Focused Dynamic Slicing for Large Applications using an Abstract Memory-Model

Alexis Soifer, Diego Garbervetsky, Victor Braberman
[asoifer,diegog,vbraber]@dc.uba.ar
UBA, ICC, CONICET
Buenos Aires, Argentina

ABSTRACT
Dynamic slicing techniques compute program dependencies to find all statements that affect the value of a variable at a program point for a specific execution. Despite their many potential uses, applicability is limited by the fact that they typically cannot scale beyond small-sized applications. We believe that at the heart of this limitation is the use of memory references to identify data-dependencies. Particularly, working with memory references hinders distinct treatment of the code-to-be-sliced (e.g., classes the user has an interest in) from the rest of the code (including libraries and frameworks). The ability to perform a coarser-grained analysis for the code that is not under focus may provide performance gains and could become one avenue toward scalability.

In this paper, we propose a novel approach that completely replaces memory reference registering and processing with a memory analysis model that works with program symbols (i.e., terms). In fact, this approach enables the alternative of not instrumenting -thus, not generating any trace- for code that is not part of the code-to-be-sliced. We report on an implementation of an abstract dynamic slicer for C#, DynAbs, and an evaluation that shows how large and relevant parts of Roslyn and Powershell - two of the largest and modern C# applications that can be found in GitHub- can be sliced for their test cases assertions in at most a few minutes. We also show how reducing the code-to-be-sliced focus can bring important speedups with marginal relative precision loss.

Index terms—dynamic slicing, program slicing.

1 INTRODUCTION
Dynamic slicing [16] shows developers which program statements impact the value of a variable at a specific line of code when the program is executed with a particular input. The study of dynamic slicing dates back over thirty years and its results have been applied to, for example, debugging and fault localization, cohesion measurement, program comprehension and testing [10, 18, 19, 21, 25, 32]. Dynamic slicing techniques compute program dependencies analyzing the information within the execution trace generated through program or run-time environment instrumentation. Control and data dependencies of the trace statements are analyzed to identifying those that affect the slicing criterion [12, 31, 37]. Even though there are many papers that propose and optimize dynamic slicing techniques, including trace compression, only a few dynamic slicing tools are publicly available and none are reported to scale to long executions of complex modern systems.

We believe one of the main obstacles to scalability of dynamic slicers is their need to record, trace, store, and later process concrete memory references (e.g., [35]). Ultimately, those references are needed to query and get -from the slicer’s internal data structures- the statement occurrence that last defined a given memory location. However, this introduces some difficulties: there is a time overhead on the instrumented code to get the memory references, trace size grows with the unavoidable need for compression (and decompression) [30], this impacts size of internal data structures and time-cost of traversing them to get the last definitions [35].

Working at the level of concrete memory references also hinders the ability to focus just on the code-to-be-sliced, which could be responsible for only a tiny fraction of the system trace. Of course, any part of the executed code may alter the def-use relationship between statements that are in the code-to-be-sliced, so in principle memory operations of executed statements of, for instance, the library code (i.e., code that is not to be sliced) must be accounted for too, but this incurs in a performance penalty. Introducing within the memory-references paradigm a way to decouple library code is far from trivial and, to the best of our knowledge, there are no general approaches that soundly handle arbitrary library code which may involve complex memory operations that can alias current and new allocated objects. We believe the ability to define and focus on code-to-be-sliced is a must if one expects to handle modern and large framework-based applications in which most behavior is performed by either framework code or even application modules that are not relevant to the user.

We propose abstract dynamic slicing that deals with accesses of memory locations through high-level program symbols (i.e., terms) instead of memory references. In contrast to traditional dynamic slicing, trace information collected only includes just the necessary data to identify executed statements, avoiding storage of concrete memory references accessed. This technique is based on a memory analysis model (or just analysis model) that can map terms (access paths) into abstract memory locations. As slicing progresses over a program trace with no concrete memory location information, the memory model is updated and a dependency graph constructed.

This approach naturally supports diverse treatment of code: when a call to a method which is not relevant for the user (or client analysis of the slicer) is made, the code need not be instrumented nor its trace collected, simply a conservative effect can be applied to the memory model – i.e., a havoc effect. This consists in representing every possible read, write, or object allocation that could have been produced within the external method execution. The same can be done if the method relevant for the user is call-backed from
methods that are not. Indeed, this allows soundly trading precision for scalability.

In order to show that abstract dynamic slicing can be used for dealing with modern and large applications we implemented DynAbs, an abstract dynamic slicer for C#. We used DynAbs to slice some of the biggest C# applications that can be found in GitHub, Roslyn -the C# compiler- and Powershell -the Microsoft console using slicing criteria taken from tests found their GitHub projects.

The main contributions of this paper are: (1) a new sound technique called abstract dynamic slicing which uses a memory analysis model instead of real memory locations, (2) a tool, DynAbs, which implements the abstract dynamic slicing for C#, and (3) an evaluation of DynAbs that includes (i) a comparison with a state of the art tool on a common benchmark, and (ii) an evaluation showing both that abstract dynamic slicing can handle even without trace compression– slicing real test assertions in large C# applications and how removing instrumentation can sometimes accelerate analysis with marginal loss of precision.

2 ARCHITECTURE AND OVERVIEW

Our approach produces slices without having to trace all the executed code by treating symbolically the read/write effects of traced and untraced code.

Figure 1 shows an overview of the approach. We require an executable Program separated into a portion that is to be traced ($P_T$) and another that is not ($P_{NT}$). The former must be in an instrumentable format (i.e., in source code or bytecode), the latter may include compiled third-party code. An Instrumenter modifies the code-to-be-traced to record control information. The code is executed with a user-chosen Input and a Trace is produced that includes only indications of the blocks of code executed. Note that trace generation is lightweight, not recording concrete memory references, and thus, avoiding instrumentation costs and the runtime overhead of recording every access to the memory. The trace is then processed incrementally by the Analyzer to construct a Dynamic Dependency Graph (DDG) \cite{1}. The trace is conceptually transformed into a stream of statements that is processed to update both the DDG and a model that abstracts program’s memory (c.f., Memory Analysis Model). Finally, the analyzer outputs a program Slice traversing the DDG from the Slicing Criterion provided by a user.

This schema looks like classic dynamic slicing based on DDG (e.g., \cite{1}) but it solves the problem of determining the last definition of memory locations denoted by variables differently than later approaches and tools (e.g., \cite{3, 4, 12, 30, 31, 34}). Our approach differs because we obtain symbolic terms (e.g, variable names, access paths) from the source code rather than actual memory references collected at run-time.

Consider the program in Figure 2 that is to be sliced at line 3 (variable r). A traditional slicing approach would instrument the whole program in order to produce a trace that registers the updates of each memory location accessed by the program. For instance, upon execution of line 2 a tuple $(2^0, 0x8f f3, 0xef e9)$ would be recorded in a trace file to indicate that in the first execution of line 2 the memory location of variable a (0x8f f3) has the new memory allocated address 0xef e9. Later, a dependency graph would be updated using this information by first adding a node for that line for registering that the last update of a is, effectively, at line 2.

In contrast, in our approach a program execution trace stores the executed statements plus any entrance and exit to control blocks. No memory references are stored. For the example above, the trace would be: EnterMain$^0$, S2$^0$, S3$^0$, EnterF$^0$, S6$^0$, S7$^0$, EnterH$^0$, S15$^0$, S16$^0$, ExitH$^0$, S8$^0$, ExitF$^0$, ExitMain$^0$.

Informally, in our approach the processing of the trace proceeds as follows: when processing $2^0$ our slicing algorithm tells the analysis memory model that a new object should be added. After that, on reading EnterF$^0$ it tells the analysis model to add a new context for F, and with S6$^0$ that an object was created. Something different occurs when processing S7$^0$. Here, instead of processing a sub-trace corresponding to g (as in traditional approaches to slicing), the trace analyzer tells the analysis model that a non-instrumented method (i.e., code in $P_{NT}$) was executed. The model conservatively updates the part of the head reachable from b (the havoc effect, see section 4). The trace analyzer can detect a non-instrumented call because instead of receiving EnterH$^0$, it received EnterF$^0$. This implies something else: a callback method has to be processed. Hence, a conservative operation in the memory model assigns the method parameters (in this case, z), processes the sub-trace corresponding to h, and conservatively assigns its returned value to the state of the non-instrumented method g. Finally, the trace analyzer detects the exit from g when observing the statement S8$^0$ from f.

Notice that the slice can work without the need of analyzing the whole program. In particular, it can avoid tracing and analyzing a framework of library code that can represent thousand of even millions of line of code. The price to pay is precision in the resulting slice. In a traditional approach, lines 3, 7, 8, 11, 12, 15, and 16 will be part of the slice. In contrast, this approach adds the line 6 although it has not been used. In addition, lines 11 and 12 are not part of the slice (as they are part of external, untraced and unsliced code).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Abstract Dynamic Slicing Overview.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Basic example. External call is not instrumented.}
\end{figure}
In the next sections we introduce the analysis model we use to allow reasoning abstractly about the dependencies of a program execution trace.

3 POINTS-TO CONFIGURATION TRANSITION SYSTEM

We first provide a simple formalization of a (concrete) points-to configuration that comprises both the heap and the stack of a program. This allows formalizing how our memory analysis-model (presented in the next section) relates to actual memory snapshots of the program under analysis.

Terms are sequences of program symbols (and, potentially, concrete values for indexes in the case or arrays) used to access memory locations. The analysis model works with Static Terms or just Terms (when it is clear from the context), which are sequences of program symbols (i.e., access paths with no concrete values for indexes). In what follows, consider that last(t) returns the last field name of the term t, base(t) returns the same term t excluding only its last field. When t is a variable name v, last(v) = v, base(v) = e. For the sake of simplifying presentations we omit dealing with types of terms and objects (see Section 6 for some details on how they are used to improve precision). Also, and without loss of generality, we avoid modeling scalar variables (but their treatment is very similar to variables that map to memory references). (Reference) variables are modelled as fields of potentially stacked environment objects.

Concrete Points-to Configuration. Let c ::= ⟨O, E, σ⟩ be a concrete points-to configuration such that O ⊆ Oc where Oc is the universe of nodes representing dynamically allocated objects. E ⊆ O×Fields×O (which is a partial function on the two first parameters) is a set of labeled edges that represents values of object fields (i.e., (n, f, m) ∈ E means that the field f of object n points to the object m), and σ = σ0...σk is a stack of objects σi ∈ O, that stand for the current activation stack. Object σ0 is the current activation object; its fields play the role of local variables and current method parameters.

Points-to Configuration Transitions. We now define the actions that make configurations transition to other configurations. Basically, actions are associated to the following basic operations: new-object (fresh object allocation (alloc(term)), field assignment (assign(term, term)). Besides these basic Points-to operations, we need to take into account the effects of entering a method, which creates a new environment on top of stack with parameters pointing-to arguments, (enterMethod(m, args, params)), and exiting a method, that pops the environment and assigns the return value to the corresponding left hand side field of the call to the method that has finalized (exitMethod(m, lhs)). Thus, the semantics is that of the standard points-to effects of those operations. We will denote such configuration transitions as c op1 p c′ where p is the operation’s parameters.

Configuration and Action Traces. A Configuration Trace is a sequence of consecutive configuration transitions, schematically, c0 op1(p1) c1 ... opn(pn) cn generated by the program under analysis. For the sake of simplicity, let’s assume, that c0 is the empty configuration and the first operation, op1, is enterMethod(main, _, _.)

and stands for entering the main. We will call CT(P) the set of all possible configurations traces of P.

Observable Configuration-Traces. Now, Let M2S (methods to be sliced) be a subset of program methods that are of interest (i.e., the methods in the code-to-be-sliced). Given a valid points-to trace c0 op1(p1) c1 ... opn(pn) cn we are interested in formally describing what the slicer can actually observe and infer about control flow and actions given that methods not in M2S do not externalize actions. The observable trace is obtained by hiding (i.e., λ transition) all basic points-to actions executed by methods not in M2C. Entrances to methods will be then divided into four different categories and relabelled (and/or hidden) accordingly:

- enter + ++Method(m, params, args) when a method in M2S is entered from a M2S’s method call-site,
- enter + −Method(m, args) when a method not in M2S is entered from a M2S’s method call-site (e.g., a call to a non-instrumented library method),
- enter − ++Method(m, params) when a method in M2S is entered from a method that is not in M2S (e.g., a call back from a non-instrumented part of a framework),
- enter − −Method() when the a method not in M2S is indicated to enter from a method not in M2S, those entrances are actually hidden.

Something analogous is applied to exitMethod actions:

- exit + ++Method(m, lhs) when a method in M2S exits to a method in M2S and the result is stored into lhs,
- exit + −Method(m) when call-backed method-to-be-sliced exits with results in result stored in an observable left hand side term,
- exit − ++Method(m, lhs) when a method not in M2S exits to an instrumented method and the result is stored into lhs,
- exit − −Method(m) when the a method not in M2S is indicated to exit into a method not in M2S, those exits are actually hidden.

Let oCT(M2S) the set of configuration traces. Even if the configuration of those traces are not actually observable by the slicer, their projection onto action label sequences can be inferred by the algorithm. For structured programs, those observable action traces are generated by the following grammar and, as shown later, that grammar provides the basis for getting the observable action trace from a stream of actions (some of them hidden).

```
(start) ::= (++Method) | (NoStatements)
(++)Method ::= enter+Method (Statements) exit++Method (Statements) ::= (basicPAction (Statements)) | (++)Method (Statements) | (−−Method) (Statements) | ε
(−−Method) ::= enter−Method (NoStatements) exit−Method (NoStatements) ::= (−−Method) (NoStatements) | ε
(→Method) ::= enter+Method (Statements) exit−Method (basicPAction) ::= alloc | assign
```

On the other hand, the set of configuration traces also defines a new transition system, Observable Points-to Configuration LTS, for the program P with the same configurations and new transitions
for the new labels as follows: enter+++, exit++, enter−−, exit−− are the defined as the transitions Points-to Configuration LTS. For the case of enter+++ and exit−−, transitions relate the source configuration to any configuration reachable by following 0 or more hidden actions.

Following the initial example in Figure 2, and by defining M2S = ⟨main, f, h⟩ the observable action trace associated to the control-flow trace presented in Section 2 is as follows: 1. enter++Methodmain, 2. alloc(a), 3. enter+Method(x,a), 4. alloc(b), 5. enter+Methody(b), 6. enter−Methodz, 7. alloc(c), 8. assign(rv,s), 9. exit++Method, 10. exit+Method(y), 11. assign(rv,s), 12. exit++Method(r), 13. exit++Methodmain.

4 MEMORY ANALYSIS MODEL

The key point of abstract dynamic slicing is that it works symbolically with the effects of read/writes in an execution. By symbolic, we mean that the analysis of the trace works with syntactic terms instead of memory references. More concretely, a data dependency of DDG between node r and w is added when w stands for a statement occurrence that may have been the last update of one of the underlying memory locations that is denoted by a (syntactic) term t occurring in the statement associated to node r. Thus, the core functionality of the analysis model is to get the potential (last) writers of a memory location that could be currently dereferenced by term t.

Analysis Model Configuration. Like the points-to configurations, a configuration of the analysis model is, in essence, a graph. However, nodes may represent more than one concrete object of the points-to state (e.g., all objects of an array, all objects created by a method, etc.) and fields (labeled edges) might point to more than one abstract object to handle the fact that one may not be able to precisely determine where terms point to when symbolically tracking memory updates. That is, let m := ⟨O, Oa, λ, Ė, ˆd⟩ be a configuration of the analysis model where O ⊆ Oa is a set of abstract objects, some of them Oa ⊆ O are marked as singletons (the others are called regions), Ė ⊆ O × Field ∪ {?} × O represents the connections between abstract objects. Connections may be via field names or a special wildcard symbol ?, which underspecifies the field responsible for the connection. λ ⊆ O × Ė is an equivalence relation between model objects used to indicate that those model objects may be denoting the same concrete object. Sequence ˆd denotes abstract stacks of abstract activation objects.

Identifying singleton abstract nodes enables a more precise treatment of updates as will be indicated later on. A key observer operation of the Analysis Model is get which given a term t returns the set of all abstract objects that are reached by traversing the transitions in Ė by following the sequence of fields in t starting at σ0 via matching transitions (or wildcard transitions) and taking λ related objects at each navigation step. Note that get(e) = ˆΔ0.

Sound Abstraction. To define which concrete points-to configurations are modelled by an analysis model configuration we use a denotation function δ. Given c := ⟨O, E, σ⟩ a concrete configuration, m := ⟨O, Ė, λ, ˆd⟩ a configuration of the analysis model, and a function δ : Ė × O → 2O, we say that m soundly abstracts c according to δ (noted as abs(m, c, δ)) if the following hold:

1. ∀o ∈ O, δ(o) = 1
2. ∀o1, o2 ∈ O, δ(o1) ∩ δ(o2) = ø implies (o1, o2) ∈ λ
3. ∀o ∈ O, δ(o) = ø implies (o, o) ∈ δ(o)
4. ∀o ∈ O, ∀o′ ∈ δ(o), ∀o′′ ∈ O, o′′ → o′′′ implies ∃o′′′′ ∈ δ(o′′′) ∧ (o′′′′) ∈ λ ∧ (o′′′′, o′′′) ∈ λ
5. |σ| = |δ| = k and ∀i. |σi| = δ(σi)

The first property says singleton objects map to exactly one object. The second and third, that concrete objects are abstracted by exactly one λ-partition of abstract objects. The fourth property states that navigation through fields is preserved in partitions (although a wildcard field can be used). The fifth, that the denotation function should maintain the correspondence between concrete and abstract local variable environments. Finally, the type of abstraction object should be compatible with those of denoted objects. Given two denotation functions δ and δ′, we say that δ′ extends δ (δ ⊆ δ′) iff dom(δ) ⊆ dom(δ′) and ∀a ∈ dom(δ), (δ(a) ⊆ δ′(a)).

Analysis-Model Update. The slicing algorithm we present maintains a current analysis model configuration that, after processing an action occurrence, is an abstract version of the concrete points-to configuration reached by the application at that point (which is not explicitly informed to the algorithm). Given the actions on the observable concrete points-to configurations LTS presented in the previous section, analysis model features the corresponding actions (m → m′) and they abstractly mimic the evolution of the concrete points-to configuration by just knowing the observable actions with (static) terms. This is done because, while processing the stream of statements of the trace, a key invariant is that there exists a denotation function between the analysis model and the current (and unknown in detail) concrete points-to configuration². Thus, we design these corresponding operations to extend denotation functions. More formally, the memory model “API” must guarantee that for any operation op and its abstract counterpart

²This also implies simulation and thus the existence of access paths mentioned by terms in the analysis model.
Focused Dynamic Slicing for Large Applications using an Abstract Memory-Model

Operation | Postcondition
--- | ---
m_init(t) → m' | m' is an empty analysis model
m_get(t ≈ tern) → O | O is the set of all abstract objects reached by traversing the transitions in E by following the sequence of fields in t starting at s0 via matching
m_allocate(t ≈ tern, n) → DDGGnode = m' | m' is like m but for all abstract objects b ∈ m.get(base(t)), o ∈ m' get(b,last(t)), where o is a new singleton denoting the newly allocated object. n is added to the set lastdef[last(t)] of lhs objects. If b happens to be just one analysis object marked as singleton whatever lastdef[last(t)] mapped to is replaced by (m) and o ≡ m.get(b,last(t)) (strong update).
m_assign(t, t', tern, n) → DDGGnode = m' | m' is like m but now for all abstract objects b ∈ m.get(base(t)), t ∈ m.get(t), t' ∈ m.get(b,last(t)). If m.get(b) happens to be just one analysis object marked as singleton then the new edges are subject to read and write.
m_enter→Method(p, d) → tern, n : DDGGnode = m' | Pushes new local variables environment into the stack (σ denotes the new stack in m') and assigns arguments d to formal parameters names p of the pushed environment object (i.e. m' get[p][1] = m.get(d[1]) for all t). It records (the call-site) as the last definition node for formal parameters p.
m_exit→Method(lhs) → tern, n : DDGGnode = m' | Pops m's current environment σ0 and assigns σ0, lhs (σ denotes the new stack in m) to a distinguished environment field σ0, rhs from the previous environment. This assignment follows the same logic for determining if a strong update is possible.
m_enter→Method(a) → tern, n : DDGGnode = m' | Let m(a) all the reachable obj. in m from those terms. m' adds to m a new region r (with a ± annotated self-loop transition). r is ρ-equivalent to every object in m(a). Also, r.LastDef[?'] = (n) and, m' has a new environment on top in such that m' , σ0, UpdatedHeap > r.
m_exit→Method(lhs) → tern, n : DDGGnode = m' | Pops current environment σ0 and assigns σ0, lhs to the special field σ0, UpdatedHeap from the previous environment.
m_enter→Method(p) → tern, n : DDGGnode = m' | Pushes a new environment σ0 featuring formal parameters p of the call-back method that points to the caller region, i.e., m' get(p) = m get[UpdatedHeap] using previous environment.
m_exit→Method(n) → DDGGnode = m' | Pops the current environment and m' get[UpdatedHeap] is ρ-equivalent to m get(rhs) using previous environment. Also, Vo ∈ m' get[UpdatedHeap]: n ∈ o.LastDef[?'].

Table 1: Analysis Model Interface

| Term | Description |
|---|---|
| m.exit++Method | Pops the current environment and m' get[UpdatedHeap] is ρ-equivalent to m get(rhs) using previous environment. Also, Vo ∈ m' get[UpdatedHeap]: n ∈ o.LastDef[?'].
| m.get(t ≈ tern) | O is the set of all abstract objects reached by traversing the transitions in E by following the sequence of fields in t starting at s0 via matching
| m_allocate(t ≈ tern, n) | m' is like m but for all abstract objects b ∈ m.get(base(t)), o ∈ m' get(b,last(t)), where o is a new singleton denoting the newly allocated object. n is added to the set lastdef[last(t)] of lhs objects. If b happens to be just one analysis object marked as singleton whatever lastdef[last(t)] mapped to is replaced by (m) and o ≡ m.get(b,last(t)) (strong update).
| m_assign(t, t', tern, n) | m' is like m but now for all abstract objects b ∈ m.get(base(t)), t ∈ m.get(t), t' ∈ m.get(b,last(t)). If m.get(b) happens to be just one analysis object marked as singleton then the new edges are subject to read and write.
| m_enter→Method(p, d) | Pushes new local variables environment into the stack (σ denotes the new stack in m') and assigns arguments d to formal parameters names p of the pushed environment object (i.e. m' get[p][1] = m.get(d[1]) for all t). It records (the call-site) as the last definition node for formal parameters p.
| m_exit→Method(lhs) | Pops m's current environment σ0 and assigns σ0, lhs (σ denotes the new stack in m) to a distinguished environment field σ0, rhs from the previous environment. This assignment follows the same logic for determining if a strong update is possible.
| m_enter→Method(a) | Let m(a) all the reachable obj. in m from those terms. m' adds to m a new region r (with a ± annotated self-loop transition). r is ρ-equivalent to every object in m(a). Also, r.LastDef[?'] = (n) and, m' has a new environment on top in such that m' , σ0, UpdatedHeap > r.
| m_exit→Method(lhs) | Pops current environment σ0 and assigns σ0, lhs to the special field σ0, UpdatedHeap from the previous environment.
| m_enter→Method(p) | Pushes a new environment σ0 featuring formal parameters p of the call-back method that points to the caller region, i.e., m' get(p) = m get[UpdatedHeap] using previous environment.
| m_exit→Method(n) | Pops the current environment and m' get[UpdatedHeap] is ρ-equivalent to m get(rhs) using previous environment. Also, Vo ∈ m' get[UpdatedHeap]: n ∈ o.LastDef[?'].

5 ABSTRACT DYNAMIC SLICING ALGORITHM

Algorithm 1: Start Process

**Input**: :P::Program, M2S:: Set[MethodNames], i:: Input, c:: Criterion

**Output**: Slice

1. global ddg ← create dynamic dependency graph
2. global model ← create analysis model
3. global statActionStream ← StatActionStream(execute(instrument(p, M2S), i))
4. s ← peek(statActionStream) // look ahead
5. switch s.actionType do
6. case enter ++Method do
7. Process ++Method() do
8. case enter ++Method do
9. ProcessNoStatements
10 return ddg.slice(c)

In Algorithms 1 through 7 we show the abstract slicing algorithm. It works a grammar directed translation to transduce the visible points-to actions into analysis model actions. It adds a couple of concerns as well: it identifies and categorizes some of the entrance and exit to methods to define which grammar production to process, it also updates the DDG. We omit the treatment of control dependencies as it is an orthogonal concern to the technique. The implementation uses a stack of DDG nodes associated to control
Algorithm 2: Process++Method

```java
1 s ← consume(statActionStream) // It must be an enter++Method (those are generated by the filter pre-processing) and carry all necessary information about parameters and arguments
2 n ← UpdateDDG(s, s.args.uses) // (potentially) new node into the DDG that depends on last def of args
3 model.enter ++Method(s.arity, s.params, n)
4 // model is notified
5 ProcessStatements()
6 s ← consume(statActionStream) // It must be a return statement that will be treated as an exit++Method
7 l ← consume(statActionStream) // Thus, this must be the lhs assignment
8 n' ← UpdateDDG(l, rv) // “rv” is a distinguished identifier the model use for the return value
9 model.exit ++Method(l.arity, n') // update the model to reflect it has exited
10 return
```

Algorithm 3: Process+-Method

```java
1 s ← consume(statActionStream) // It must be an enter+-Method
2 n ← UpdateDDG(s, s.args.uses)
3 model.enter +-Method(s.arity, s.params, n)
4 ProcessNoStatements()
5 s ← consume(statActionStream) // a statement different from method enter (no call back) is at the peek of the stream. This indicates the flow has returned to the last instrumented part that called the processed un-instrumented method
6 l ← consume(statActionStream) // Thus, this is the lhs assignment for the return value of the the method not in M2C (i.e. uninstrumented)
7 n' ← UpdateDDGwithNode(l, n) n is the DDG node standing for the call to the +Method //
8 model.exit +=Method(l.arity) // update the model to reflect it has exited
9 return
```

Algorithm 4: Process+-Method

```java
1 s ← consume(statActionStream) // It must be an enter method
2 n ← UpdateDDG(s, s.args.uses)
3 model.enter +=Method(s.params, n)
4 ProcessStatements()
5 s ← consume(statActionStream) this is the return from the call back
6 model.exit +=Method()
7 return
```

Algorithm 5: ProcessStatements

```java
1 s ← peek(statActionStream)
2 while (s.type = return) do
3   switch s.actionType do
4     case assignment do
5       n ← UpdateDDG(s, s.uses)
6       model.assign(s.lhs, s.rhs, n)
7       s ← consume statActionStream
8     case object creation do
9       n ← UpdateDDG(s, 0)
10      model.alloc(s.lhs, n)
11      s ← consume statActionStream
12     case enter++Method do
13       Process++Method()
14     case enter+-Method do
15       Process+-Method()
16   return
```

Algorithm 6: ProcessNoStatements

```java
1 s ← peek(statActionStream)
2 while (s.type = MethodEnter) do
3   Process+-Method() // call-back detected
4   return
```

Algorithm 7: Update DDG

```java
1 Input : s :: Statement, uses :: Set[Term]
2 dd ← 0
3 for each t ∈ uses do
4   objs ← model.get(base(t))
5   dd ← dd ∪ (lastDef[o][last(t)] | o ∈ objs )
6 n ← ddg.addNode(s, {cd})
7 return n
```

statements and actions that is subject to the parsing and transducing phase later on. More concretely, StatActionStream is the function that yields a stream of statements modified to feature also some of the categories of enter/exit method actions underlying the control flow followed by the application. StatActionStream can be regarded as the composition of two filters: one uses compilation infrastructure to convert the control-flow trace yielded by (partial) instrumentation into a stream of statements. Those statements include compilation information like parseable access path that play the role of static terms. Statements, by demand, yield which terms are being read and which term is being defined. We also assume that invocation statements and statements that stand for the entry point of a method can be queried for the name and number of arguments of the method being called or, respectively, entered (s.method).

The other filter, has as purpose to replace some of the enter/return method statements with higher-level actions using the following logic: when an invocation statement s is followed by a method enter l such that s.method = l.method then an enter++Method with arguments and parameters replace both statements. When,
s.method and l.method are different then enter++Method with the invocation arguments replaces the invocation statement \( ^5 \).

The other possible combinations are not detected by this filter and are actually detected by the algorithm while parsing the statement action stream by following the grammar presented previously. enter++Method: is the case during NoStatements processing when method enter occurs in the trace (i.e., an enter without a call has happened), exit++Method during Statements processing when a return appears and Statements was executed from the context of a ++Method processing, exit++Method when a return statement arrives while processing Statements in the context of processing a ++Method, and exit++Method, during NoStatements processing when a statement different from method enter occurs in the trace.

Method UpdateDDG deals with dependencies. As mentioned, the approach handles data-dependencies using the analysis model. In lines 2-4 of Algorithm 7, abstract objects are obtained from the analysis model, and are used together with the last field of each term to collect data dependencies. Finally (line 5), adds a node to the dependency graph with the collected dependencies.

6 IMPLEMENTATION

We developed a tool, DynAbs, that implements the abstract dynamic slicing algorithm presented above for core features of the C# language. We discuss in this section some key design decisions aiming at performance.

Dynamic Program Dependency Graph. We implement a graph where nodes represent statement occurrences and edges represent data and control dependencies between executed statements. To avoid a linear growth of this structure we follow the solution proposed in [1] that only creates a new node when a statement occurrence does not share the same dependencies as a previous occurrence (a subsumption of transactional dependencies).

Analysis Model It is worth mentioning that as \( \lambda \) partitions the set of abstract nodes, we use Disjoint-Set Union data structure [29] to summarize and quickly access information in partition representatives that would be spread all over the nodes of the partition otherwise (rendering imprecision also a factor for being slower [6]). Besides, when created, objects are actually annotated with type using information what the compilation infrastructure knows about statements in the program. Then, for model operations that receive terms as input and need to find all possible model objects that need to be queried or modified, the type information in the terms passed as parameters is used to filter out objects whose annotated type is not compatible. In fact, partition representatives stores information qualified by types.

7 EVALUATION

Now, we describe our experimental setup and discuss the evaluation results. The general aim of the evaluation is to assess if DynAbs can slice large applications with reasonable precision and performance. Given that existing slicing tools cannot process such applications and that manual slicing of large applications is extremely challenging, error prone and laborious, we do not perform an an evaluation of precision in absolute terms, i.e., against a ground truth. Instead, we compare precision against the state of the art under conditions in which this comparison is possible, and then evaluate relative precision loss in large applications. In summary, to evaluate DynAbs, we answer the following research questions:

RQ1: To what extent is DynAbs precise when compared to the state-of-the-art in dynamic slicing?
RQ2: Can DynAbs slice large apps. using real slicing criteria?
RQ3: To what extent does DynAbs lose precision (and gain performance) when non-traced code is increased in large applications?

7.1 Experimental Setup

To answer RQ1, we compared DynAbs with the state-of-the-art dynamic slicer, Javaslicer [12]. Note that the more recent Java slicer, Slicer4J, supports the latest Java features that Javaslicer does not but has worst performance [3]. Also note that there are no publicly available slicers for C#.

To compare a Java and a C# Slicer, we require a benchmark with programs that can be written in syntactically (and semantically) equivalent versions for both languages. We chose Olden [6, 24] in its Java version, J0lden [7], and we manually wrote its version for C# preserving line per line, the syntactic structure of the programs. Key in the choice of Olden is that programs do not contain calls to library, thus, translation effort is contained and controlled, and slices can be completely analyzed. Note that, olden has been used as a benchmark in numerous studies [13, 14, 20, 27]. J0lden consists of a collection of 10 programs that includes small and medium-sized scientific code (bh and em3d), process simulations (health and power), graph optimization routines (mst and tsp), graphics utilities (perimeter and voronoi), a sorting routine (bisort) and a toy tree benchmark (treeadd).

As slicing criteria, we selected the lines of code that contained variables storing the final result values calculated during the executions. If one of these results was an instance of a complex element (e.g., class, struct) then we selected the instance and its fields. In the case of arrays, we selected the array and its first element. To select program inputs, consider the fact that the Olden programs take as input the size of the problem to be solved (e.g., number of cities for the travelling salesperson problem) and then randomly pick an instance of that size. We selected minimal sized inputs to facilitate comparing slice sizes, and fixed a random instance with that size to allow reproducibility.

To answer RQ2 and RQ3, we chose Roslyn [9] and Powershell [23]. Roslyn, the .NET Compiler Platform, is the biggest C# application on GitHub. It is a set of open-source compilers and code analysis APIs for .NET languages. On the other hand, Powershell, the Microsoft console, is the highest ranked C# application with more than 500K LoC. To identify realistic slicing scenarios we chose as slicing criteria assertions in existing test cases and selected the classes under test as the code-to-be-sliced, havoc-ing the rest of the code. We compared precision and performance of computing these slices against slicing for the same criteria but with an extended instrumentation of the code.

More specifically, for Roslyn we did the following. The core of compilers project in Roslyn (roslyn/src/Compilers) consists of three main folders. The first contains generic implementations and base classes for every .NET language (core, 83K LoC). The other
two are the VisualBasic and CSharp specific implementations, with 206K LoC and 230K LoC respectively. We chose, the largest of the these two (i.e., ./Compilers/CSharp).

The test folder for the C# compiler code (i.e., ./.CSharp/Portable) is loosely structured according to the roslyn official pipeline architecture [9] which describes four APIs: Syntax Tree, Symbols, Binder and Flow Analyzer, and Emit. We selected one relevant test class for each of the APIs except for the Binder and Flow Analyzer API for which we chose two test classes (one for the Binder and one for the Flow analysis). The classes selected were: StatementParsingTests.cs, TypeTests.cs, BindingTests.cs, FlowTests.cs, and CompilationEmitTests.cs. For each of the classes we randomly selected for each test class, 20 tests.

Establishing the code-to-be-sliced required determining what code might be relevant in a debugging scenario when a tests fails. We took a syntactic approach based on the names of source code folders for this. We selected from the ././CSharp/Portable folder any subfolder that has a name that corresponds to the API under test. For instance, for the syntax tree API tests taken from StatementParsingTests.cs, we instrumented both the tests and all the classes in the ././Portable/Syntax folder. In Table 3, column Inst. Code indicates the percentage of instrumented code for each test class with respect to the total code in ././Portable. We then computed slices for every assertion in every test. In addition, as an extension to the previous instrumentation we instrumented all of the code in ././Portable and generated the slices for the same criteria as before to allow comparing differences in terms of performance and precision.

In PowerShell, tests can be of two kinds: based on external scripts or using xUnit test framework. Only the latter are written in C# and are amenable to slicing using DynAbs. We used all the tests in ././test/xUnit/csharp which, as documented, contains all the C# tests for the PowerShell Core project. To select the code-to-be-sliced we inspected each test (both code and assertions) and determined the classes it referenced directly. We then generated slices for each assertion in each test. To produce the additional set of slices with an extended instrumentation of the code, we selected for instrumentation folder ./src/System.Management.Automation which subsumes all of the previously instrumented classes. An important note is that both Roslyn and Powershell code include some advanced C# language features that the DynAbs instrumentation does not support. To handle these cases, manual intervention of the instrumented code was required in a few specific locations.

7.2 Results

RQ 1. Table 2 shows for each subject, in the first three columns, the number of different slicing criteria that were used and the average size of executions (measured in statements) and also the average number of unique statements (or lines) that were covered in the executions. The table also shows for DynAbs and Javaslicer the number of statements in the resulting slice, the relative size of the slice and the time to compute it.

In BiSort, Perimeter, TreeAdd, and TSP, there are no array structures (nor calls to external libraries). This means that, by construction, DynAbs slices are sound and precise, assuming no implementation bugs. However, in BiSort, Javaslicer produces a slice that is smaller but unsound due to previously reported [3] unsound treatment of recursion and return statements dependencies.

The remaining 6 subjects include arrays which are a source of imprecision for DynAbs. However, DynAbs slices were smaller in 4 of them (BH, Em3d, Health, and Voronoi). Javaslicer’s additional imprecision in these 4 subjects is related to its inaccurate modelling of control dependencies. Finally, in MST and Power, DynAbs imprecision with arrays leads to a 14% and 5% increment in slice size. Note that although DynAbs shows improved performance over Javaslicer, such comparison should not be considered relevant as the choice of slice tasks with minimal sized inputs is biased towards DynAbs (smaller inputs means smaller execution sizes). Indeed, note that in Power, which has a significantly larger execution size than the rest, trace compression of Javaslicer starts to show better performance than DynAbs.

In summary, only in 3 out of 10 cases did DynAbs produce larger slices than Javaslicer. One of these (BiSort) is due to unsoundness of Javaslicer. Consequently, only in 2 out of 10 cases were DynAbs slices less precise than Javaslicer.

| Subject | Criteria | ES | Stmts. | SS | SS (%) | t (s) | SS | SS (%) |
|---------|----------|----|--------|----|--------|------|----|--------|
| BH      | 6        | 3626| 356    | 120| 34     | 2.1  | 141| 40     | 9.5    |
| BiSort  | 3        | 347 | 115    | 60 | 52     | 0.4  | 57 | 50     | 9.7    |
| Em3d    | 5        | 444 | 146    | 55 | 38     | 0.5  | 66 | 45     | 9.0    |
| Health  | 3        | 1377 | 87    | 29 | 16     | 0.7  | 36 | 19     | 9.0    |
| MST     | 1        | 802 | 187    | 111| 59     | 0.4  | 97 | 52     | 8.0    |
| Perimeter| 2       | 113 | 111    | 28 | 25     | 0.3  | 40 | 36     | 8.5    |
| Power   | 4        | 22746 | 296   | 201| 68     | 39.3 | 192| 65     | 10.8   |
| TreeAdd | 1        | 59  | 44     | 20 | 45     | 0.3  | 20 | 45     | 9.0    |
| TSP     | 5        | 665 | 147    | 57 | 39     | 0.5  | 59 | 40     | 8.6    |
| Voronoi | 6        | 1490| 319    | 163| 51     | 0.9  | 177| 56     | 9.0    |

Table 2: Olden results. ES: average execution size in # of statements, Stmts.: average number of unique statements that can be sliced, SS: average number of statements in the slice, SS(%)=SS/Stmts., and t is the average slice computation time in seconds.

RQ 2. The first columns of Table 3 show how DynAbs performed when slicing for assertions in real Roslyn and PowerShell tests. For each test class we show the number of criteria for which slices were produced, the average execution size (in number of statements) and the average number of unique statements (lines of code) that can be sliced. We report on the size of the resulting slice and the reduction that the slice represents over the total number of unique statements. We also report the average slice computation time after compilation. This is because, due to the size of the projects, compilation is an order of magnitude larger (in the order or minutes) than the execution, tracing plus analysis time that we report. Finally, we report on the number of calls to external code, that is calls from the instrumented code to non-instrumented (and havoc-ed) code. Recall that these external calls are not traced nor analyzed, and may represent complex behaviour that contains many more calls to non-instrumented code.

Results show that DynAbs took on average per slice less than a few seconds for all but two classes (TypeTests and SessionState). Within the various slices computed for each class there was little variation in performance. In fact, no slice for tests other than from
Table 3: Results for Roslyn and Powershell. Criteria: number of slicing criteria used, ES: average execution size in # of statements, Stmts: average number of unique statements that can be sliced, SS (SS'): average slice size in number of statements and SS(%): SS / Stmts. (SS' / Stmts.), t: average slice computation time in seconds, Inst. (%): percentage of instrumented compared to that instrumented in the "Extended Instrumentation", and Speedup and Prec. loss: the slicing speedup time and precision loss percentage of the less instrumented code compared to the slicing of the code with extended instrumentation.

| Test class files (.cs) | Criteria | ES | Stmts. | SS (%): SS / Stmts. (SS': SS / Stmts.) | t (s) | Inst. (%) | Speedup | Prec. loss(%) | ES | SS' | t (s) |
|------------------------|----------|----|--------|----------------------------------------|------|-----------|---------|----------------|----|------|-------|
| Roslyn                 |          |    |        |                                        |      |           |         |                |    |      |       |
| StatementParsingTests  | 554      | 19270 | 1228 | 416 | 33.85 | 3.8 | 9.02 | x1 | 0.00 | 19338 | 416 | 3.9  |
| TypeTests              | 171      | 49211 | 3982 | 1586 | 39.84 | 23.9 | 35.85 | x2 | 0.00 | 76391 | 1586 | 53.2 |
| BindingTests           | 67       | 3176  | 974  | 258 | 26.45 | 1.1 | 14.41 | x93 | 0.78 | 105219 | 256 | 101.7 |
| FlowTests              | 40       | 667   | 388  | 104 | 26.76 | 0.4 | 4.53 | x218 | 0.00 | 106715 | 104 | 82.6 |
| CompilationErrorsTests | 99       | 3582  | 887  | 280 | 31.58 | 1.6 | 6.32 | x81 | 0.00 | 105464 | 280 | 128.2 |
| Powershell             |          |    |        |                                        |      |           |         |                |    |      |       |
| FileSystemProvider     | 13       | 14410 | 633  | 13  | 2.00 | 2.3 | 4.89 | x91 | 0.00 | 151088 | 13  | 214.3 |
| MuhSnaphinInfo         | 1        | 96    | 104  | 6   | 5.77 | 0.1 | 0.28 | x1 | 0.00 | 557    | 6   | 0.1  |
| NamedPipe              | 4        | 209   | 156  | 5   | 2.89 | 0.1 | 14.88 | x1 | 0.00 | 299    | 5   | 0.1  |
| CorePsPlatform         | 1        | 9     | 13   | 2   | 15.38 | 0.1 | 3.30 | x3 | 0.00 | 9      | 2   | 0.1  |
| PowerShellAPI          | 3        | 1728  | 1066 | 491 | 46.06 | 0.8 | 4.16 | x981 | 0.00 | 225696 | 491 | 825.9 |
| PSConfiguration        | 286      | 312   | 171  | 44  | 25.65 | 0.1 | 0.10 | x1 | 7.32 | 642    | 41  | 0.1  |
| Binders                | 1        | 12    | 18   | 4   | 22.22 | 0.1 | 4.00 | x1 | 0.00 | 12     | 4   | 0.1  |
| PSObject               | 15       | 496   | 299  | 78  | 25.97 | 0.1 | 1.19 | x43 | 0.00 | 8456   | 78  | 4.3  |
| ExtensionMethods       | 1        | 98    | 76   | 21  | 27.63 | 0.2 | 3.72 | x6 | 600.00 | 3290   | 3    | 1.0  |
| PSVersionInfo          | 1        | 202   | 149  | 11  | 7.38  | 0.1 | 0.17 | x1 | 0.00 | 202    | 11  | 0.1  |
| Runspace               | 9        | 2536  | 1130 | 548 | 48.48 | 0.9 | 8.37 | x38 | 0.00 | 105198 | 548 | 34.1 |
| SecuritySupport        | 1        | 41    | 48   | 20  | 41.67 | 0.1 | 1.33 | x1 | 0.00 | 104    | 20  | 0.1  |
| SessionState           | 159105   | 6217 | 4    | 0.06 | 165.2 | 67.98 | x1 | 0.00 | 179189 | 4    | 219.9 |
| Utilities              | 45       | 20    | 9    | 5    | 53.60 | 0.1 | 0.40 | x2 | 0.00 | 1619   | 5   | 0.2  |
| WildcardPattern        | 13       | 6     | 10   | 2   | 16.91 | 0.1 | 0.22 | x1 | 0.00 | 12     | 2   | 0.1  |

TypeTests and SessionState took over than 5 seconds. Slices for TypeTests had an average of 24s, with a maximum time of 56s. The only slice for SessionState took 165s. Slice size shows significant reduction in many cases (up to 98%) although in some cases reduction was less than 50%.

In summary, DynAbs was able to slice most real tests for Roslyn and Powershell in a few seconds (over 1300 slices). Only few slices took (less than 2%) up to 1 minute, and only one took 2.75 minutes.

RQ 3. The rightmost columns of Table 3 show the results of slicing the same files as in RQ2 but with additional information extracted from an execution in which there is more instrumented source code. As before, we report the average execution size (in number of statements). We also report on the size of the slice and report the average slice computation time. As in RQ2, we report computation time after compilation.

Column Inst (%) shows the percentage of code that was initially instrumented (the code-to-be-sliced) with respect to the total code instrumented (and for which the rightmost columns report on). Note that for all test classes, the original instrumentation is less than 10% of the extended instrumentation. In other words, the extended instrumentation adds a significant amount of additional code that is traced and analysed by DynAbs. The additional instrumentation can also be observed when comparing the columns for execution size. The execution sizes for the extended instrumented code can grow over two orders of magnitude (e.g., FlowTests, CompilationErrorsTests, PowerShellApi).

The Speedup column in Table 3 shows the factor by which performance is improved in the original (less instrumented) slicing computations when compared to the slices computed with more instrumented code. As expected, classes in which execution size is originally significantly smaller have major speedups (x10 to x981).

The Precision Loss column captures, proportionally, how many statements were added to the slice computed with less instrumentation but that thanks to the additional instrumentation were actually not true dependencies of the slice criteria. In other words, it is the average slice size for code-to-be-sliced instrumentation divided the average slice size for the extended instrumentation, minus 1.

Results show that for Roslyn, additional instrumentation does not result in increased precision. Indeed, the precision loss for only instrumenting the code-to-be-sliced (and no additional code, abstracted away using havocs) is on average marginal. Only 12 of the 171 slices computed for TypeTest lost precision and their worst case was 0.2%, and only 54 slices computed for BindingTests lost precision and their worst case was 2%. On the other hand average speedups by reduced instrumentation are achieved in 4 out of 5 case studies, 3 of which are very significant (x81 to x218).

Speedup results for Roslyn show as expected that the less instrumented code based has significant speedups when the extended instrumentation makes execution sizes larger. In three 3 out of 5 case studies, the three with the largest increment of average execution sizes, the speedup was x81, x93 and x218.

For Powershell, additional instrumentation only led to improved slices in two of the test classes (PSConfiguration and ExtensionMethods). In other words, the precision loss for only instrumenting the code-to-be-sliced (and no additional code) is on average marginal for 13 out of 15 classes. PSConfiguration had an average precision loss of 7.32% and the only slice for ExtensionMethods a
precision loss of 600%. Zooming in on PSConfiguration, only 82 slices of the 286 had a precision loss, averaging 175% and with a worst case of 400%.

Speedup results for PowerShell slices also show that less instrumented code based has significant speedups in particular when execution size increase in the more instrumented code base. For 8 out of 15 test cases, there was no speedup (nor precision loss). In these (e.g., NamedPipe) the execution size of the less instrumented code base is not significantly less than that of the more instrumented version, thus the lack of performance increase. However, there are 5 out of 15 test classes for which there is a speedup with no precision loss at all. PSVersionInfo, that has a significant precision loss also has an important speedup. PSConfiguration’s has no average speedup but when looking only at the average speedup of the 82 slices that incurred in a 175% precision loss, their average speedup is x2.9.

In summary, keeping instrumentation focused on the code-to-be-sliced of real Roslyn tests did not lose precision and achieved an average speedup of at least x2 in 4 out of 5 test class files, with significant increases for 3 out 5 (greater than x81). On the other hand, only 83 out of 395 slices taken from real PowerShell tests exhibited significant precision losses when keeping instrumentation limited to the code-to-be-sliced; the rest had no precision loss whatsoever. Average speedups were achieved in 40% of the test classes and for very large run-time overheads.

The use of a static slice might render such sort of approaches too imprecise, for instance, when dealing with large classes hierarchies and rich inputs.

Besides compression [12, 30, 31] there exist a few other approaches to deal with size. [38] works with user-delimited part of the execution. Even approaches based on record & replay technology like [32], that collect memory references at replay time, still require users to keep small "buggy regions" to avoid incurring in large run-time overheads.

In statistical program slicing [28] control and data-flow information is selectively collected employing a hardware extension in order to compute an observed program dependency graph. This graph is formed by nodes which represent static instructions and arcs represent control or data dependencies in any of the monitored executions. This approach deals with the scalability problem of tracing runtime at the cost of reducing accuracy (i.e., missing some statements). ORBS [5] is based on a different notion of slice (observational), thus it avoids the instrumentation of the program.

The cost of this approach is in the need of multiple compilation and executions. Precision is sacrificed when insufficient iterations are performed. It is worth mentioning [3] in which, based on taint analysis summaries, library code is replaced with assignments that produce the same memory effects as the code being removed. Note that, this enables basic def/use modeling but it does not, in general, neither handle potential aliasing effects nor it is meant to be a mechanism able to avoid instrumentation of arbitrary methods. Independently and similarly to us, authors of Mandoline tool [2] follows an approach that do not resort to dynamic tracking object references (for slicing Android apps).

Unlike our approach, Mandoline identifies variables referencing the same object via alias analysis over the trace of statements. Given a use of a field, several backward and forward traversals of portion of the trace till the set of all reaching definitions is built. Code that is out of scope of slicing must be either replaced by summaries in the form of assignments or by a built-in modeling of data propagation inside Android’s framework methods.

The static analysis community has addressed the abstract representation of heaps for different client analyses [15]. Our analysis model is "storeless" [15] in the sense that is meant to answer questions about access paths (in fact, it can ask if two scope-valid access paths may alias). However, its formulation is "store-based" since we establish an easy denotation relationship with the concrete points-to model, plus we want to operate with a graph view point when doing summarization. In fact, the context of use of our analysis model, dynamic analysis, implies our need of summarization is not iteration or recursion but the access to indexed structures and the need for accounting for effects of uninstrumented methods. Summarization of access paths is done in a graph-based fashion by adding regions and establishing \( \lambda \)-equivalence among nodes.

8 RELATED WORK

Dynamic slicing (e.g.,[1, 16]) has been regarded as a potentially useful idea but hard to scale [5, 11, 17, 22, 28, 30, 33, 36, 38]. For instance, due to the scalability issues, works like [26] end up resorting to filtering out static slices with information collected at run-time on code actually executed. The use of a static slice might render such sort of approaches too imprecise, for instance, when dealing with large classes hierarchies and rich inputs.

9 CONCLUSION AND FUTURE WORK

The contribution of this paper is a sound dynamic slicing approach, Abstract Dynamic Slicing, that just need to generate a control-flow trace of methods-to-be-sliced. It then updates and uses an abstract memory model to query about how the memory may be configured. The approach enables users to tradeoff between precision and performance by setting the code that is instrumented. We report an
implementation of the approach, DynAbs, for C# and experimenta-
tion that provides evidence that Abstract Dynamic Slicing can slice
test case assertions of large C# applications.
Future work may include defining summaries to avoid always
applying the most conservative model updates for common external
libraries and systematically dealing with multi-threading.

REFERENCES

[1] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI ’90). ACM, New York, NY, USA, 246–256.
https://doi.org/10.1145/935942.935976

[2] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Mandoline: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis. 105–115.
https://doi.org/10.1109/ICST.2021.00002

[3] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. SlicerJ: A Dynamic Slicer for Java. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’21). Association for Computing Machinery, New York, NY, USA, 1379–1388.
https://doi.org/10.1145/3451381.3451854

[4] Tanizirul Amin, Arash Alavi, Juliano Neamtiu, and Rajiv Gupta. 2019. Dynamic Slicing for Android. In Proceedings of the 41st International Conference on Software Engineering (ICSE ’19). IEEE Press, Piscataway, NJ, USA, 1154–1164.
https://doi.org/10.1109/ICSE.2019.00118

[5] David Binkley, Nicolaus Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent Program Slicing. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE ’14). ACM, New York, NY, USA, 109–120.
https://doi.org/10.1145/2635868.2635893

[6] Eric Boddien. 2018. The Secret Sauce in Efficient and Precise Static Analysis: The Beauty of Distinctive, Summary-Based Static Analyses (and How to Master Them). In Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA ’18). Association for Computing Machinery, New York, NY, USA, 85–93.
https://doi.org/10.1145/3264543.3265500

[7] Brendon Cahoon and Kathryn S. McKinley. 2001. Data Flow Analysis for Soft-
ware Prefetching Linked Data Structures in Java. In Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT ’01). IEEE Computer Society, USA, 280–291.

[8] Martin C. Carlisle and Anne Rogers. 1995. Software Caching and Computation Migration in Older. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’95). Association for Computing Machinery, New York, NY, USA, 29–38.
https://doi.org/10.1145/209596.209594

[9] dotnet. [n. d.] Roslyn, The NET Compiler Platform. https://github.com/dotnet/roslyn. Accessed: 2022-04-01.

[10] Anbang Guo, Xiaoguang Mao, Deheng Yang, and Shangwen Wang. 2018. An Empirical Study on the Effect of Dynamic Slicing on Automated Program Repair Efficiency. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME’2018). 554–558.
https://doi.org/10.1109/ICSME.2018.00066

[11] Rajiv Gupta, Mary Lou Soffa, and John Howard. 1997. Hybrid Slicing: Integrating Dynamic Information with Static Analysis. ACM Trans. Softw. Eng. Methodol. 6, 4 (Oct. 1997), 379–397.
https://doi.org/10.1145/266416.266444

[12] Clemens Hammeracher, Kevin Steett, Sebastian Hack, and Andreas Zeller. 2009. Profiling Java Programs for Parallelism. In Proc. 2nd International Workshop on Multi-Core Software Engineering (IWME9). 49–53.
https://doi.org/10.1145/1920956.1920941

[13] Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin. 2014. Alenbic: Au-
tomatic Locality Extraction via Migration. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA ’14). Association for Computing Machinery, New York, NY, USA, 879–894.
https://doi.org/10.1145/2660193.2660194

[14] Hajime Inose and Stephanie Forrest. 2002. Anomaly Intrusion Detection in Dynamic Execution Environments. In Proceedings of the 2002 Workshop on New Security Paradigms (NSPW ’02). Association for Computing Machinery, New York, NY, USA, 52–60.
https://doi.org/10.1145/844102.844112

[15] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. ACM Comput. Surv. 49, 2 (June 2016), 47 pages.
https://doi.org/10.1145/2913098

[16] B. Korel and J. Laski. 1988. Dynamic Program Slicing. Inf. Process. Lett. 29, 3 (Oct. 1988), 155–163.
https://doi.org/10.1016/0020-0190(88)90054-3

[17] Jaakko Korpela and Jussi Koskinen. 2010. Constructive Dynamic Program Slicing Research. Int. J. Adv. Comp. Techn. 2 (6) (2010), 7–23.
https://doi.org/10.1456/jact.vol2.issue2.1

[18] Xiangyu Li and Alessandro Orso. 2020. More Accurate Dynamic Slicing for Better Supporting Software Debugging. In 2020 IEEE 13th International Conference on