formal description for inter-procedural (i.e., *call, return, call-to-return*) flow and edge functions and describe them only informally! The *call* and *return* flow functions map actual parameters onto the formal parameters at a call site, and vice versa at a callee's exit instructions (return or throw instructions). The *call-to-return* flow function generates flow facts for calls to heap-allocating functions, such as, `malloc()` or `operator new()`, and propagates all data-flow facts alongside a call site that are not involved in the function call under analysis. The *call* and *return* edge function implementations are realized as identity, and the *call-to-return* edge function implementation forwards to the *normal* edge function implementation.

The analysis $\mathcal{T}$, starting at the program's entry point `main`, *taints* the target program's variables (e.g., **`alloca`** instructions, cf. Figure 2(b)) as they occur and propagates them as data-flow

$$\llbracket i : \textbf{alloca}\ x \rrbracket(d) \triangleq \begin{cases} \{d, x\} & \text{if } d = \Lambda \\ \{d\} & \text{otherwise} \end{cases}$$

$$\llbracket i : y = \textbf{load}\ x \rrbracket(d) \triangleq \begin{cases} \{y, x\} \cup pts(x) & \text{if } d \in pts(x) \\ \{d\} & \text{otherwise} \end{cases}$$

$$\llbracket i : \textbf{store}\ x\ y \rrbracket(d) \triangleq \begin{cases} \{x, y\} \cup pts(y) \cup pts(x) & \text{if } d \in pts(x) \\ \{i\} & \text{if } d = y \vee d \in pts(y) \\ \{d\} & \text{otherwise} \end{cases}$$

$$\llbracket i : \textbf{inst}\ o_j \rrbracket(d) \triangleq \begin{cases} \{d, i\} & \text{if } d = o_j \\ \{d\} & \text{otherwise} \end{cases}$$

Fig. 5. $\mathcal{T}$'s *normal* flow functions. Each type of instruction that is important to the analysis is associated with its respective distributive flow function that specifies the ESG edge that needs to be constructed. Function `pts()` returns the points-to set of a given value. We assume that $x \in$ `pts(x)` always holds.

$$\llbracket i : \textbf{alloca}\ x \rrbracket(d_1, d_2) \triangleq \begin{cases} \lambda x.\{base(i)\} & \text{if } d_1 = \Lambda \wedge d_2 = x \\ \lambda x.x & \text{otherwise} \end{cases}$$

$$\llbracket i : \textbf{store}\ x\ y \rrbracket(d_1, d_2) \triangleq \begin{cases} \lambda x.\{base(i)\} & \text{if } x = \mathbb{C} \wedge d_1 = d_2 \wedge d_1 \in pts(y) \\ \lambda x.x \cup base(i) & \text{if } d_1 \in pts(x) \wedge d_2 \in pts(y) \\ \lambda x.\emptyset & \text{if } d_1 = d_2 \wedge d_1 \in pts(y) \\ \lambda x.x & \text{otherwise} \end{cases}$$

$$\llbracket i : \textbf{inst}\ o_j \rrbracket(d_1, d_2) \triangleq \begin{cases} \lambda x.x \cup base(i) & \text{if } (d_1 = \Lambda \wedge d_2 = o_{j_0^n}) \vee (o_{j_0^n} = d_1 \wedge d_1 = d_2) \vee \\ & \quad (o_{j_0^n} = d_1 \wedge i = d_2) \\ \lambda x.x & \text{otherwise} \end{cases}$$

Fig. 6. $\mathcal{T}$'s *normal* edge functions that specify a computation along the ESG edge $d_1 \longrightarrow d_2$ for a given instruction and data-flow facts d_1, d_2. $\mathbb{C}$ represents a constant literal. The function `pts()` retrieves the points-to set for a given variable. We assume that $x \in$ `pts(x)` always holds.

facts through the program. When analyzing libraries, the analysis treats every publicly accessible function as an entry point. Each *tainted* variable (i.e., data-flow fact) is associated with a set of commits that is encoded in lambda calculus using IDE's *edge functions*. Initially, it contains only a data-flow fact's base commit produced by $base(i)$. Whenever an instruction i interacts with one of the data-flow facts d, base(i) is added to d's associated set of commits. A set of commits of a data-flow fact d can be overwritten if it is used as a target of a store instruction. In this case, all elements of d are removed and the commit of the store instruction itself as well as all elements of the set whose associated data-flow fact is to be stored are added. An excerpt of the exploded super-graph for our taint analysis $\mathcal{T}$ conducted on the program from Figure 2(b) is shown in Figure 4.

In the context of our general definitions from Section 2.2, $\mathcal{T}$ computes the interactions (data flows) and constructs a CIP_i for every instruction $i \in \mathcal{P}$ as output.

2.3.1 *Indirect Function Calls.*

Our IDE-based taint analysis can "see" through indirect call sites [50]. The IDE algorithm is guided through the program under analysis with help of an *inter-procedural control-flow graph (ICFG)* that includes call-graph information. We use a call-graph algorithm that resolves indirect calls to function pointers or virtual functions using points-to information computed by a scalable (inter-procedural) Andersen-Style [1] points-to analysis. There is no difference between C and C++ according to the analysis. Calls to function pointers are resolved by computing the points-to set of the respective function pointer. Calls to virtual functions are resolved by computing the points-to set of the respective receiver object to find the corresponding virtual function table for statically determining potential callee targets.

2.3.2 Soundness and Completeness. Our taint analysis presented in Section 2.3 is *unsound*. This has good reasons: Implementing an analysis that computes a more complex semantic property on realistic C/C++ programs in a sound manner *and* in an inter-procedural (i.e., whole program) setting is virtually impossible or would introduce so much imprecision that it renders the analysis results unusable [40]. Instead, our analysis aims at *soundiness* [40], a well-known term in static analysis. *Soundy* analyses apply sensible underapproximations to compute meaningful results in an inter-procedural analysis setting and are widely accepted in the static analysis community [40, 56]. A soundy analysis, for instance, would sanely assume that system calls and calls to libC behave as expected: Calls to such functions are not analyzed and instead, a summary that models their effects is consulted when they have a relevant effect on the client analysis. This is also why all static analyses used for compiler optimization that aim at computing more complex properties are intra-procedural only.

With respect to completeness, our taint analysis is set up to analyze all functions whose definitions are available. The call targets of system calls and calls to libC are typically available only as declarations or are modeled as intrinsic functions by the LLVM framework. LLVM represents specific low-level functions, such as `memcpy` or `memset` as intrinsic functions for which there are no definitions. Instead, these function declarations are used to describe only semantics. So, it is up to the code generator to replace them with a software or hardware implementation when generating machine code for the desired target architecture. Our taint analysis hence models function calls to the system and libC by applying summaries that describe their effects on points-to, call-graph, and data-flow information. All other call sites, for which the corresponding call targets are available only as declarations, are *soundily* [40] modeled using the identity transformation.

3 IMPLEMENTATION

In this section, we explain how we instantiated SEAL on top of Clang and LLVM, creating Vlang[4] as a modified version of Clang. In what follows, we provide an overview of the full commit-analysis pipeline and how we compile and analyze real-world projects, with the help of Vlang and the analysis framework PhASAR [51].

3.1 Lowering Commit Information to LLVM-IR

The first step of our analysis pipeline (cf. Figure 7) is the lowering step from an *abstract syntax tree (AST)* to LLVM-IR, in which we enrich LLVM-IR with commit meta-data. During the compilation of a translation unit, Vlang computes for each AST node the last commit that modified the related code and adds it as meta-data to the corresponding generated IR instructions. When lowering an instruction, Vlang queries our framework for relevant commit information, providing a corresponding file, line number, and line offset from the AST node's expansion location (i.e., the location in the file before macro expansion). The framework then computes the blame of the file using the library libgit2.[5] It is important to note that the method for identifying last modifying

[4]https://github.com/se-sic/vara-llvm-project/.
[5]https://github.com/libgit2/libgit2/.

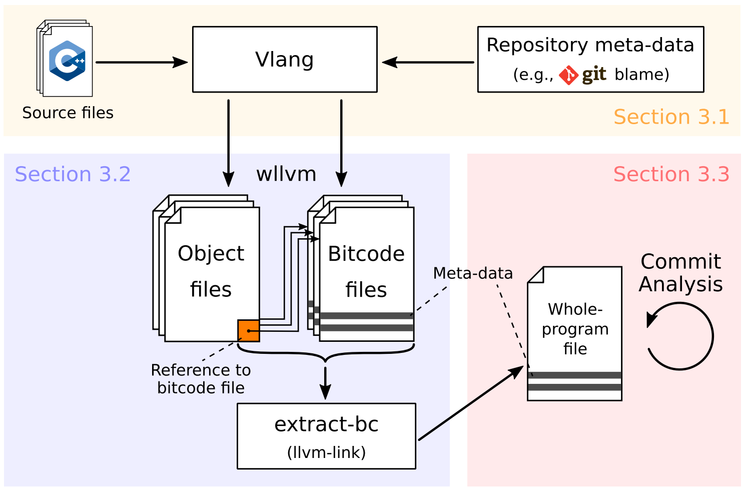


Fig. 7. Overview of the full build and commit-analysis pipeline. WLLVM injects Vlang into the project's compilation process to generate LLVM-IR files, containing commit meta-data for every translation unit. Then, all LLVM-IR files are combined into one whole-program file, which is subsequently analyzed to compute commit interactions.

commits can be configured by the user. In our study, we use git blame. Then, Vlang generates a commit meta-data object based on the line number's commit hash and the commit meta-data returned back to Vlang and attaches it to the instruction.

For illustration, consider our example of Figure 2(a): On the right, we show commit hashes per code line. During compilation, Vlang creates an AST and lowers it to LLVM-IR. Figure 2(b) shows the LLVM-IR output after lowering. Let us start with the first statement in Line 3. During lowering, Vlang creates two instructions for this statement: one allocation for the stack variable (`alloca` in Line 4 of Figure 2(b)) and one to initialize it to `20` (`store` in Line 7). As one can see, Vlang added commit meta-data to both of these instructions referenced by the meta-data tag `!Commit` and an identifier ID. The ID `!9` points to the meta-data section of the file containing the commit hash. For the sake of simplicity, we leave out the actual meta-data nodes and depict the hash on the right-hand side of Figure 2(b).

After Vlang has processed the input file, all generated LLVM-IR that are related to commits from the project's Git repository are tagged with the corresponding commit meta-data. The enriched LLVM-IR serves as input to our data-flow analysis (cf. Section 3.3).

3.2 Creating a Whole-program Bitcode File

A precise commit-interaction analysis requires the data-flow analysis to be inter-procedural, i.e., whole program and context sensitive [53]. Analyzing every compilation unit in separate bitcode file leads to approximations whenever callees of a call site are defined in another translation unit. We thus implemented our analysis to be whole program. To create a whole-program bitcode file from the project's source code, we inject our tool Vlang into the build process. This enables us to reuse the existing build scripts by using using Whole Program LLVM (WLLVM)[6] as a compiler wrapper, which invokes Vlang and generates and links bitcode files.

[6]https://github.com/travitch/whole-program-llvm/.

3.3 Taint Analysis for Commit Interactions

We implement our taint analysis $\mathcal{T}$ as an IDE [50] analysis in the PhASAR [51] framework. PhASAR has been built on top of LLVM and provides, among others, a generic IDE solver implementation and all required infrastructure (e.g., control-flow analysis) to solve concrete client data-flow analysis problems.

PhASAR's generic IDE [50] solver operates on a problem interface type whose implementations correspond to concrete data-flow analysis problems. The interface mainly comprises flow and edge function factories that are queried by the data-flow solver to construct the exploded super-graph and solve the value computation problems that are specified along the edges. We implemented these flow and edge functions factories according to our descriptions in Section 2.3.

3.4 Data Overview

SEAL aims to be extendable, and the results shall be integratable into existing study setups and tool. For this purpose, SEAL has a four-layered design, where at each layer the relevant information can be extracted for use by external tools. In what follows, we give a short overview of the data that is produced by each layer and describe how other tools can access it.

AST-level commit information. In the preparation step, before data-flow analysis, SEAL's compiler extension VLANG makes commit information accessible through LLVM's AST. The commit data is provided by a general abstraction in VLANG that offers an interface to map AST nodes to commit hashes. Through this interface, tools have an easy way to query commit information related to a particular AST node (e.g., to combine commit information with error messages).

LLVM-IR with commit information. During LLVM-IR code generation, VLANG attaches commit information provided by the AST interface to the generated llvmir information in form of meta-data. This way, commit information is attached to LLVM's internal representation and can be queried, like any other meta data, through the usual framework API (e.g., to attribute LLVM's warnings about missed optimizations with author information).

Data-flow-based commit interactions. After commit interactions have been computed, they can be accessed within LLVM's analysis infrastructure, enabling other analyses to query this information. As described in Section 4.2, this enables other analyses to attach socio-technical information to their analysis results (e.g., an analysis that detected an SQL injection can automatically determine developers interacting with the vulnerable code and include them in the resolving process). In addition, these raw commit-interaction data can be exported into a yaml for further analysis.

Aggregated socio-technical information. To ease the analysis of data-flow-based commit interactions, SEAL provides different graph aggregations of the raw commit interaction data, including: commit, author, and commit-author graphs, which we use in Section 4.1. With these graphs, existing approaches have a straightforward way to integrate the data-flow-based commit information.

4 COMMIT INTERACTIONS: APPLICATIONS

To illustrate the merits and potential of SEAL, we discuss research problems from different domains that can be addressed only by a combination of high-level repository information and low-level data-flow information. More importantly, some problems can be analyzed only if the high-level information is already available *during* analysis. We will use these problems in the next section to evaluate SEAL from two angles: *qualitatively*, using scenarios we found in real-world software projects demonstrating how the questions can be answered using SEAL, and

```c
/* Make sure everything is properly aligned. */
static OPUS_INLINE int align(int i)
{
  struct foo {char c; union { void* p; opus_int32 i; opus_val32 v; } u;};

  int alignment = offsetof(struct foo, u);

  /* Optimizing compilers should optimize div and multiply into and
  for all sensible alignment values. */
  return ((i + alignment - 1) / alignment) * alignment;
}
```

Fig. 8. The function `align` in file `src/opus_private.h` from OPUS revision `348e694`.

quantitatively, using a number of real-world case studies demonstrating that our approach is indeed practical.

4.1 Commit Interaction Graph Analysis

First, we address three problems related to the field of repository mining and similar areas of research that are hard to approach using only high-level or low-level information, but become much more tangible when combining these two. For our evaluation, we aggregate commit interactions in a *commit interaction graph*. Not only does a graph representation come very naturally, because of the data-flow relation that is used to compute commit interactions, but it also allows us to use methods from graph theory to reason about commit interactions. We can also apply transformations to the graph to access different kinds of information, e.g., information about commit authors.

Central code. With commit interactions, we are able to quantify the impact of the changes made by a commit on the program's data-flow dependency structure. This is related to the area of *change impact analysis*, where researchers have developed a multitude of techniques on how to estimate the impact of a change on a software project, using either high-level information or dependency information [36, 38]. While there are approaches that use both [15, 18, 20, 31], with SEAL, we can *combine both kinds of information at the same time in a joint analysis*.

Previously, Zimmermann and Nagappan [63] used dependency graphs to identify central program units in a software project and highlight their role as a proxy for defect-prone code. In a similar fashion, Ferreira et al. [14] investigated how interactions between functions in the presence of preprocessor directives relate to the occurrence of vulnerabilities. With commit interactions, we can identify code locations that are central in the dependency structure of the software system under analysis *at a much finer granularity* and, in addition, link them to commit meta-data (e.g., when or by whom the code was introduced). Changes to such *central code* are interesting, because their effect on the data that flows through the program is likely very high. For example, the function `int align(int i)` from the audio codec OPUS—one of the subject projects of our evaluation in Section 5—calculates how much memory an object of size `i` needs when it is stored with proper memory alignment (Figure 8). The function receives the size of the object via the parameter `i`. That parameter is then used to calculate the size with alignment, which is then returned from the function, meaning that there is a data flow from the parameter to the return value. As a consequence, there are also data flows between values passed to the function and usages of its return value. Also, the function is widely used throughout OPUS with 67 usages across 8 of 22 source files, which causes it to have many data flow connections to many different locations in the system. So, it is fair to say that this function is indeed central. To show how commit interactions can be used to identify such central code, we formulate the following problem:

P*$_1$*:* *Which fraction of commits affects central code?

It is important to reiterate at this point that this problem, while being interesting in itself, serves here as a showcase of demonstrating SEAL's ability to address this and similar problems in practice and research in a more systematic and efficient way to what is possible so far. Before we measure the centrality of the code introduced by a commit, we first define how we represent commit interactions in a commit interaction graph.

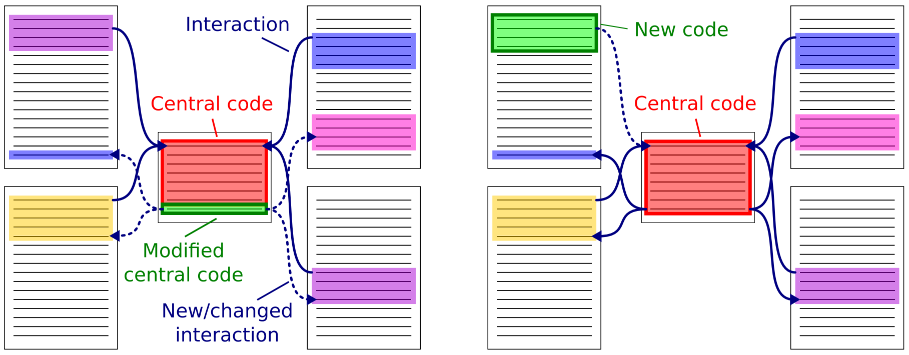
Definition 6. The *commit interaction graph* of a program is a directed graph with commits as nodes and interactions among commits as directed edges: CIG $= (\mathcal{C}', \mathcal{CI})$.

In this definition, $\mathcal{CI}$ refers to the set of all commit interactions of a program, and $\mathcal{C}' \subseteq \mathcal{C}$ is the set of commits that participates at least in one commit interaction, that is, $\mathcal{C}' = \{ c_1, c_2 \mid (c_1, c_2) \in \mathcal{CI} \}$. Note that this graph may contain multi-edges, since there may be commit interactions that originate from different instructions but have the same base commits.

With commit interactions represented in a graph, we can now employ methods from graph theory to identify commits affecting central code. We identify such commits by identifying central nodes in the commit interaction graph using the node degrees of the commits. We use the node degree to measure centrality, since it relates to our definition of central code quite well; other centrality measures are possible as well. A high node degree means that a commit participates in interactions with many other commits, which is an indicator that the code introduced by that commit is central in the dependency structure of the software system under analysis. We are particularly interested in commits that introduce a relatively small change, since such commits are more easily overlooked than very large commits. This does not mean, however, that large commits cannot introduce central code. In fact, just because of their size, we expect large commits to be very likely to introduce, at least, some central code. For our example from OPUS, each usage of function `align` produces incoming interactions (size of the object) and outgoing interactions (size with alignment), and thus, commits associated with that function have a high node degree. Therefore, any—even small—change to `align` that affects its return value affects *every* location the function is used at. At the same time, the function itself and consequently any commit that touches only that function is only a few lines long. This scenario is illustrated in Figure 9(a), where a small change to central code introduces many outgoing interactions with other commits. In Figure 9(b), new code is introduced that is not central by itself, but merely interacts with central code. In this case, only a few interactions are attributed to the new commit. This scenario illustrates that commits touching central code can have a large effect on the rest of a software project and, if such commits can be detected, this information might guide testing and review efforts on these critical changes to the code base. SEAL helps to identify such cases.

Author interactions. Commits consist not only of code changes, but they also contain metadata, such as when or by which author a commit was created. From these data, SEAL can extract information about socio-technical interactions in a software project. Socio-technical interactions are often analyzed using communication or collaboration relationships between developers at the file or function level and may also include dependencies between artifacts [17, 24–26, 42]. However, indirect dependencies are missed this way, which may still be relevant to properly characterize certain aspects of the socio-technical interactions in a project, e.g., classifying the roles of developers.

For example, consider the function `ssh_handle_packets` from LIBSSH in Figure 10. In this function, a context object, which is returned by `ssh_poll_get_ctx`, is passed to `ssh_poll_ctx_dopoll`, creating an indirect dependency between the two functions. This indirect dependency is interesting, because information carried by the context object could be relevant for the computation in `ssh_poll_ctx_dopoll`. Thus, if a developer modifies the context object



(a) A line belonging to central code is changed (green) resulting in many changed interactions (dashed arrows).

(b) Introduced code (green) that interacts with central code but only adds one new interaction (dashed arrow).

Fig. 9. The influence of changes related to central code on commit interactions. The colored boxes represent different commits and arrows denote commit interactions.

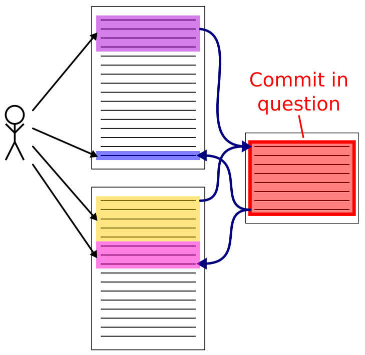
```
1  int ssh_handle_packets(ssh_session session, int timeout) {
2      ssh_poll_handle spoll;
3      ssh_poll_ctx ctx;
4      ...
5      spoll = ssh_socket_get_poll_handle(session->socket);
6      ssh_poll_add_events(spoll, POLLIN);
7      ctx = ssh_poll_get_ctx(spoll);
8      ...
9      int rc = ssh_poll_ctx_dopoll(ctx, tm);
10     ...
11     return rc;
12 }
```

Fig. 10. Shortened version of function `ssh_handle_packets` from libssh. The context object `ctx` returned by `ssh_poll_get_ctx` in line 7 is passed to `ssh_poll_ctx_dopoll` in line 9, creating an indirect data flow between the two functions. SEAL can detect such indirect connections.

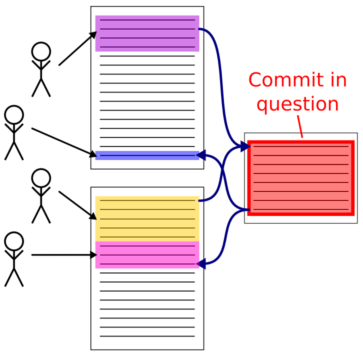
in `ssh_poll_get_ctx`, then he/she also indirectly influences the function `ssh_poll_ctx_dopoll` but might not be aware of that. With SEAL, we can incorporate such indirect dependencies in socio-technical analyses that can only be detected with a data-flow analysis and, thus, uncover hidden dependencies between developers. We formulate the following problem to demonstrate that SEAL's information on commit interactions are useful for answering socio-technical questions about a software project:

P_2: *Which authors interact via commit interactions and what are the characteristics of the arising socio-technical structure?*

By lifting SEAL's commit interaction graph to author information, we can easily identify which authors interact with each other indirectly via data flow. For this purpose, we project on the commit interaction graph such that the authors of the commits become nodes and interactions between authors become directed edges. This projection can be implemented with vertex identification, that is, all nodes whose commits have the same author are identified with each other. The remaining edges then represent interactions between authors at a data-flow level. With this information, we



(a) A commit (right) interacts with many commits authored by a *single* developer.



(b) A commit (right) interacts with many commits authored by *many* developers.

Fig. 11. Code from a commit can interact with code authored by one or multiple developers.

can not only see which authors interact with each other, but also which authors interact with especially many (or few) other authors, for example, to determine an author's role in a software project [8, 25].

Commit–author interactions. A commit can interact with commits from one (Figure 11(a)) or multiple authors (Figure 11(b)). The fact that a high number of authors participate in interactions for a commit suggests that the author needs to be familiar with code from many different developers rather than with their own code or the code of only few developers. This again may have implications for the bug-proneness of the associated source code [23] and emerging coordination requirements between authors [22]. Another example where commit–author interaction data is useful is to select potential candidates for code review by determining which authors' code is affected by the commit to be reviewed. To collect this information, we need both, commit interaction data (data flow) and commit meta-data (author names). We demonstrate that with SEAL, we indeed can combine these, addressing the following problem:

P_3: *How many authors interact via commit interactions?*

To address this problem, we combine the commit and author interaction graphs. The resulting graph contains all commits and authors as nodes, and a directed edge from a commit c_1 to an author a if and only if there is an edge (c_1, c_2) in the program's commit interaction graph and a is the author of commit c_2. Note that this graph can also be constructed directly from information contained in the commit interaction graph. The outgoing node degree of commit nodes gives us the number of interacting authors showing whether a commit interacts with many or only a few authors.

4.2 Socio-technical Data-flow Analysis

Data-flow analysis is typically not concerned with commit or socio-technical information, but it can highly benefit from this additional information. First, computing commit interactions helps solve a multitude of additional data-flow problems with virtually no additional overhead compared to computing commit interactions only. Second, combining information on commit interactions computed by SEAL with information of a client data-flow analysis provides new insights that were previously locked away. As an example, taint analysis is frequently used to detect code injection vulnerabilities, such as SQL injections, but it can report *only* the detected potential security issues; it cannot attribute its findings to a specific project version, author(s), or development team.

```
string sanitize(const string &s) {                              ▷ de8781b
  if (test) { return s; }                                       ▷ ea8426c
  return sanitizeSQLString(s);                                  ▷ de8781b
}                                                               ▷ de8781b
int main(int argc, char **argv) {                               ▷ c4d9b1a
  ...
  string input = argv[1];                                       ▷ 0872f49
  string sani = sanitize(input);                                ▷ 5341f7b
  auto *res = stmt->executeQuery(q + sani);                     ▷ 5341f7b
  ...
}                                                               ▷ c4d9b1a
```

Fig. 12. A program that is vulnerable to SQL injections. Each ▷ indicates the commit that last modified each line. The complete code example is shown later in Figure 21.

As described in Section 2.3, our commit interaction analysis $\mathcal{T}$ needs to exhaustively compute the precise, full exploded supergraph using IDE [50] to generate the commit data. It propagates all variables of a given target program and therefore computes all data flows for all program variables. Thus, $\mathcal{T}$ also solves all data-flow problems that are concerned with data flows of program variables. All variations of taint analysis—for any given set of sources and sinks—can therefore be directly answered using the exploded supergraph that has already been constructed for $\mathcal{T}$. Reusing the parts of a previously computed exploded supergraph for a new analysis is beneficial, since IDE's [50] runtime complexity is $\mathcal{O}(|\mathrm{N}| \cdot |\mathrm{D}|^3)$.

More importantly, a taint analysis that is set up to detect SQL injections, for example, cannot only be solved on $\mathcal{T}$'s exploded supergraph, but, in contrast to a traditional data-flow analysis, also access commit information. SEAL allows one to *compose* different data-flow analyses with commit information and, for the first time, allows us to embed statements obtained by program analysis into the socio-technical context of a software project. To demonstrate that augmenting client data-flow analysis results with information on commit interactions indeed provides novel insights, we formulate the following problem:

P_4: *Can commit information be utilized to gain additional insights from traditional data-flow analyses by making them change-aware?*

To showcase how SEAL can be used to enrich existing data-flow analyses, we employ an existing taint analysis from PhASAR. We parameterize the taint analysis for detecting SQL injection vulnerabilities, and we enrich it with commit information. For an example, consider Figure 12, which depicts a program snippet that is vulnerable to SQL injection attacks. Any user input is considered as tainted and must be *sanitized* by a call to function `sanitizeSQLString` before it is sent to the SQL database server using the *sink* function `executeQuery`. By allowing the taint analysis to exchange information with the commit analysis $\mathcal{T}$, SEAL can attribute the findings directly to the commits, authors, as well as development teams that are involved in the critical data flows (and potential security vulnerabilities) as reported by the taint analysis. This allows one to determine which commit introduced a potential vulnerability, which authors worked on the code that caused the data flows involved, and which developers should look into and the reported issue. A further illustration of how commit data can help to find code authors related to an SQL injection vulnerability is given in Section 7.2. In practice, static analysis—especially if conducted in a whole program manner—produces lots of potential findings, many of which may be false positives [2, 19], such that it has become a real challenge to prioritize and check them. It is important to note that SEAL cannot determine per se whether a finding is a true or false positive with respect to the original analysis' semantics. But with socio-technical information, one can contextualize the results to asses their likelihood, and SEAL helps to prioritize and distribute them by providing socio-technical context information. For example, findings that concern multiple people are potentially more complex and could therefore be processed later, or findings could be filtered to involve developers only from a specific team, so people who potentially understand the issue better can look at it.

5 SEAL IN ACTION

To demonstrate SEAL's merits and potential, we apply it to investigate the problems presented in Section 4. Again, our goal is not to evaluate these problems in full detail, as this would surely require an entire research paper on its own. Instead, we aim at demonstrating that, with SEAL, we are indeed able to tackle such problems in a way that was hard to achieve without it.

Experimental setup. We use PhASAR in its most precise configuration for our analysis $\mathcal{T}$. It uses a call graph based on points-to information computed by an *Andersen*-style [1] pointer analysis to resolve indirect call sites, and it provides our client taint analysis with control-flow, type-hierarchy, and points-to information. Note that this setting aims at *soundyness*. Soundy analyses, introduced by Livshits et al. [39], use sensible under-approximations to cope with hard-to-analyze language features that would otherwise produce unduly imprecise results.

We selected 13 open-source C/C++ projects from a diverse set of application domains and with different sizes to increase external validity. Table 1 lists these projects along with relevant information. It also lists the revisions that we used for our analysis.

5.1 Commit Interaction Graph Analysis

In the first part of our evaluation, we address the problems P_1–P_3 of Section 4: (1) quantitatively using our subject projects and (2) qualitatively highlighting interesting insights we obtained.

Central code. P_1 is concerned with the fraction of commits that affects code that is central in the dependency structure of a program. Thereby, we are interested in commits that introduce a relatively small change (cf. Section 4).

As an example, Figure 13 depicts a scatterplot of the two relevant variables—commit size and node degree in the commit interaction graph—for OPUS and HTOP.[7] The horizontal line denotes the 20-percentile and the vertical line the 80-percentile, respectively, putting small commits with a high node degree in the upper left quadrant. The marginal distributions show that most commits are relatively small and have a low node degree. While the distribution of the commit size has a similar shape for all of our subject projects, there are some differences when it comes to node degrees. For example, for OPUS, we have three clusters: The biggest cluster consists of commits with a low node degree (below 400), followed by a smaller cluster with medium-sized node degrees (400–800), and an even smaller one where the commits have very high node degrees (above 800). It is this latter cluster that is relevant to identify small commits with central code.

As an example for a change to central code, we qualitatively inspect commit `348e694` from OPUS (marked in Figure 13(a)). This commit has a very high node degree but touches only one line that belongs to the function `int align(int i)` (cf. Figure 14), which we already established to play a central role in the program's data flow in Section 4. This demonstrates how even a small change can have a huge influence on the data that flows through a program, also showing that blame interactions carry different information than code churn.

[7]The plot excludes a few very large commits that can be considered outliers (e.g., import from the old repository, large-scale code reformatting). We excluded outliers using Tukey's fence ($k = 3$), which removes data points that lie more than k times the interquartile range above or below the first or third quartile.

Table 1. Subject Projects

	Domain	LOC	Commits	Authors	Revision
BISON	Parser	591,687	26,281	253	849ba01b8b
BROTLI	Compression	34,833	1,030	87	ce222e317e
CURL	Web tools	195,685	26,949	871	1803be5746
GREP	UNIX utils	619,598	23,794	262	70517057c9
GZIP	Compression	622,480	22,193	242	7d3a3c0a12
HTOP	UNIX utils	25,775	2,243	156	44d1200ca4
LIBPNG	File format	74,571	4,098	58	a37d483651
LIBSSH	Protocol	95,235	5,126	115	cd15043656
LIBTIFF	File format	88,561	3,470	45	1373f8dacb
LRZIP	Compression	19,215	935	25	465afe830f
LZ4	Compression	18,813	2,541	130	bdc9d3b0c1
OPUS	Codec	70,267	4,077	107	7b05f44f4b
XZ	Compression	38,441	1,298	22	e7da44d515

Metrics for BISON, GREP, and GZIP include submodules.

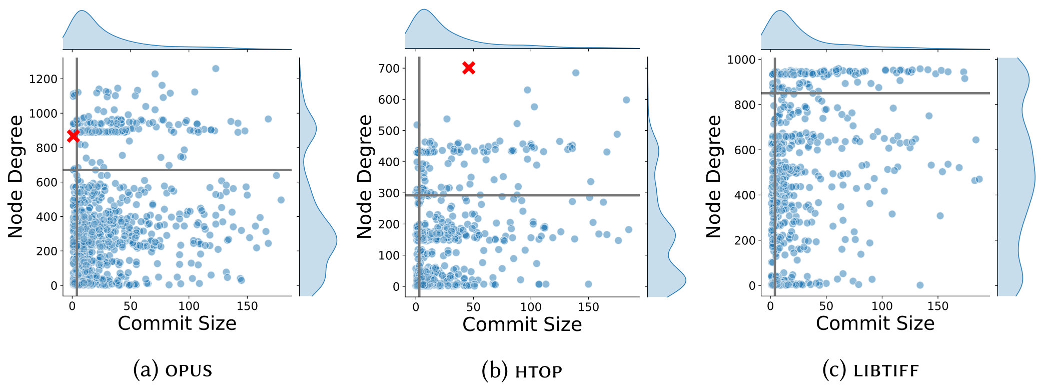
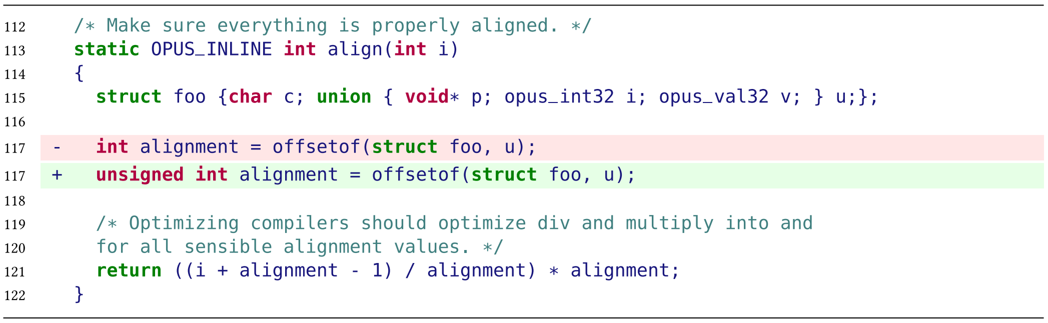


Fig. 13. Number of insertions vs. node degree for commits including kernel density estimations.



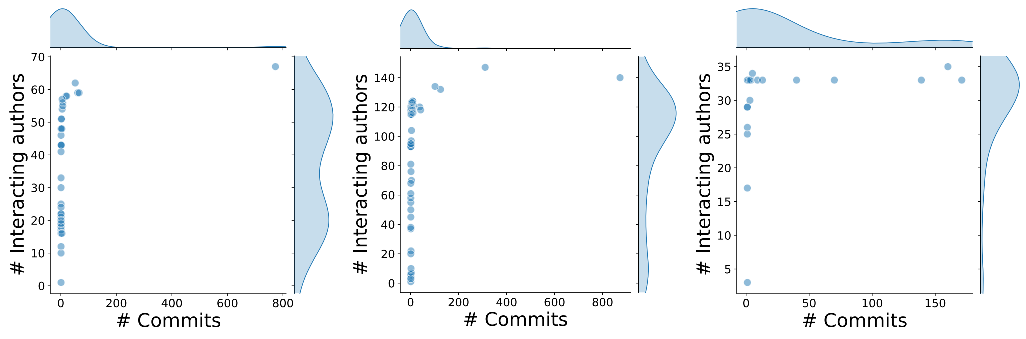
```
  /* Make sure everything is properly aligned. */
  static OPUS_INLINE int align(int i)
  {
    struct foo {char c; union { void* p; opus_int32 i; opus_val32 v; } u;};

-   int alignment = offsetof(struct foo, u);
+   unsigned int alignment = offsetof(struct foo, u);

    /* Optimizing compilers should optimize div and multiply into and
    for all sensible alignment values. */
    return ((i + alignment - 1) / alignment) * alignment;
  }
```

Fig. 14. Commit 348e694 from opus changes only a single line in the function align.

A second example is commit 5e4b182 from HTOP. This commit refactors functions that are intended to replace C's memory allocation functions and shows a very high node degree. These functions were initially introduced back in commit a1f7f28, and its successor commit b54d2dd replaced all occurrences of the original memory allocation functions in HTOP with the new



(a) OPUS (b) LIBSSH (c) LIBTIFF

Fig. 15. Number of commits vs. number of interacting authors for authors, including kernel density estimations.

substitutes contributing to their central role in the project. Commit b54d2dd also has a high node degree, but it is much larger and it touches 42 different files. This example matches the scenario described in Figure 9(a) that we used to motivate our notion of central code: Commit 5e4b182 modifies central code and, thus, affects many commit interactions despite being relatively small.

Investigating P_1 demonstrates that, with SEAL, one can gain deep insights into the structure of a software project by combining high-level repository information and low-level data-flow information into novel software metrics, e.g., estimating the importance of a change.

Author interactions. P_2 is concerned with the socio-technical structure of a software project demonstrating how meta-data associated with commits can be utilized with SEAL. In particular, we investigate how developers interact with each other at a data-flow level.

As an example, Figure 15 shows for each author of OPUS, LIBSSH, and LIBTIFF (blue dots) the number of surviving commits (commits that occur in the commit interaction graph) and the number of other authors his/her commits interact with at a data-flow level. It is immediately apparent that, in OPUS, there is one main developer who authored the vast majority of commits and hence, interacts with code from all other authors. All other authors submitted only comparatively few commits to the project. This pattern can be observed in many of our subject projects, especially the smaller ones. One notable exception to this pattern is LIBTIFF (Figure 15(c)), which has more authors that contributed a larger number of commits. Interestingly, despite this inequality in the distribution of number of commits, the number of interacting authors is more evenly distributed. That is, there are authors who introduce interactions to code from many other authors with only a few commits, whereas the other authors' code interacts with only very few other authors. Such information is useful to identify which authors contributed to central or only peripheral parts of a project [25], which is interesting, since changes to central code can have a bigger impact on the project.

SEAL is able to identify interactions between authors that cannot be detected by purely textual or syntactical approaches, which are commonly used when analyzing socio-technical aspects of software projects [17, 24–26]. Figure 16 shows the difference between author interactions computed by a file-based approach, where one considers two commits interacting if they edit the same file (co-edits), and CI-based author interactions as computed by SEAL. We notice that SEAL identifies less interactions as compared to the file-based approach, which is apparent in the negative range of the y-axis (orange). This result is in line with recent findings that a file-based approach reports many spurious links that using a more precise static analysis can avoid [18]. In addition to removing spurious interactions, for almost all of our subject projects, SEAL also finds

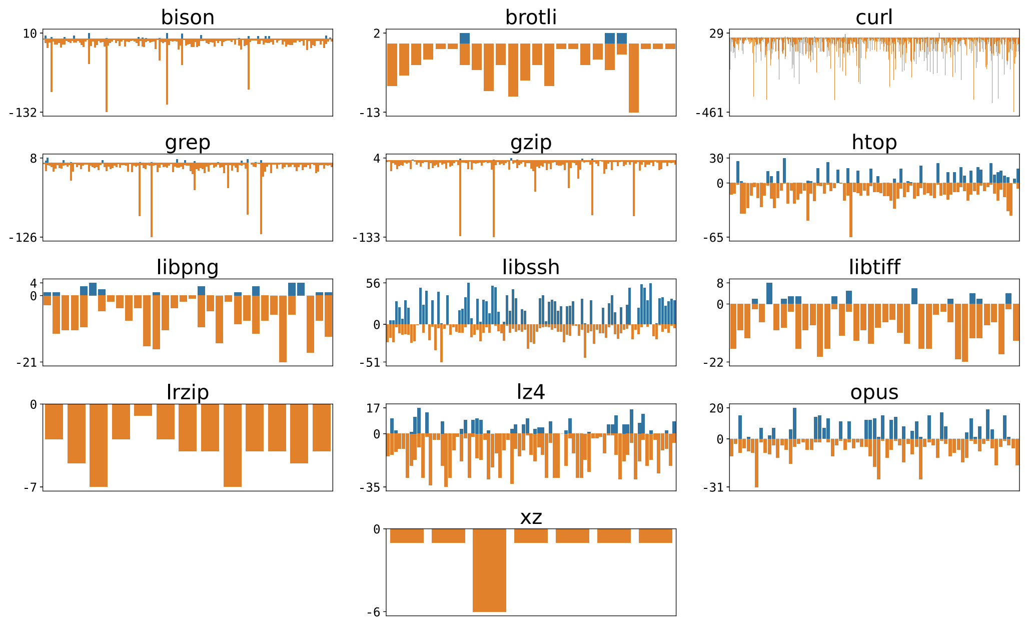


Fig. 16. Changes between CI-based and file-based author interactions. Each column represents an individual author. Orange boxes depict links to other authors that are produced by the file-based approach but are not inferred by CI, as there exists no data-flow indicating a connection between the two authors. Blue boxes depict additional links that CI discovers through data-flow information that a file-based approach does not.

unknown interactions between authors that the file-based approach could not detect, especially in larger projects. This includes interactions across files and among distant code fragments that are connected via data flow.

As an example, let us consider the author of commit 3659e8c of LIBSSH. This author contributed only one commit affecting source code to the project, which happens to implement asynchronous socket handling and, according to the commit message, "is intended as a ground work for making LIBSSH asynchronous." The implementation is mostly restricted to a single file, so approaches based on file- or function-level co-edits can only ever find author interactions within that file. However, since socket handling is a very integral part to LIBSSH, this commit actually interacts with code from all over the code base written by many different authors (we consider this a central commit according to P_1). Some of the interactions even originate from indirect dependencies that can only be detected with a precise data-flow analysis. Indeed, the indirect dependency in the function `ssh_handle_packets` as described in Section 4 involves the commit in this example. With SEAL, we not only detect this case, indeed, we find 50 additional authors that interact with code from the author in question, indirectly, via data flow.

Another common approach for calculating interactions between authors relies on call relations between functions. Figure 16 and 17 compares SEAL to an analysis that extracts call-graph data directly from LLVM (i.e., that establishes a link between two authors if a function where one contributed code to calls another where the other author contributed code to). It is important to note that both the call-based approach and SEAL report less author interactions than the file-based approach. This is further evidence that the links inferred by the file-based approach are spurious. Furthermore, the CI-based approach finds considerably more links than the call-graph-based approach, especially for larger projects. This is because SEAL also considers indirect dependencies

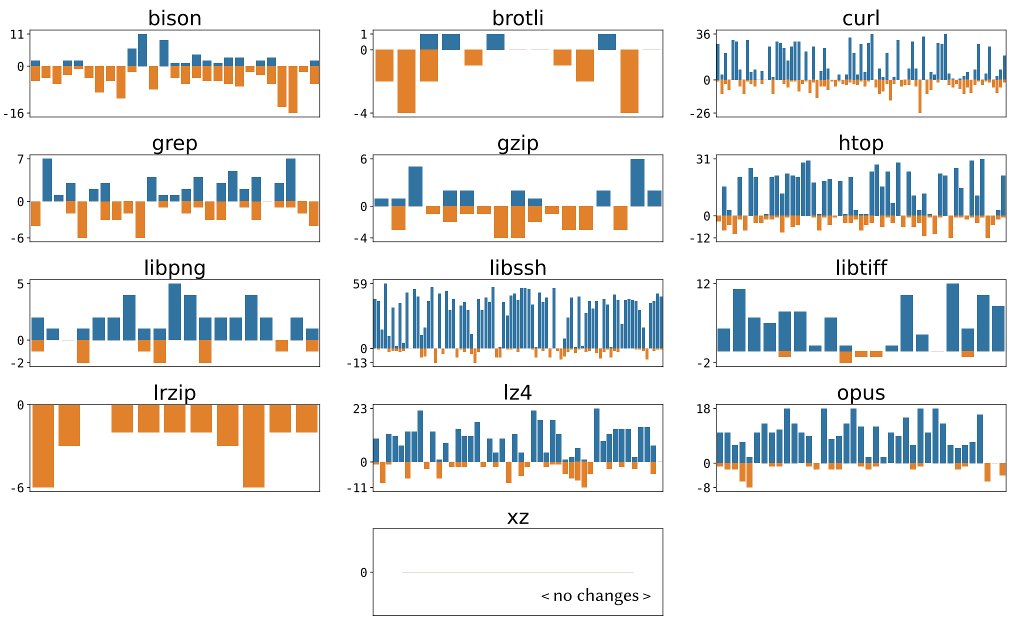


Fig. 17. Changes between CI-based and call-graph-based author interactions. Each column represents an individual author. Orange boxes depict links to other authors that are produced by the call-graph-based approach, but not by the CI-based approach, as there is no data flow indicating a connection between the two authors. Blue boxes depict additional links that CI discovers through data-flow information that a call-graph-based approach does not.

that can propagate across functions that are not connected via a call-relation. We refer the reader to Section 7.1 for a detailed example illustrating the role of indirect dependencies.

Commit–author interactions. P_3 is concerned with identifying which other authors' code a commit affects via data flow. To address this problem, we need both commit interactions and author information. Figure 18 shows for each commit its number of interacting authors, normalized by the number of distinct authors per subject project that have at least one commit participating in a commit interaction. The violin plot visualizes the associated probability density. There are significant differences across projects. An extreme is XZ: All surviving commits are from the same author and, hence, every commit interacts with commits from that one author. For other projects with few authors, such as GZIP and LRZIP, most commits interact only with code from few other authors. As projects grow and gain more contributors, most commits tend to interact with more authors (relative to the total number of authors of a project), as can be observed, for example, for BISON and LZ4. However, we do not only observe differences between projects of different size, but also between projects of similar size and a similar number of contributors. For example, while for OPUS most commits interact with about half of the authors, the results for LIBSSH show two groups of commits—one where commits interact with comparatively few authors and one where commits interact with comparatively many authors. There could be various reasons for such differences, and they cannot all be explained by commit interactions alone. For example, the roles of the authors in a project can influence with which authors their commits interact [25], and the architecture of a project could also be of relevance. The crucial point is that, with SEAL, we are able to study these kinds of socio-technical problems in the first place.

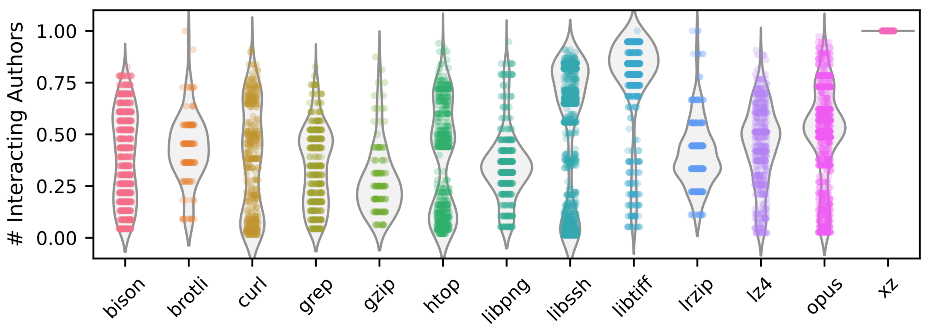


Fig. 18. Number of distinct authors each commit interacts with, normalized by the number of authors per project. Violins show the associated probability densities.

5.2 Socio-technical Data-flow Analysis

To illustrate that making a data-flow analysis change-aware provides additional insights, let us apply SEAL's augmented taint analysis (Section 4.2) on the program shown in Figure 12. The taint analysis will reveal an SQL injection vulnerability in Line 11. The analysis is able to detect the undesired data flow by checking the data-flow information for variable `argv` already computed by the commit analysis. The data-flow path that causes the potential SQL injection vulnerability comprises the following sequence of instructions: $p = i_9 \to i_{10}^{\text{callsite}} \to i_2 \to i_{10}^{\text{retsite}} \to i_{11}$.

Besides being able to answer data-flow queries directly, the taint analysis is able to access any information on commit interactions at any point in the program. For example, the analysis can query the commit that generated an instruction with $\mathsf{base}(i)$. It can therefore determine that the commit $\mathsf{base}(i_3)$ does—contrary to the intended semantics of `sanitize`—circumvent the sanitization of variable `s`. Or, using $\text{CIP}(i)$, the analysis can determine all commits—and thus all authors—involved in the disallowed data-flow path. This is highly interesting in actual software development practice, since the findings of any data-flow analysis can now be associated with the commits and developers that have been involved in the code for which an issue has been found. Combining commit and data-flow information opens up a multitude of useful scenarios: The taint analysis, in our example, is able to compute $\text{CIP}(i_{13})$, and it can therefore report that the authors of the commits `3e8882e` and `ea8426c` have been working on the code between which the undesired data flow has been found. It can report the potential SQL injection directly to these authors. Since the commit information is data-flow sensitive, the taint analysis can relate commits and their corresponding authors to the SQL injection even when they did not touch the part of the code that executes the SQL statement directly. Furthermore, the analysis is able to issue that commit `ea8426c` and its respective author (knowingly or unknowingly) introduced the vulnerability by only sanitizing the input when the `test` switch is disabled. This ability has a huge potential for addressing a large number of interesting follow-up research questions.

6 THREATS TO VALIDITY

Internal validity: Lowering commit information to LLVM-IR is complex due to the inherent technical complexity and, therefore, might introduce errors in our commit mapping. So, to validate the correctness of our commit meta-data mapping, we devised a validation procedure based on LLVM's debug meta-data that allows us to validate the lowered information. The validation procedure additionally computes the commit information on the fly, based on debug meta-data, and compares these to the annotated commit meta-data from our lowering strategy. This way, we guarantee that our meta-data are, at least, as precise as if we had used LLVM debug information for calculation. Note that using LLVM debug information in general (and not only for evaluation) would constrain our approach to only being usable with debug builds, so, building our own lowering strategy is necessary to support release builds.

We use git blame to determine the last modifying commit, which only offers line-based precision. From what we have seen, this does not distort the overall picture, as we where still able to locate many interesting cases in our subject projects, demonstrating the general merits and potential of SEAL. Nevertheless, to open up our framework for further improvements, we have set up it in a way that it enables users to exchange the commit querying functionality.

Another source that may introduce imprecision in our mapping are compiler optimizations. For certain code transformations, it is not clear how the commit meta-data should be handled, for example, when the common-subexpression-elimination pass removes code where two subexpressions originate from code added by different commits. To circumvent the influence of compiler optimization, we run our analysis passes before all optimization passes. A more general but laborious solution would be to modify all optimization passes to preserve and update commit meta-data, but running our analysis first renders this unnecessary, producing the same results.

External validity: In our evaluation, we test our approach on several relevant problems and scenarios with the goal to cover a wide range of different use cases, which we evaluated on different real-world projects, varying in size, age, and maturity. We selected 13 common C/C++ open source projects from different application domains and analyzed them qualitatively and quantitatively. Our results revealed many interesting events in the analyzed real-world software projects, which demonstrates the potential of our approach and shows that combining repository information with precise data-flow analysis can uncover previously unobservable interactions.

Since our approach and a large part of our implementation is language-independent, we see no principle roadblock for using SEAL on projects written in languages close to C/C++, such as Rust, Go, or Swift, which already have mature LLVM front-ends.

7 DISCUSSION

In the following, we illustrate the advantages of determining commit interactions based on data flow, highlight key strengths of SEAL, and discuss potential limitations. Furthermore, we lay out potential applications of SEAL and discuss how existing study setups and tools could benefit from data-flow-based CI.

7.1 Data-flow-based Commit Interactions

***Example*.** SEAL leverages a data-flow analysis to determine which commits interact, and by that, which code written by one developer influences the code of other developers. In Section 5.1, we demonstrated that SEAL can identify small but central changes to a code base by using information on data flow. We also showed that, by lifting SEAL's commit interaction graph to author information, we can build an author-interaction graph that establishes interactions based on data-flow relationships between their code. Author interactions hint at coordination requirements between authors [4, 5, 34] in that when one author's code consumes data produced by another author, the two should coordinate. Compared to existing approaches to compute coordination requirements, such as file-based [26] or call-graph-based approaches [41], considering data flow has two clear benefits: First, using data-flow information, spurious connections can be excluded (if there is no data flow between two code parts, they do not influence each other). Second, data flows, especially inter-procedural data flows, reveal dependencies between code that current approaches cannot find.

```
int DataService::process() {                        ▷ f2f294b    Sven
  int data = loadDataFromFile();                    ▷ f2f294b    Sven
  return compute(data);                             ▷ f2f294b    Sven
}                                                   ▷ f2f294b    Sven
```

(a) Data processing service written by the backend developer Sven. (File: Service.cpp)

```
int loadDataFromFile() {                            ▷ 9209cff    Eric
  return 10;                                        ▷ 9209cff    Eric
}                                                   ▷ 9209cff    Eric
                                                    ▷ 9209cff    Eric
int loadDataFromDatabase() {                        ▷ ffb0fb5    Ada
  return 21;                                        ▷ ffb0fb5    Ada
}                                                   ▷ ffb0fb5    Ada
```

(b) Loading utilities to access user data written by Eric and database developer Ada. (File: UserData.cpp)

```
int compute(int Val) {                              ▷ 28f1624    Leonie
  return Val + someComplexComputation();            ▷ 28f1624    Leonie
}                                                   ▷ 28f1624    Leonie
```

(c) Conceptual implementation of a heavy computation step built by Leonie. (File: Compute.cpp)

Fig. 19. Conceptual example of a software project with multiple authors and files.

For illustration, let us highlight conceptual differences in the data produced by existing approaches and data-flow-based commit interactions on an exemplary study in which we want to determine coordination requirements between developers based on code dependencies. Consider the example code in Figure 19: A data processing service consisting of the service itself (Figure 19(a)), a user-data access layer (Figure 19(b)), and a computation worker (Figure 19(c)). Like in the real world, the different parts of the code base are implemented by different developers (names shown on the right). To demonstrate key differences in the data produced, we compare the interactions computed by SEAL against the two commonly used approaches for computing artifact coupling, file-based and call-graph-based, which we did also use in our evaluation (Section 8). The coordination graphs computed by the different approaches are depicted in Figure 20. When comparing the file-based graph (Figure 20(a)) to the other two, we notice that link 1 between Eric and Ada is included only in the file-based graph. The point is that link 1 is actually spurious as, at the code level, there is no coordination requirement between the two functions in the file. The reason is that the file-based approach over-approximates by considering everything in a file as related without taking program semantics into account [18]. A programmer can easily see that these two functions are alternatives and, therefore, Eric and Ada can work independently. Furthermore, the file-based graph, compared to the other two graphs, does not contain the links, such as the one between Sven and Leonie and between Sven and Eric. This is due to the limitation of not considering dependencies across files.

When we compare the call-graph-based graph with SEAL, we notice that SEAL inverts the direction of link 2 and finds two additional links (3, 4). SEAL inverts the direction of link 2 because, from a data-flow perspective. Sven's code consumes data produced only by Eric but does not supply input to Eric's code.[8] When we consider link 3, we see a bidirectional connection between Sven

[8]This is in itself is neither a drawback nor advantage of SEAL compared to the call-graph-based approaches, as the meaning of directionality depends on the use case and follows from the actual research question (e.g., where determining a link between to artifacts does not require directionality, a link between developers does if we want to determine who is using whose code).

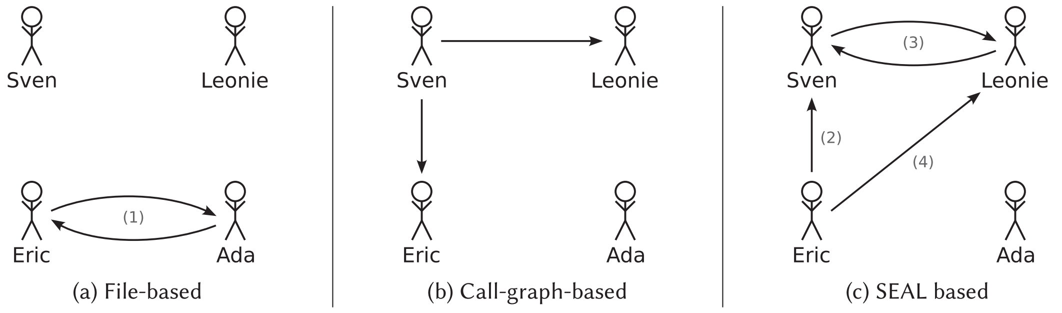


Fig. 20. Coordination requirements between authors as determined by different artifact coupling approaches.

and Leonie, capturing the fact that data from Sven are used as an input to the code of Leonie and vice versa. SEAL differentiates between a function call with and without data. In contrast, when using a call-graph approach, we find only that there is a connection between Sven and Leonie and between Sven and Eric. With SEAL, however, we now know that Eric's code does not depend on Sven's code, since no input is passed to Eric's implementation, but Leonie's code does depend on Sven's code, as her code works on input provided by Sven.

Many studies treat collaboration links between developers as undirected [41] or cannot infer a direction in links between artifacts, such as in co-changes retrieved from files that are commonly committed within one commit [33]. With SEAL, we obtain information about the direction that is based on the underlying data flows. This code-based directionality adds additional value, compared to temporal directionality [28], as it encodes who is using whose code. Parts of this information is also present in call graphs; we can infer only who is calling whose code but not, for instance, which data are passed to the function and how the returned data are used. Ignoring this information, we would not be able to notice indirect data dependencies, such as the one between Eric and Leonie that produces link 4. This link arises from the fact that Sven forwards data computed by Eric's loading function to Leonie's compute implementation. This information hints at an important coordination requirement that should not be missed, as when Eric adapts what user data are loaded, Leonie's implementation needs to handle it. Hence, coordination between Eric and Leonie is important before Eric makes a change. This kind of coordination requirement is found only with SEAL and can be found only by utilizing a whole-program inter-procedural data-flow analysis.

Integration of SEAL into other research. A good example for illustrating SEAL's merits arises from the work from Mauerer et al. [41]: In a large-scale empirical study, they explore the relationship between software quality metrics and socio-technical congruence by analyzing the alignment of social communication structures with technical dependencies. Important for our discussion is that they determine these dependencies by extracting them from the version control system: Two artifacts are related when they appear in the same commit, and two artifacts are dependent when they have static language-level dependencies, such as call-graph connections and type references. As explained in our example above, data flows carry additional information compared to file or call-graph-based approaches and thus enrich the artifact dependency networks of Mauerer et al. [41].

SEAL can not only help to determine artifact relationships, but with our author-interaction graph, approaches that analyze developer communities [28] or organizational structures [27] can profit from detailed data-flow-based interaction data, such as the work from Joblin et al. [28], who mine developer communities based on commit information and source-code structure. Specifically, they use a function-based and a committer-author-based approach to determine related authors.

```
 1 string sanitize(const string &s) {                              ▷ de8781b   Eric
 2   if (test) { return s; }                                       ▷ ea8426c   Eric
 3   return sanitizeSQLString(s);                                  ▷ de8781b   Eric
 4 }                                                               ▷ de8781b   Eric
 5 int main(int argc, char **argv) {                               ▷ c4d9b1a   Sven
 6   auto *con = driver->connect(/* credentials */);               ▷ 3e8882e   Sven
 7   auto *smt = con->createStatement();                           ▷ 3e8882e   Sven
 8   string q = "SELECT name FROM students where id=";             ▷ 3e8882e   Sven
 9   string input = argv[1];                                       ▷ 0872f49   Ada
10   string sani = sanitize(input);                                ▷ 5341f7b   Leonie
11   auto *res = stmt->executeQuery(q + sani);                     ▷ 5341f7b   Leonie
12   if (!res->rowsCount()) cout << "no records\n";                ▷ 0872f49   Ada
13   while (res->next()) cout << res->getString("name");           ▷ 0872f49   Ada
14   delete stmt; delete res; delete con;                          ▷ 3e8882e   Sven
15   return 0;                                                     ▷ c4d9b1a   Sven
16 }                                                               ▷ c4d9b1a   Sven
```

Fig. 21. A program that is vulnerable to SQL injections. Each ▷ indicates the commit that last modified each line, and right next to it is the name of the commits author.

This way, they potentially miss important non-local links between developers, as conceptually shown in our example. With SEAL, they are able to extend their information on developers with data-flow-based author interactions and expand their author network as well as remove potential spurious links.

Another example in which SEAL's commit interactions can be beneficial is in coordinating bug fixes or semantic code changes in large code bases or ecosystems. It has been reported that, for large industry code bases, even small code changes can cause severe bugs in other parts of the code [62] (e.g., a pointer can now be null, an integer variable can now have values larger than `x`, or a list that was previously sorted is now unsorted). As preventing changes is not an option (applying a bug fix that patches a vulnerability can not be delayed indefinitely), developers need to find ways to coordinate and let others know about the change. With commit-author interaction data produced by SEAL, we can easily figure out whose code could be affected by a change. Think about link 4 from Eric to Leonie that SEAL found because data produced by Eric are passed via Sven to Leonie's code. With this information, Eric can proactively contact Leonie to look at his proposed change to determine that her code works well with it.

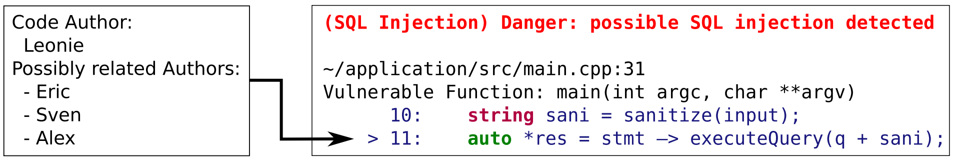
7.2 Socio-technical Data-flow Analysis

Example. Interpreting or acting on program analysis findings is a difficult task, especially in larger software projects [2]. One reason is that common program analysis tools focus on technical aspects, showing what is wrong in the code or highlighting conceptual problems [44, 45]. Typically, program analysis tools do not put their findings into a socio-technical context, ignoring the social structure around the code. Manually fitting this information post hoc onto the analysis results is cumbersome and difficult, because developers do not have this information in their heads and also because tools such as git blame provide only raw information that is not contextualized with regard to the analysis semantics. With SEAL, we build a bridge and automatically attach socio-technical information onto low-level analysis findings.

Consider the example in Figure 21 (cf. Section 4.2), where a program analysis tool, such as PhASAR, identified an SQL injection. After the tool analyzed the code, it reports that there is a possible SQL injection vulnerability at Line 11, arising from a data flow from variable `sani` (Line 10), which contains unsanitized user input. Following this report, an engineer from the company's security team can investigate the problem and finds that the recent change 5341f7b, depicted in

```
10  +   string sani = sanitize(input);
11  +   auto *res = stmt->executeQuery(q + sani);
12  -   auto *res = stmt->executeQuery(q + input);
```

Fig. 22. Commit `5341f7b` in which Leonie wants to prevent SQL injections by adding a call to `sanitize`.



```
Code Author:
  Leonie
Possibly related Authors:
  - Eric
  - Sven
  - Alex
```

```
(SQL Injection) Danger: possible SQL injection detected

~/application/src/main.cpp:31
Vulnerable Function: main(int argc, char **argv)
      10:    string sani = sanitize(input);
    > 11:    auto *res = stmt -> executeQuery(q + sani);
```

Fig. 23. A socio-technical bug report, linking the author of the vulnerable code and possibly related authors that interact through data flow with the vulnerable code location to a found SQL injection, for example, from GOKART https://www.praetorian.com/blog/introducing-gokart/.

Figure 22, introduced the offending variable in Line 10. Based on this information, Leonie can be contacted by the security engineer and asked to fix the SQL injection. However, this initial conclusion is wrong! Leonie's code did not introduce the SQL injection, since the actual problem is in the implementation of `sanitize`, where unsanitized data are leaked when running in test mode. This is likely found out only later when she starts digging into her code and the implementation of `sanitize` and refers the problem back to the security engineer.

Misattributions like this arise due to missing information and cost valuable developer time. From the point of the security engineer, the tool report was plausible, and Leonie's change seemed related, so the ticket was forwarded to her. With SEAL, this scenario could have worked out differently if the tool's initial report would be combined with socio-technical information, obtained by SEAL's data-flow analysis. By attaching the commit-interaction path of the offending call instruction for `executeQuery`, which contains all commits that had influence on the data flowing into the call, we can determine all authors that are involved. Figure 23 depicts an exemplary error message, where the SQL injection tool's finding is contextualized with socio-technical information. So, in our example, the security engineer would see the report about the SQL injection, together with the information that the two developers Leonie and Eric are involved. The security engineer would then assign both the ticket and involve Eric from the beginning.

Integration of SEAL to other research. Inspired by our example, a number of existing program analysis tools qualify to be extended with additional socio-technical information. For example, Bessey et al. present an extensive experience report [2] that details on how industrial-grade static analyzers are used to find bugs in the real world and how these tools are perceived by companies and software developers: When confronted with analysis findings, developers tend to get emotional; at the end, it is their code that is supposedly flawed. Experience shows that the more peers are involved in discussing and dealing with analysis findings, the more likely it is that someone can diagnose an error (or identify a reported error as a false positive), report on experiences of similar errors, and eventually fix it [2]. SEAL provides this additional socio-technical information. For instance, it reports the set of authors involved in a given analysis finding, which makes it easier to assign errors to the relevant developers.

Furthermore, the information produced by program analysis tools can be filtered according to the attached socio-technical information (e.g., an analysis tool used within an integrated development environment could show only the findings that actually have a socio-technical connection to the developer using it). In this vein, Harman and O'Hearn report on the difficulty of making bug reports more actionable [19]. In particular, they debunk the implicit assumption that analysis and testing technology that follow the *"ROFL" (Report Only Failure List)* strategy is enough to get engineers to fix reported bugs. Software developers, however, are never short on lists of fail and failure reports that need to be taken care of. Any additional piece of information that helps developers prioritize these todos are helpful. The report of Harman and O'Hearn underlines the importance of the following information to find and prioritize bugs: (i) *relevance*, the developer to whom the bug report is sent is one of the set of suitable people to fix the bug; (ii) *context*, the bug can be understood effectively; (iii) *timeliness*, the information arrives in time to allow an effective bug fix; and (iv) *debug payload*, the information provided by the tool makes the fix process efficient (reproduced from Reference [19]). SEAL helps to deliver such information.

The socio-technical interaction provided by SEAL may benefit scheduling code reviews, e.g., developers with a lot of data dependencies to the changed code can be suggested as reviewers, as the changes could potentially interact with their code and introduce bugs. In 2008, a Debian developer accidentally broke a random number generator in a particular version of OpenSSL with what was thought to be a fix [49]. This shows that—in practice—it can be quite difficult to assign suitable reviewers for a given pull-request. With help of information as computed by SEAL, this could possibly have been avoided by allowing peers that are familiar with this complex part of OpenSSL's code base to intervene.

7.3 Limitations

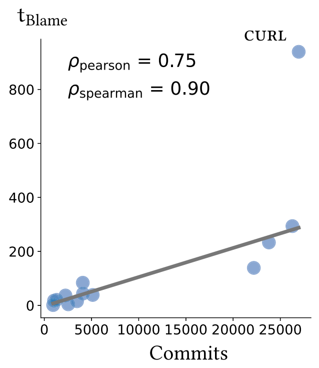
A bottleneck of using SEAL is clearly the computational cost of the underlying data-flow analysis. Table 2 depicts the overhead generated by SEAL for the projects of our study (cf. Section 5.1). We see that, for the average project, computing the blame data adds roughly two minutes. Interesting to note, computing blame information correlates with the history length of a project ($\rho_{\text{pearson}} = 0.75$ and $\rho_{\text{spearman}} = 0.90$), so projects with longer histories should expect a bit more overhead.[9] Overall, we can see that for nearly all projects the analysis time dominates the overhead. As described in Section 2.3, we tuned SEAL's analysis to be as precise as practically feasible. Specifically, we made our analysis context-sensitive, alias-aware, and inter-procedural. From our point of view, this is not too problematic, since SEAL is designed to run once to capture a full and precise picture of a given software project. Nevertheless, the underlying data-flow analysis is highly configurable in the sense that PhASAR allows one to select different helper analyses and to change each analysis' parameters to trade off precision and performance. For instance, PhASAR allows its users to choose a less precise but faster points-to analysis. Similarly, one can choose a call-graph algorithm that underapproximates information and does not resolve indirect function calls instead of the one we chose. SEAL's call-graph algorithm more expensively tries to identify potential call targets at indirect function calls to improve analysis precision, using points-to information and type hierarchies. Especially C++ developers seem to minimize the amount of indirect jumps [52] such that underapproximating call-graph algorithms may still provide enough precision for a user's needs. Through these tuning knobs, users can reduce the analysis time if (slightly) less precise results are acceptable for their setting.

Another limitation of SEAL is that the granularity of commit data are currently line-based, which is the common granularity used by git and other tools that work on git data. Previous

[9]For the history-length correlation, we observed only CURL as an outlier, which took particularly long to compute its blame data. A manual inspection of the project revealed that many files in CURL contain very old code (committed pre-2007), which, combined with the amount of commits in the history, means that the blame computation often needs to traverse a very large part of the history.

Table 2. Blame-annotation Overhead (t_{Blame}) and Analysis Time ($t_{Analysis}$) Measurements in Seconds

	LOC	Commits	t_{Blame}	$\frac{t_{Blame}}{Commit}$	$\frac{t_{Blame}}{LOC}$	$t_{Analysis}$	$\frac{t_{Analysis}}{LOC}$
BISON	591,687	26,281	293.82	1.34	0.06	14,849.20	0.03
BROTLI	34,833	1,030	17.68	0.62	0.02	614.95	0.02
CURL	195,685	26,949	939.42	6.83	0.94	47,639.50	0.24
GREP	619,598	23,794	233.14	1.09	0.04	669.76	0.00
GZIP	622,480	22,193	139.03	0.43	0.02	70.75	0.00
HTOP	25,775	2,243	36.78	1.15	0.10	313.90	0.01
LIBPNG	74,571	4,098	43.93	0.31	0.02	559.62	0.01
LIBSSH	95,235	5,126	38.74	0.65	0.03	13,469.70	0.14
LIBTIFF	88,561	3,470	15.48	0.32	0.01	6,005.81	0.07
LRZIP	19,215	935	1.27	0.03	0.00	593.82	0.03
LZ4	18,813	2,541	4.08	0.01	0.00	1,404.06	0.07
OPUS	70,267	4,077	84.02	3.48	0.20	10,379.50	0.15
XZ	38,441	1,298	20.73	1.44	0.05	9.18	0.00



These measurements are further contextualized with history size (t_{Blame}/Commit) and project size ($t_{Analysis}$/LOC).

work already demonstrated for different use cases [16, 37] that a syntax-based approach could, with a additional cost, improve the precision. With token-based blame information, as proposed by Germán et al. [16], SEAL could be even more precise.

Commit interactions based on data-flow dependencies provide a new way to infer dependencies between developers that is based on how their code interacts. However, there might be other kinds of dependencies that give rise to coordination requirements or are otherwise of interest. For example, external communication via file system or network, interactions through non-functional properties, side channels, and operating-system-level functionality could also require coordination between developers. Therefore, even if we can include data-flow dependencies with SEAL, we should still keep broadening the scope and exploring new kinds of information that is currently not available to determine developer interactions.

8 RELATED WORK

SEAL can be applied to different areas, since many techniques can profit from either incorporating repository information or detailed low-level program information. So, we discuss work that is directly related to our evaluation, and we put our work in context to related areas that can benefit from our commit analysis.

Code complexity metrics. Various code complexity metrics have been proposed [13]. Tornhill [57] discusses multiple software metrics and analysis approaches that are used in companies on real-world software projects to drive decision making and help with software maintenance. Software metrics help developers to focus on important code regions and maximize their improvement efforts. In their systematic mapping study, Varela et al. [58] categorized almost 300 different source code metrics from 226 studies into programming paradigms for which they are used. They compare how and from which systems the metrics are extracted and rank them by the number of occurrences in studies.

A close look at Tornhill [57] and Varela et al. [58] reveals that nearly all common software metrics rely only on syntactical information. Very few software metrics go beyond syntax and, even if they do, they mostly consider information that is simple to obtain, like inheritance relationships. We see a potential for improvement here by incorporating more combined analysis approaches, such as commit interactions, especially when the analyses incorporate change information as well, which can help to refine results by putting them into a historical context.

Bug prediction. Bug prediction focuses on identifying and predicting potentially buggy code locations. D'Ambros et al. [9] compare a wide range of different bug prediction approaches. Many of the approaches focus on high-level information, such as change metrics (e.g., number of revisions), source code metrics (e.g., depth of inheritance tree), and code churn, and do not incorporate lower-level information, such as data flow, which can be used to approximate the program semantics. Khatri and Singh [32] performed a SWOT analysis on *cross-project defect prediction (CPDP)*, analyzing a wide range of approaches. Interestingly, one key opportunity for improvement of CPDP that they highlight is the integration of more process metrics, such as number of developers working on a module. Our work can enable CPDP approaches to integrate process metrics that also incorporate low-level interactions between code changes or authors.

Program analysis. Program analysis techniques have been used successfully to prove specific properties about a program [12] and to detect bugs [48]. Program analysis tools, such as SpotBugs or clang-tidy, build on these techniques to catch bugs early in the development cycle. However, typical program analysis techniques, including srcML, analyze only one specific version of a program and do not incorporate version control information, which precludes detecting evolutionary problems, for example, detecting architectural decay by measuring the gradual deterioration of the coupling between classes. In addition, one could also extend already existing regression analyses by incorporating version control information directly into the analysis semantics. By utilizing both high-level repository information and data-flow information in a joined analysis, one could bridge this gap.

Socio-technical software analytics. In a special issue about software analytics, Menzies and Zimmermann highlight the importance of using analytical methods that incorporate data from real-world software projects to reason about software development processes [43]. They predict that the field will develop and benefit from more and different data sources. In the same vein, in a meta-analysis on socio-technical software-engineering research, Storey et al. [54] examine a wide range of publications to determine the current state-of-the-art and areas for further improvement. They highlight that many research papers employ a data-driven approach. So far, the used data sources (e.g., issue, bug, or commits) often do not incorporate low-level dependency information as provided by data-flow analysis. For example, the state-of-the-art in socio-technical analyses is to use function-level semantic coupling [26] or fine-grained co-edits [17] to represent interdependencies between software artifacts. Our work tries to address this shortcoming by providing a conceptual framework that combines the existing data sources with program semantics. This information could then be used to discover previously hidden socio-technical connections and coupling between developers that are invisible when only looking at syntactical information.

Software repository mining. Software repository mining focuses on gathering, modeling, and studying the data and software artifacts produced by developers during the software development process [10]. Kagdi et al. [29] state that source code changes are the fundamental unit of software evolution but also mention that current version control systems do not provide information about code semantics. This moved the initial focus of the research area to consider mostly change metadata; a shortcoming that we address with SEAL.

Change impact analysis. The goal of change-impact analysis is to determine the consequences of a change using dependency analysis techniques, such as data-flow or control-flow analyses [3].

In a survey, Li et al. [38] analyze 23 different code-based change-impact analyses and build a framework to compare them. The authors state that many approaches use traditional program analysis techniques. They further highlight that useful information for improving change-impact analysis can be obtained with repository mining techniques. For example, Kagdi et al. [30] ran their conceptual coupling analysis on the current and previous version of source code to improve their analysis precision. Lehnert [36] lists only few approaches that combine repository mining with traditional program analysis. For example, Kagdi and Maletic [31] use high-level information and dependency information separately and look whether predictions based on either of these two agree. Conceptually, SEAL is able to improve existing change-impact analysis approaches, as we combine syntactic and semantic information into one joint analysis.

In a user study, Hanam et al. [18] have shown that semantic relations computed by static analysis helped users in completing code review tasks faster compared to using only syntactic relations. Their approach uses abstract interpretation in combination with an AST-based diff to extract semantic relations from JavaScript projects to reduce unwanted "noise" in purely syntactic relations. In contrast to their approach, SEAL has information about *all* commits available during the analysis, allowing for even more control in determining which interactions are noise and which are not. Still, the results of Hanam et al. support our claims that combining static analysis with change information can be beneficial for change impact analysis.

AST-based analysis. There are tools that combine repository information with syntax information. With such light-weight syntax-based repository mining tools, most notably Boa [11], researchers can gather repository metrics and high-level code information about a wide range of different software projects. However, those do not model language semantics and do not allow us to attribute socio-technical information to the results of other more sophisticated program analyses.

Some research in this direction builds on srcML, "an infrastructure for the exploration, analysis, and manipulation of source code" [6]. srcML has been used, among other things, for type checking [46], program slicing [47], and pointer analysis [64]. However, srcML only provides an AST-based view on the code and does not come with support for more sophisticated analyses, such as a data-flow analysis; the reason is that srcML by itself does not model language semantics. For example, srcML does not run the preprocessor or model C/C++ language feature, such as overload resolution or template instantiation, which are important to infer semantics. That is why we built SEAL on top of Clang and LLVM, a industry-strength compiler framework, enabling us to combining program analysis with repository mining. In any case, syntax analysis is less precise than a data-flow analysis in determining dependencies between program parts.

Variability-aware analysis. *Variability-aware analysis* aims at efficiently analyzing variant-rich software systems [55]. The key is that, instead of analyzing all variants individually, a *variational* program representation (i.e., a program representation that retains all points of variability) is analyzed [61]. The goal is to save analysis effort by efficiently reusing analysis results across variants [60]. Variability-aware analysis is related to SEAL's approach in that it incorporates multiple variants of a software system (possibly generated by different variability implementation or configuration mechanisms) in a program analysis run. In contrast to SEAL, the goal is performance; incorporating historical and socio-technical repository information is not in scope.

9 CONCLUSION

State-of-the-art software repository analyses often do not have precise information about a program's operational semantics at their disposal, if at all, or they try to include selected information in an ad hoc manner. On the flip side, program analyses such as data-flow analysis do not have access to repository information, which restricts the interpretability of their results by excluding the socio-technical context. SEAL bridges this gap by conceptually mapping repository-specific information into the compiler's internal representation. The mapped information can be used by specialized data-flow analyses to infer relationships between commits (i.e., determine commit interactions) or by existing data-flow analyses to augment their results with repository information.

In a evaluation on 13 open-source projects, we have demonstrated that SEAL and the generated data-flow-aware repository information can be utilized to answer relevant questions in research and practice. The first part of our evaluation shows that, with SEAL, we can obtain new insights that could not have be found with existing methods that do not integrate both repository information and data-flow information. Our qualitative analysis has uncovered interesting cases, for example, where textually small changes have a far-reaching impact, which could only be pinned down by considering data flows. The second part of the evaluation demonstrates how repository information computed by SEAL can be utilized to augment existing program analyses. This allows us to put the analysis results into a socio-technical context for further interpretation, for example, by relating SQL injection vulnerability directly to the involved developers.

Overall, our evaluation demonstrates SEAL can be used to track down previously hidden interactions and to augment existing analyses with socio-technical information, which enables us to gain insights that could previously not be tracked down.

REFERENCES

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

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53 (2010), 66–75.

[3] Shawn Bohner and Robert Arnold. 1996. *Software Change Impact Analysis*. IEEE.

[4] Marcelo Cataldo and James D. Herbsleb. 2013. Coordination breakdowns and their impact on development productivity and software failures. *IEEE Trans. Softw. Eng.* 39, 3 (2013), 343–360.

[5] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. 2008. Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2–11.

[6] Michael Collard and Jonathan Maletic. 2016. srcML 1.0: Explore, analyze, and manipulate source code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 649.

[7] Kevin Crowston and James Howison. 2005. The social structure of free and open source software development. *First Mond.* 10, 2 (2005).

[8] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. 2006. Core and periphery in free/libre and open source software team communications. In *Proceedings of the Hawaii International Conference on Systems Science (HICSS)*. IEEE.

[9] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 31–41.

[10] Marco D'Ambros and Romain Robbes. 2011. Effective mining of software repositories. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 598.

[11] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 422–431.

[12] Pär Emanuelsson and Ulf Nilsson. 2008. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.* 217 (2008), 5–21.

[13] Norman Fenton and Shari Pfleeger. 1996. *Software Metrics - A Practical and Rigorous Approach*. International Thomson.

[14] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #ifdefs influence the occurrence of vulnerabilities? An empirical study of the Linux kernel. In *Proceedings of the International Software Product Line Conference (SPLC)*. ACM, 65–73.

[15] Daniel Germán, Ahmed Hassan, and Gregorio Robles. 2009. Change impact graphs: Determining the impact of prior codechanges. *Inf. Softw. Technol.* 51 (2009), 1394–1408.

[16] Daniel M. Germán, Bram Adams, and Kate Stewart. 2019. cregit: Token-level blame information in git version control repositories. *Empir. Softw. Eng.* 24, 4 (2019), 2725–2763.

[17] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. 2021. Analysing time-stamped co-editing networks in software development teams using git2net. *Empir. Softw. Eng.* 26, 4 (2021).

[18] Quinn Hanam, Ali Mesbah, and Reid Holmes. 2019. Aiding code change understanding with semantic change impact analysis. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 202–212.

[19] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE, 1–23.

[20] Lile Hattori, Gilson Pereira dos Santos Jr., Fernando Cardoso, and Marcus Sampaio. 2008. Mining software repositories for software change impact analysis: A case study. In *Proceedings of the Brazilian Symposium on Databases*. SBC, 210–223.

[21] Jincheng He, Sitao Min, Kelechi Ogudu, Michael Shoga, Alex Polak, Iordanis Fostiropoulos, Barry Boehm, and Pooyan Behnamghader. 2020. The characteristics and impact of uncompilable code changes on software quality evolution. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 418–429.

[22] Claus Hunsen, Janet Siegmund, and Sven Apel. 2020. On the fulfillment of coordination requirements in open-source software projects: An exploratory study. *Empir. Softw. Eng.* 25, 6 (2020), 4379–4426.

[23] Timea Illes-Seifert and Barbara Paech. 2008. Exploring the relationship of history characteristics and defect count: An empirical study. In *Proceedings of the ISSTA Workshop on Defects in Large Software Systems*. ACM, 11–15.

[24] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. 2011. Mining and visualizing developer networks from version control systems. In *Proceedings of the Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 24–31.

[25] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. 2017. Classifying developers into core and peripheral: An empirical study on count and network metrics. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 164–174.

[26] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. 2017. Evolutionary trends of developer coordination: A network approach. *Empir. Softw. Eng.* 22, 4 (2017), 2050–2094.

[27] Mitchell Joblin, Barbara Eckl-Ganser, Thomas Bock, Angelika Schmid, Janet Siegmund, and Sven Apel. 2022. Hierarchical and hybrid organizational structures in open-source software projects: A longitudinal study. *ACM Trans. Softw. Eng. Methodol.* (2022).

[28] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. 2015. From developer networks to verified communities: A fine-grained approach. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 563–573.

[29] Huzefa Kagdi, Michael Collard, and Jonathan Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.: Res. Pract.* 19, 2 (2007), 77–131.

[30] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael Collard. 2010. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 119–128.

[31] Huzefa Kagdi and Jonathan Maletic. 2007. Combining single-version and evolutionary dependencies for software-change prediction. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE, 17–17.

[32] Yogita Khatri and Sandeep Singh. 2022. Cross project defect prediction: A comprehensive survey with its SWOT analysis. *Innovations in Systems and Software Engineering* 18, 2 (2022), 263–281.

[33] Hassan Khosravi and Recep Colak. 2009. Exploratory analysis of co-change graphs for code refactoring. In *Proceedings of the International Conference on Advances in Artificial Intelligence (AI)*. Springer-Verlag, 219–223.

[34] Irwin Kwan, Adrian Schröter, and Daniela E. Damian. 2011. Does socio-technical congruence have an effect on software build success? A study of coordination in a software project. *IEEE Trans. Softw. Eng.* 37, 3 (2011), 307–324.

[35] Chris Lattner and Vikram S. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE, 75–88.

[36] Steffen Lehnert. 2011. *A Review of Software Change Impact Analysis*. Citeseer.

[37] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing precision and performance in structured merge. *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 367–397.

[38] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013. A survey of code-based change impact analysis techniques. *Softw. Test. Verif. Reliab.* 23, 8 (2013), 613–646.

[39] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Amaral, Bor-Yuh Chang, Samuel Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[40] 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 (2015), 44–46.

[41] W. Mauerer, M. Joblin, D. A. Tamburri, C. Paradis, R. Kazman, and S. Apel. 2022. In search of socio-technical congruence: A large-scale longitudinal study. *IEEE Trans. Softw. Eng.* 48 (2022), 3159–3184.

[42] Andrew Meneely and Laurie Williams. 2011. Socio-technical developer networks: Should we trust our measurements? In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 281–290.

[43] Tim Menzies and Thomas Zimmermann. 2013. Software analytics: So what? *IEEE Softw.* 30 (2013), 31–37.

[44] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Explaining static analysis—A perspective. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 29–32.

[45] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 532–543.

[46] Christian Newman, Jonathan Maletic, and Michael Collard. 2016. srcType: A tool for efficient static type resolution. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 604–606.

[47] Christian Newman, Tessandra Sage, Michael Collard, Hakam Alomari, and Jonathan Maletic. 2016. srcSlice: A tool for efficient static forward slicing. In *Companion Volume ICSE*. ACM, 621–624.

[48] Flemming Nielson, Hanne Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag.

[49] OpenSSLRandomNumberGeneratorBug. 2008. Predictable random number generator discovered in the Debian version of OpenSSL. Retrieved from https://en.wikinews.org/wiki/Predictable_random_number_generator_discovered_in_the_Debian_version_of_OpenSSL.

[50] Shmuel Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170.

[51] Philipp Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An inter-procedural static analysis framework for C/C++. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 393–410.

[52] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2021. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2:1–2:31.

[53] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 17–30.

[54] Margaret-Anne Storey, Neil Ernst, Courtney Williams, and Eirini Kalliamvakou. 2020. The who, what, how of software engineering research: A socio-technical framework. *Empir. Softw. Eng.* 25 (2020), 4097–4129.

[55] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.

[56] John Toman and Dan Grossman. 2017. Taming the static analysis beast. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 18:1–18:14.

[57] Adam Tornhill. 2018. *Software Design X-Rays*. Pragmatic Bookshelf.

[58] Alberto Varela, Héctor Pérez-González, Francisco Martínez-Pérez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *J. Syst. Softw.* 128 (2017), 164–197.

[59] Markos Viggiato, Johnatan Oliveira, Eduardo Figueiredo, Pooyan Jamshidi, and Christian Kästner. 2019. How do code changes evolve in different platforms? A mining-based investigation. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 218–222.

[60] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-aware static analysis at scale: An empirical study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33.

[61] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. 2014. Variational data structures: Exploring tradeoffs in computing with variability. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (SPLASH)*. ACM, 213–226.

[62] T. Winters, T. Manshreck, and H. Wright. 2020. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media.

[63] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 531–540.

[64] Vlas Zyrianov, Christian Newman, Drew Guarnera, Michael Collard, and Jonathan Maletic. 2019. srcPtr: A framework for implementing static pointer analysis approaches. *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE/ACM, 144–147.

Received 26 August 2022; revised 10 December 2022; accepted 24 January 2023