Measuring Enforcement Windows with Symbolic Trace Interpretation: What Well-Behaved Programs Say

Devin Coughlin* Bor-Yuh Evan Chang* Amer Diwan† Jeremy G. Siek*
*University of Colorado Boulder, USA †Google, USA
{devin.coughlin,evan.chang,jeremy.siek}@colorado.edu diwan@google.com

ABSTRACT

A static analysis design is *sufficient* if it can prove the property of interest with an acceptable number of false alarms. Ultimately, the only way to confirm that an analysis design is sufficient is to implement it and run it on real-world programs. If the evaluation shows that the design is insufficient, the designer must return to the drawing board and repeat the process—wasting expensive implementation effort over and over again. In this paper, we make the observation that there is a minimal range of code needed to prove a property of interest under an ideal static analysis; we call such a range of code a *validation scope*. Armed with this observation, we create a dynamic measurement framework that quantifies validation scopes and thus enables designers to rule out insufficient designs at lower cost. A novel attribute of our framework is the ability to model aspects of static reasoning using dynamic execution measurements. To evaluate the flexibility of our framework, we instantiate it on an example property—null dereference errors—and measure validation scopes on real-world programs. We use a broad range of metrics that capture the difficulty of analyzing programs along varying dimensions. We also examine how validation scopes evolve as developers fix null dereference errors and as code matures. We find that bug fixes shorten validation scopes, that longer validation scopes are more likely to be buggy, and that overall validation scopes are remarkably stable as programs evolve.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Languages, Verification

Keywords

symbolic trace interpretation, validation scope, enforcement windows, static analysis design, dynamic measurement

ISSTA '12, July 15-20, 2012, Minneapolis, MN, USA

1. INTRODUCTION

In the 1990s, a large program had hundreds of thousands of lines of code. By today's standards, such a program is tiny! For example, Windows Vista has a code base of 60 million lines of code created by ∼3,000 developers [1]. It is clear that no programmer can fully understand every line of code and how they relate to each other in such a large system. Rather, programmers rely on "isolation boundaries" following from modular design to reason about their code. These isolation boundaries do not always follow explicit modularization (e.g., methods, classes, and packages) but can be implicit (e.g., around groups of tightly coupled methods).

Static analysis tools, which help find bugs in software, can take advantage of isolation boundaries to scale to real-world programs. Can we find and leverage the implicit isolation boundaries created by software developers to improve static analysis? To attack this question, we define the concepts of a *validation scope* and an *enforcement window* in this paper. We then create a framework for measuring enforcement windows with dynamic analyses. At a high level, a validation scope captures a *property-based* isolation boundary implied by the code itself, and an enforcement window is a dynamic approximation of a validation scope. We define these concepts in detail in the remainder of this introduction.

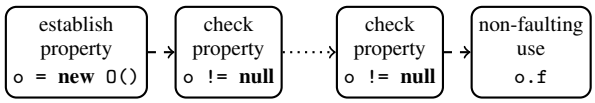
One of our key insights is that proving a property about a particular operation does not always require the entire program. In particular, we define a *validation scope* as a part of the code where if we reason operationally (e.g., directly by analyzing the code precisely), then we can prove a property of interest without any assumptions about its context. As an example, consider the inset Java fragment and what it takes to validate that the read of `o.f` cannot dereference **null**. The highlighted code fragment between the null check on line 3 and the dereference `o.f` on line 5 (shaded and marked with vertical lines) may be a sufficient validation scope to prove that `o.f` does not dereference **null** (depending on what is on line 4).

```
1  o = new O(); ...
2  if (o != null) { ... } ...
3  if (o != null) {
4    ...
5    x = o.f;
6    ...
7  }
```

Intuitively, a validation scope captures an implied isolation boundary with respect to a potential fault based on the *enforcements* inserted in the code. We use the term *enforcement* to refer generically to an operation that *establishes* or *checks* the property of interest (e.g., `o != null`). Validation scopes get at an important aspect of static analysis design and program reasoning: on one hand, a static analysis can leverage validation scopes to limit the precision use outside validation scopes, while on the other hand, a static analysis must be able to reason precisely enough inside the scope to capture the property of interest. In this paper, we propose techniques to identify potential validation scopes and ways to measure their "size"

or "complexity" before designing a static analysis.

To do so, another key insight is that analyzing well-behaved executions can provide evidence for validation scopes. In particular, given a safety property and a *non-faulting* trace (i.e., one that does not violate the property of interest), there is an event that establishes the property, potentially followed by (redundant) checks that confirm that the property continues to hold, and finally ending with a non-faulting use as diagrammed below:



For example, a non-faulting object dereference (i.e., does not dereference null) is established by the object allocation and may be validated by any number of null checks before reaching the dereference site. We call such a sequence of establish, check, and use events an *enforcement window*. An enforcement window is a dynamic approximation in that we can begin to search for candidate validation scopes by mapping enforcement events back to source code locations. Our definition of an enforcement window is property independent—-all that needs to be defined for each property is what events count as an "establish," a "check," or a "use."

In this paper, we describe a measurement framework for enforcement windows and then measure enforcement windows for an example property—specifically, non-null dereference. One measurement of interest is the distance between a use (e.g., a dereference) and its closest enforcement (e.g., a null check or an allocation) for several different notions of "distance." Intuitively, such a measure captures the "complexity" of the candidate validation scope from the closest enforcement to the use. As a simple example, consider the following three-line Java fragment where the methods corresponding to the called methods are shown inline, that is, path projected [13] (in grey backgrounds):

```
1   id = new Id(); x = new X();
2   x.setId(id);
      void setId(Id id) {
2.1    assert id != null; this.id = id;
      }
3   return x.getIdAsInt();
      int getIdAsInt() {
3.1    if (this.id == null) { this.id = new Id(); }
3.2    return this.getRawId();
          int getRawId() {
3.2.1      return this.id.raw;
          }
      }
```

Focusing on the `.raw` dereference at line 3.2.1, the enforcement window is as follows: (a) establish with the allocation of an `Id` at line 1 in the global context, (b) check at line 2.1 in `setId` (i.e., `id != null`), (c) check at line 3.1 in `getIdAsInt`, and (d) use at line 3.2.1 in `getRawId`. One interesting distance metric that we consider is the *inlining depth* needed to bring the path between the check and the use into the same method scope. In this case, there is an inlining depth of 1 between the last null check at line 3.1 in `getIdAsInt` and the dereference at line 3.2.1 in `getRawId`.

From the point of view of static analysis design, these dynamic measurements are interesting because they rule out insufficient designs. In the Java fragment and successful execution trace above, an inlining depth of 1 witnesses that a simple, conservative, *intra*procedural null deference analysis is insufficient and would necessarily result in a false alarm at the dereference site at line 3.2.1. We mean specifically that this analysis when analyzing `getRawId` has no precondition that it can assume about its context. Note that with these dynamic measurements, we get *necessary* conditions but not *sufficient* ones in that even after inlining `getRawId` into `getIdAsInt` a null dereference analysis may not be able to prove that the dereference site at line 3.2.1 is safe (perhaps because it is imprecise on aspects not captured by this particular measurement or because the dynamic analysis did not measure all program paths). We discuss why we chose dynamic over static analysis in Section 2.2 and consider this potential insufficiency further in Section 3.3.

From a software engineering perspective, our measurement framework also enables us to empirically support or refute widely-held intuitions about how programmers use enforcements in their code.

Overall, this paper makes the following contributions:

- We introduce the notion of enforcement windows that enables us to rule out insufficient static analysis designs. We systematically examine choices in deciding, where, what, and how to measure enforcement windows, and we describe distance metrics that capture reasoning about both *control* and *data* (Section 2.2).
- We present a flexible framework for measuring enforcement windows dynamically (Section 3). A challenging requirement for these measurements is a way to get at static, source code notions with dynamic analysis. We address this challenge by applying symbolic reasoning techniques and propose *symbolic trace interpretation*, whose essence is an intertwined concrete-symbolic analysis state (Section 3.1). Taking these measurements dynamically rather than statically enables us to measure one aspect of analysis precision (e.g., context sensitivity) while factoring out others, such as imprecise heap reasoning (Section 2.2).

  Measuring enforcement windows in the presence of heap objects requires careful design and special mechanisms to scale to even modestly-sized benchmarks. We describe *piggybacked garbage collection*, which collects a "shadow heap" by instrumenting the collector of the concrete heap, and we propose *measurement update partitions* that capture ways to update groups of symbolic heap values simultaneously (Section 3.2).
- We study the extent to which our dynamic measurements of enforcement windows are sufficient from a static analysis perspective by measuring whether the check sites in our observed enforcement windows are static *bottlenecks* in the control-flow graph for their use sites (Section 3.3). We find that a significant portion (30% to 80%) of use sites are statically protected by their observed closest check sites, suggesting that these measured enforcement window distances are quite likely to indicate useful validation scopes.
- We apply our trace interpretation framework to study the evolution and distribution of enforcement windows for dereferences using metrics from four broad categories (Section 4). In particular, we measure how enforcement windows for dereferences change across bug fixes for `NullPointerException`s in Java. We find that (1) enforcement windows get shorter after bug fixes and (2) that longer enforcement windows are more likely to result in bugs. These findings provide empirical evidence supporting the commonly held but difficult to verify belief that programmers find it easier to reason locally than non-locally. We also find that (3) enforcement window sizes are remarkably stable over project lifetimes, even as code bases nearly double in size, and that (4) while measured enforcement windows are in general small, in some cases they are large along certain dimensions.

2. OVERVIEW AND METRICS

In this section, we give an overview of our enforcement window measurement framework by following an example symbolic trace interpretation. Recall that our goal is to measure the "complexity" of candidate validation scopes that can potentially inform static analysis design or simply provide insights into how enforcements appear in code. We argue why symbolic reasoning on dynamic analysis is needed to get useful information by systematically laying out the various choices in deciding *where*, *what*, and *how* to measure. This discussion leads to metrics that we apply to get the measurement data presented in Section 4.

2.1 Preliminaries: Trace Instructions

Our measurement framework consists of two main components. The *trace collector* instruments Java bytecode to obtain a log of interesting events upon execution, a technique that is is fairly standard in dynamic analysis (e.g., [8]). The *trace interpreter* performs a symbolic interpretation of this log to obtain measurements of *enforcement distance*. We define an enforcement distance as some measurement between two events in an enforcement window (e.g., between establish, check, or use events). The log is essentially a sequence of instructions that records a "path slice" that we can symbolically reinterpret to obtain enforcement distances and consequently a view of how enforcements appear in the program. Crucially this symbolic interpretation enables us to extract a static, source-code view of enforcement from dynamic traces without introducing imprecision from a purely static approach (Choice 4 in Section 2.2).

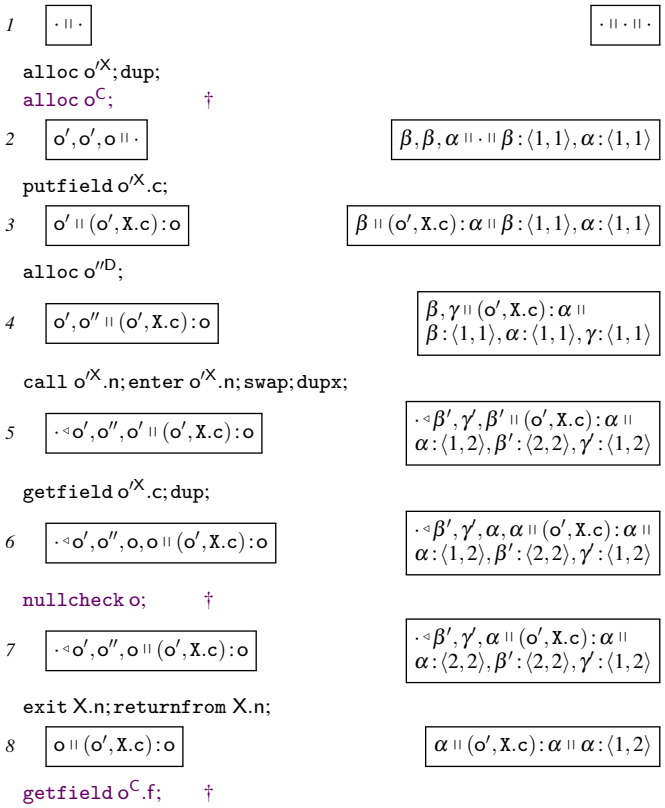
Figure 1b shows the sequence of *trace instructions* that the trace collector emits during execution of the example source in Figure 1a. Ignore the boxed items for now. The purpose of separating the collector from the interpreter is to handle most of the complexity of Java's semantics in the collector. We can write a mostly generic collector that is customized to filter (and perhaps simplify) instructions for the properties of interest. Here, we show a trace language specialized to null-dereference analysis. For exposition, we use an operand-stack–based language like Java bytecode; that is, the local store is a stack of activation records where each activation record is a stack of values. There are no integer or numeric operations here because they can be filtered out for this example analysis.

Simply to explain the semantics of this trace language, we show a concrete (re)interpretation of trace instructions that (re)creates the states of interest that would be observed in the original execution (shown in the left column of boxed items). Ignore the right column of boxed items for now, we describe them in Section 2.2. Concrete states consist of a concrete stack of activation records on the left side of the $\shortmid\!\shortmid$ and a concrete heap on the right (i.e., *stack* $\shortmid\!\shortmid$ *heap*). An activation record (i.e., a value stack) is represented by a sequence of values separated by commas, while the $\triangleleft$ symbol is used to separate activation records. Stacks grow to the right (i.e., the rightmost element is the top of the stack). For example from point 3 to 4, we have pushed o'' onto the top of the current activation after executing an allocation instruction (`alloc`), while from point 4 to point 5, we have pushed a new activation record onto the activation stack after executing method call and method enter instructions (`call` and `enter`). There are a few basic instructions that manipulate the value stack: `dup` duplicates the top value, `dupx` duplicates the top value while placing it under the top two values, and `swap` swaps the top two values. We write $\cdot$ for an empty state element (stack or heap). A heap is a map from object-field pairs to objects. For example, at point 3, we have the mapping $(o',\texttt{X.c}):o$ in the heap, which means "field `X.c` of object o' contains the value o." The other instructions note a check for null (`nullcheck`), method exit and return (`exit`, `returnfrom`), and uses of fields (`getfield`, `putfield`).

```
x = new X(); x.c = new C(); d = new D();
c = x.getCIfOk(d); return c.f;

C getCIfOk(C ow) { return this.c != null ? this.c : ow; }
```

(a) source code



(b) trace instructions

Figure 1: Concrete and symbolic trace interpretation of a short example. The left-hand–side of the figure is explained in Section 2.1, the right in Section 2.2.

Some instructions contain elements of the original execution state when the instruction was generated. For example, `alloc` o'^{X} at point 1 has as usual a type X but also an object identifier o' (e.g., address) of the allocated object. These pieces of the original concrete state serve to include concrete information for combined concrete-symbolic reasoning (e.g., somewhat similar to [9, 18]).

2.2 Measuring: Where, What, and How

Recall that an enforcement window is an establish-check-use sequence. Depending on the property of interest, particular trace instructions will correspond to establish, check, and use events. In the case of dereference reasoning, the *establish* is an `alloc`, followed by some number of `nullcheck` *check*s, and finally a `getfield`, `putfield`, or `call` *use* on the same object reference. For example, we have the establish-check-use sequence highlighted and marked by †s in Figure 1b (i.e., the sequence for the value dereferenced with `c.f` at the source-level).

The central question is given such a dynamic trace, where, what, and how can we measure to find candidate validation scopes with a static, source code notion of complexity. We devote the rest of this section to this question. Finding candidate validation scopes corresponds closely to *where* we measure (Choice 1). The measure of complexity is determined primarily by *what* we measure. We

	Static	Dynamic
Data	**Field Set** measurement: set of fields distance: |set of fields| increment: on flow to field $(o, C.f)$, add $C.f$ to set of fields.	**Flow Count** measurement: count of flows distance: count of flows increment: on flow to field $(o, C.f)$, increment count by 1.
Control	**Method Set** measurement: set of methods distance: |set of methods| increment: on exposure to $C.m()$, add $C.m()$ to set of methods.	**Inlining Depth** measurement: $(h_{\text{min}}, h_{\text{max}})$ distance: $h_{\text{max}} - h_{\text{min}}$ increment: on exposure to stack height h, update to $(\min(h_{\text{min}},h), \max(h_{\text{max}},h))$

Figure 2: Distance metrics capturing combinations of control versus data reasoning and static versus dynamic reasoning.

consider complexity in different dimensions (Choice 2) and what makes something more or less complex (Choice 3). One particularly interesting distance metric that we define is *inlining depth* alluded to in Section 1. A static, source code view is one that considers (all) other possible executions than the one observed, which typically requires some over-approximation of possible behavior. A key observation in this paper is that by controlling *how* we measure, we can model "loss of information" due to over approximation (Choices 5, 6, and 7). Such modeling necessitates intertwined concrete-symbolic reasoning and motivates *symbolic trace interpretation*.

WHERE TO MEASURE: Defining the measurement points in an enforcement window.

CHOICE 1 (MEASUREMENT POINTS): *In an enforcement window, which pairs of points are of interest?*

There are several potentially interesting points, any of which can be measured with our framework. In Section 4, we focus on uses and their closest check (or establish if there is no check). This distance captures the smallest validation scope needed to show that the use in this trace is non-faulting.

WHAT TO MEASURE: Defining the metric.

CHOICE 2 (MEASUREMENT DIMENSIONS): *What kinds of events contribute to the complexity of a validation scope?*

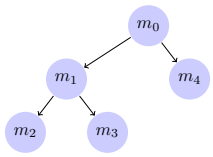
We consider two orthogonal dimensions that we hypothesize affect validation complexity: control versus data reasoning. First, *control* reasoning is what code or statements would a developer have to reason about to make sure that a dereferenced value is not null. The events that we record for control reasoning are the methods that a value is *exposed* to as it travels from an enforcement to a use (i.e., an enforcement and a use in the same method has the minimum measurement). We use the term *expose* to refer generically to observing an event that updates a measurement. We chose methods as our atomic unit of distance because they capture a source code view of the program that is always preserved by compilation to bytecode, unlike control structures or statements. Second, *data* reasoning is the memory locations that the programmer must reason about to ensure that a dereference will not fault. For this dimension, we record the fields that a value flows through between an enforcement and a use. Discovering validation scopes with respect to data reasoning might help determine where coarse heap abstractions are sufficient and where they need to be more precise.

CHOICE 3 (INCREMENTS OF MEASURE): *How do interesting events (e.g. calls or field writes) increase measured distance?*

For the data and control reasoning dimensions identified in Choice 2, what events capture an increase in complexity?

In Section 4, we take measurements using four different distance metrics: Field Set, Flow Count, Method Set, and Inlining Depth. Inlining Depth is particularly interesting from a static analysis design perspective because it captures needed context-sensitivity—a standard concept. Thus, in Figure 1b, we use Inlining Depth as the example distance metric to illustrate symbolic trace interpretation.

To describe what is Inlining Depth and why we measure it, consider the source code of our running example in Figure 1a. The last enforcement for the use of `c.f` is the null check in the call to `x.getCIfOk(d)`. Thus, a validation scope for `c.f` must also include `getCIfOk`. We want to capture the additional power needed to reason across a method call. In particular, we want to measure the *inlining depth* that is needed to bring the path that the value takes from the enforcement to the use all into the same method (i.e., into the unit scope). Note that this measurement is different and more nuanced than simply counting the number of method calls in the dynamic trace between the enforcement and the use. Consider the fragment: **assert this**`.o != null;` **this**`.m1().m2();` **this**`.o.f = 0;` where `m1` and `m2` are leaf methods (i.e., they do not make further calls), then the needed level of context is to check that `o` is not null is 1, not 2. In contrast to counting method calls in a trace, measuring Inlining Depth requires symbolic trace interpretation.



To see what needs to be measured for the inlining depth metric, consider the call tree shown inset. Each node represents a method, and each edge indicates a call from the source to the target node. The simple case is when the use is downwards along a call path (e.g., the enforcement is in m_0 and the use is in m_2), then the inlining depth is the length of the path between them (e.g., 2). The general case is that an enforcement happens in a previously called and returned method (e.g., the enforcement is in m_2 while the use is in m_4). The inlining depth is then the difference between the height of the shallowest method that the value has traveled through and that of the deepest method (e.g., m_0 and m_2, respectively, leading to a depth of 2). Thus, we measure a pair of integers $\langle h_{\text{min}}, h_{\text{max}} \rangle$ summarizing the lowest and highest activation stack height to which a value has been exposed since its last enforcement.

In Figure 1b, the right column of boxed items shows a symbolic interpretation with Inlining Depth. Consider the symbolic state at point 3. The state consists of three components, separated by ||. On the left is the symbolic stack. We use letters $\alpha, \beta, \ldots$ for symbolic values (i.e., symbolic object identities), which represent concrete values (i.e., concrete object identities). For the moment, however, we can view β and α as simply the names of $\mathtt{o}'$ and $\mathtt{o}$, respectively, in the symbolic world. In the middle, we have the symbolic heap; ignore this component for now, as we detail it under Choice 6. Finally, the rightmost component of the symbolic state associates measurements with symbolic object identities (e.g., $\beta : \langle 1,1 \rangle$); that is, it tracks an event history summary with each value individually and independently. The domain of measurements is what would vary from metric to metric.

At point 2 in Figure 1b, both objects β and α have just been established as non-null (by an allocation), so we have that both $\beta : \langle 1,1 \rangle$ and $\alpha : \langle 1,1 \rangle$ since the height of the current activation stack is 1. At points 3 and 4, these facts do not change. The `putfield` $\mathtt{o}'^{\mathsf{X}}$`.c` instruction pops arguments from the value stack and writes to the heap, and `alloc` $\mathtt{o}''^{\mathsf{D}}$ creates a new symbolic value γ with fact $\langle 1,1 \rangle$. On a call, we set h_{max} to the new height if that height is greater than h_{max}, as the value has now been exposed to a height one more call step away. So, for example, at point 5, h_{max} is incremented for α (i.e., $\alpha : \langle 1,2 \rangle$).

The measurements for β' and γ' are discussed in the next choice (Choice 5). The measurement for α is the same until encountering the `nullcheck o` instruction, which is a null-check on α, and thus resets its measurement to the current stack height (i.e., $\alpha:\langle 2,2\rangle$ at point 7). The action for a return is analogous to the call, except that $h_{\min}$ is updated, as from point 7 to point 8 for α. The inlining depth (i.e., the distance measure of interest) is then given by $h_{\max} - h_{\min}$. If, for example, α were to be dereferenced at point 5 where $\alpha:\langle 1,2\rangle$, then the check-use distance for this inlining depth metric would be 1, as expected; if it were dereferenced at point 7 after the check, the distance would be 0.

The symbolic interpretation for our other metrics is analogous, but uses a different domain of measurements. We summarize these in Figure 2 where we classify the metrics along the data versus control dimension and also along a spectrum from more static to more dynamic. For example, the Flow Count metric is a rather dynamic, execution-based view that counts the number of copies from field to field from the check until the use. This metric counts all copies, even those between the same fields of different objects (e.g., between `o1.f` and `o2.f`). The Field Set metric instead counts only the number of distinct fields through which a value flows. Method Set is the control reasoning analogue that counts the number of distinct methods to which a value is exposed.

How to Measure: Defining measurements so that they relate to source code views.

Choice 4 (Static vs. Dynamic Analysis): *Should we measure enforcements windows with a static or a dynamic analysis?*

We use *dynamic* analysis to measure enforcement windows. An initially attractive alternative would be to do so *statically* because validation scopes are inherently a static, source code notion. But upon further inspection, we see that this approach is quite problematic. For one, a "fully precise" pointer analysis, which remains a difficult problem [11], is required to statically tie together establish-check-use sequences on an object. In particular, any imprecision in the static analysis could cloud what we find—which is especially problematic when the goal was to find validation scopes to rule out insufficient static analysis designs.

Dynamic analysis is attractive because, fundamentally, it is easier to lose precision than to get it in the first place. In Choices 5 and 6, we show how we use symbolic trace interpretation to selectively forget concrete information from a dynamic trace in order to selectively emulate how a static analysis would reason about a program. However, a potential disadvantage of any dynamic analysis is that its quality depends on how well the collected traces generalize to cover all possible executions (we evaluate this in Section 3.3).

Choice 5 (Applying Measurements to Objects): *How do we connect measurements to the object that they measure?*

The two most obvious choices seem wrong. Just keeping a measurement map from concrete object identities o to measurements corresponds to the questionable assumption that perfect aliasing information is available statically. Alternatively, keeping an abstract stack and heap of measurements like in a standard type system corresponds to assuming the static analysis is incapable of resolving any aliasing.

Instead, we measure over symbolic object identities that allow us to "lose" or "forget" aliasing information known dynamically in a controlled and selective manner. Specifically, two different symbolic object identities may represent the same concrete object (modeling lost aliasing information). At point 5 in Figure 1b, we use a fresh symbolic object β' for the receiver in the callee as opposed to reusing the value β from the caller. While both β' and β correspond to same concrete object $\mathtt{o}'$, we have chosen to forget this information in the symbolic state. In this case, we make this split to capture: (1) in any method, the receiver object **this** is known to be non-null, so from the prospective of the callee, the `call` instruction is a check on the receiver; but (2) from the prospective of the caller, the `call` is simply a use/dereference of the receiver. Thus, in our example, β', the receiver in the callee after the call, is summarized by $\langle 2,2\rangle$ (i.e., checked in the callee). The parameter in the callee, γ', is also a fresh symbolic object identity where both γ and γ' correspond to the concrete object $\mathtt{o}''$. The measurement on γ' (i.e., $\langle 1,2\rangle$) is derived from γ's, but the cloning means any check in the callee on $\mathtt{o}''$ is not seen by the caller, unless it is passed back in the return value or through the heap. This approach respects the implicit modularization implied by function boundaries—checks on a value escape a function only if the value itself does.

Choice 6 (Memory Model): *How is the memory modeled symbolically?*

In symbolic interpretation, the symbolic state is a model of the concrete state. In Figure 1b, we use an exact model of the activation stack except that the values are symbolic rather than concrete: that is, the identity of a called method is exactly known, but the receiver and parameters are interpreted symbolically. Similarly, each cell in our symbolic heap is identified exactly (by a concrete object identity and a field) but the *contents* of those cells are symbolic values. So, for example, at point 3 the symbolic heap maps field `X.c` of object $\mathtt{o}'$ to symbolic value α. Heap accessing instructions operate on combined symbolic and concrete values; the symbolic values come from the symbolic value stack, while the concrete values are explicitly incorporated into select trace instructions.

The symbolic interpretation of `getfield` $\mathtt{o}'^{\mathtt{X}}$`.c` from point 5 to 6 pops the symbolic value β' for the field owner from the value stack but then uses the *concrete* annotation $\mathtt{o}'$ to look up the symbolic value stored in the symbolic heap at field `X.c` of $\mathtt{o}'$, which is α, that is then pushed onto the stack. The interpretation of `putfield` is similar. Analogous to the modeling of copying actuals to formals discussed above under Choice 5, there is a choice in whether (a) to copy the the symbolic value α from the heap to the stack or (b) to create a fresh symbolic value with a copy of the measurement. The former means a measurement update via the stack is reflected in the heap value and vice versa, while the latter "forgets" this aliasing relationship. In this case, we have chosen (a) to capture that enforcements on stack values obtained from instance variables (i.e., fields of **this**) (or vice versa) should seemingly apply in both places. Another reasonable option could choose (a) in some cases (e.g., only dereferences of fields of **this**) and (b) in other cases. There is no clear best choice regarding aliasing "remembering" and "forgetting," so importantly, our framework supports experimenting with different modeling decisions by switching between (a) and (b).

In essence, we record measurements on symbolic values but use concrete values to determine storage locations on the heap. Without the latter use of concrete values, the symbolic trace interpretation would itself need a precise static points-to analysis. Critically, this intertwining of concrete and symbolic modeling enables us to model source code reasoning in some respects while avoiding unrealistic static analysis imprecisions in other ways.

A significant implementation challenge is updating measurements for all heap-stored values (see Section 3.2). With Inlining Depth, for example, on every entry to and return from a method, we need to expose every value on the heap to the new activation stack height using the scheme laid out in Choice 3. So, for example, the measurements for α for the call between point 2 and point 5 change to reflect α's exposure to an activation stack with height 2. Similarly,

the return from point 7 to point 8 exposes α to height 1.

CHOICE 7 (MEASURING THE UNKNOWN): *How do values from uninstrumented library code contribute to check-use measurements?*

Some values will necessarily come from uninstrumentable code (e.g., libraries). We assign these values the measurement lib. In interpreting our measurements, we take the conservative viewpoint that a value returned from unknown code adds an unknown distance that must be viewed over-approximately as "infinite." Informally stated, unvalidated assumptions are made about the code outside of the validation scope, but unbounded inlining would be sufficient to validate those assumptions once the code is brought in. An alternative, optimistic approach would assign library values 0 distance indicating that libraries are understood through documentation of invariants and thus do not require reasoning about code at all.

3. MEASUREMENT FRAMEWORK

In this section, we describe our symbolic trace interpretation framework (Section 3.1), discuss techniques for scaling our implementation of the symbolic heap (Section 3.2), and investigate the extent to which our dynamic measurements of enforcement windows are sufficient from a static analysis perspective (Section 3.3).

3.1 Symbolic Trace Interpretation

The key challenges addressed by this framework are (1) how to extract a more static view of an execution by forgetting run-time information in a principled way (see Section 2.2, Choice 5) and (2) how to meaningfully interact with uninstrumented library code. We accomplish (1) by using an intertwined concrete and symbolic state, associating information with symbolic values, and instantiating new symbolic values when we want to forget. With this approach, a single concrete value can be represented by multiple symbolic values. We address (2) by splitting method call and returns into separate instructions that captures the call or return event from the caller's and the callee's perspectives individually.

We first focus on describing our generic framework instantiated for measuring control reasoning using the inline depth metric as an example. An activation record $A ::= \cdot \mid A, \alpha$ consists of an operand stack with symbolic object identities; the symbol $\cdot$ indicates an empty stack. Then, we have a stack of activations $S ::= \cdot \mid S \triangleleft A \mid S \triangleleft \mathsf{unins}$, which consist of normal activations A but also uninstrumented activations unins. Informally, unins models some number of activations for uninstrumented methods. A heap $H ::= \cdot \mid H, (o,f) : \alpha$ is a finite map from a *concrete object, field pair* to the symbolic value stored in the field for that object. Observe that the heap is a mixed concrete-symbolic entity. A measurement map $\Gamma ::= \cdot \mid \Gamma, \alpha : t$ is a finite map from symbolic identities to the recorded measurement for that symbolic value, and a symbolic state $\Sigma ::= S \shortparallel H \shortparallel \Gamma$ is a triple of a stack of activations, a heap, and a measurement map. A measurement $t ::= \langle h_{\min}, h_{\max} \rangle \mid \mathsf{lib}$ can be either a known measurement or lib, indicating that the value came from uninstrumented code. When instantiated for the inlining depth metric, known measurements consist of a pair of integers $\langle h_{\min}, h_{\max} \rangle$ representing the minimum and maximum stack height to which the value has been exposed. The measurements are the only portion of the symbolic state that change from metric to metric. We write $\Gamma(\alpha)$ for looking up the measurement associated with symbolic object α in Γ and $\Gamma, \alpha : t$ for a map that either extends Γ with a binding for α or updates the binding of α to t if it exists. Similarly, $H(o, C.f)$ looks up a value at field $C.f$ of concrete object o in H and $H, (o, C.f) : \alpha$ extends it.

We define an interpretation judgment $\Sigma \vdash I \Downarrow \Sigma'$ in Figure 3 that states, "In state Σ, instruction I symbolically evaluates to Σ'."

$$\textsc{ALLOC}\quad \frac{\alpha \notin \mathrm{dom}(\Gamma) \qquad t = \mathrm{enf}(S \triangleleft A \shortparallel H \shortparallel \Gamma)}{S \triangleleft A \shortparallel H \shortparallel \Gamma \vdash \texttt{alloc}\, o^C \Downarrow S \triangleleft A, \alpha \shortparallel H \shortparallel \Gamma, \alpha : t}$$

$$\textsc{CALL-INS}\quad \frac{\beta', \alpha' \notin \mathrm{dom}(\Gamma) \qquad S' = S \triangleleft A \triangleleft \beta', \alpha' \qquad \Sigma = S' \shortparallel H \shortparallel \Gamma \qquad \Gamma' = \Gamma, \mathrm{expose}(\Gamma|_{\mathrm{rng}(H)}, \Sigma), \beta' : \mathrm{enf}(\Sigma), \mathrm{expose}(\alpha' : \Gamma(\alpha), \Sigma)}{S \triangleleft A, \beta, \alpha \shortparallel H \shortparallel \Gamma \vdash \texttt{call}_{\texttt{ins}}\, o^C.m \Downarrow S' \shortparallel H \shortparallel \Gamma'}$$

$$\textsc{RETURNFROM-INS}\quad \frac{S' = S \triangleleft A_1, \alpha \quad \Gamma' = \Gamma, \mathrm{expose}(\Gamma|_{\mathrm{rng}(H) \cup \{\alpha\}}, S' \shortparallel H \shortparallel \Gamma)}{S \triangleleft A_1 \triangleleft A_2, \alpha \shortparallel H \shortparallel \Gamma \vdash \texttt{returnfrom}_{\texttt{ins}}\, C.m \Downarrow S' \shortparallel H \shortparallel \Gamma'}$$

$$\textsc{GETFIELD-INS}\quad \frac{(o, C.f) \in \mathrm{dom}(H) \qquad \beta = H(o, C.f)}{S \triangleleft A, \alpha \shortparallel H \shortparallel \Gamma \vdash \texttt{getfield}\, o^C.f \Downarrow S \triangleleft A, \beta \shortparallel H \shortparallel \Gamma}$$

Figure 3: Symbolic trace interpretation for inlining depth.

The trace instruction language is the same as in the example from Figure 1, except that we explicitly annotate `call` and `returnfrom` instructions with whether the called or returned-from method is instrumented (`ins`) or uninstrumented library code (`unins`). For completeness, we give the full trace language supplementally [4].

For non-null dereference analysis, an allocation is an establish event. Rule ALLOC pushes a fresh value α onto the stack with a measurement for an enforcement event (i.e., an establish or a check) in the current state. Under the inlining depth metric, this measurement has both $h_{\min}$ and $h_{\max}$ set to the current stack height. That is, we define $\mathrm{enf}(\Sigma) \stackrel{\text{def}}{=} \langle \mathrm{heightof}(S(\Sigma)), \mathrm{heightof}(S(\Sigma)) \rangle$ where the function $\mathrm{heightof}(S)$ gives the number of activations in stack S and $S(\Sigma)$ gives the stack component of the symbolic state Σ. Recall that the inlining depth for a measured exposure is given by $h_{\max} - h_{\min}$, so a use right after the allocation yields a 0 distance as intended. A `nullcheck` is essentially the same except that it updates the measurement for the object on the top of the stack (NULLCHECK rule elided here), as it is just another enforcement. For other properties, other instruction kinds may be identified as the enforcement events, but they have the same form: interpreting the semantics of the instruction along with asserting an enforcement in the measurements.

At a call to an instrumented method (rule CALL-INS), we create a fresh symbolic value to represent the receiver β' and assign it the enforcement measurement in current state. This constraint captures that the receiver is null-checked at this point from the callee's perspective (since **this** cannot be **null**) but it is not from the caller's viewpoint. Contrast this modeling with that for the parameter value α. It is assigned a new symbolic value in the callee α' so that checks in the callee do not automatically count in the caller. The measurements for that value are copied between the caller and the callee before exposing it to the new state in the callee. The $\mathrm{expose}(\alpha : t, \Sigma)$ function updates the measurement for object α to reflect exposure to a state Σ. Under the inlining depth metric, we define this as: $\mathrm{expose}(\alpha : \langle h_{\min}, h_{\max} \rangle, \Sigma) \stackrel{\text{def}}{=} \alpha : \langle \min(h_{\min}, h), \max(h_{\max}, h) \rangle$ and $\mathrm{expose}(\alpha : \mathsf{lib}, \Sigma) \stackrel{\text{def}}{=} \alpha : \mathsf{lib}$ where $h = \mathrm{heightof}(S(\Sigma))$. We lift expose to also apply to maps (i.e., $\mathrm{expose}(\Gamma, \Sigma)$). For control reasoning metrics, all measurements for values on the heap are also updated to reflect their exposure to a state on each call and return (i.e., $\mathrm{expose}(\Gamma|_{\mathrm{rng}(H)}, \Sigma)$). We write $\Gamma|_{\mathrm{rng}(H)}$ for the restriction of map Γ to mappings from symbolic values in the range of the heap H. Observe that this operation is prohibitively expensive to implement directly and motivates techniques described Section 3.2. On return (rule RETURNFROM-INS), the top activation is popped and the return value and the heap are exposed to the state.

The complexity of handling uninstrumented methods lies in transitions between instrumented and uninstrumented code. To detect transitions, we split a method call into two events: a `call` instruction, which is the event from the caller's perspective, and an `enter`, which is the event from the callee's perspective. When an instrumented method calls another instrumented method, then we see a `call` immediately followed by an `enter` as in Figure 1b. However, critically, this redundancy allows us to detect transitions between instrumented and uninstrumented code robustly. Specifically, we mark a call from instrumented code to an uninstrumented method by pushing an unins marker on to the stack. A call from uninstrumented code to an instrumented method is detected by an `enter` instruction while an unins marker is active. The interpretation of `enter` in this situation is to compensate for the lack of a `call` instruction right before it (and thus is analogous to rule CALL-INS). Method returns are similarly split into `exit` from the callee's perspective and `returnfrom` from the caller's perspective. The interpretation of `returnfrom` must make a similar compensation when it observes a return from uninstrumented code.

For control reasoning, getting and putting a field simply need to reflect the concrete semantics symbolically. Getting a field from an object pops the symbolic field owner off the stack and uses the *concrete* object identifier to look up the symbolic value stored for in that object's field in the heap (if it exists) and pushes it on the stack (GETFIELD-INS). Using concrete heap lookups enables us to factor out a potential source of unrealistic static analysis imprecisions. If the field has not been initialized, it pushes lib instead (as the assumption is that it was initialized in uninstrumented code). A `putfield` updates the symbolic heap to store a symbolic value from the stack in the field for the concrete object (rules are straightforward). For data reasoning, we would update measurements (i.e., apply exposures) on `getfield`s and `putfield`s instead of on `call`s and `returnfrom`s.

In this section, we have instantiated our measurement framework using the Inlining Depth metric. Using our Method Set metric is similar, except that the measurements are sets of method identities and exposing a value to a new state adds the method on the top of the stack to a measurement. How specifically our four metrics are instantiated in this framework is summarized in Figure 2.

3.2 The Symbolic Heap

Two key challenges hide in the description of symbolic trace interpretation above. First, in defining the symbolic trace interpretation judgment (Figure 3), heaps H and measurement maps Γ only grow. In essence, we assume that garbage is automatically collected from the symbolic heap (i.e., that objects on the heap disappear when they are no longer needed) and the measurement map. However, since the symbolic heap has no knowledge of heap operations in uninstrumented library code, there is no way the interpreter could ever safely garbage collect mappings in the symbolic heap. In our framework, we instrument the garbage collector running in the observed program to "piggyback" collecting an object in the symbolic heap when the object in the concrete heap is collected. Whenever the garbage collector frees a concrete object, the trace collector is signaled to emit a trace instruction telling the trace interpreter to remove that object from the symbolic heap. This "piggybacking" efficiently ensures that objects are only collected from the symbolic heap after they can no longer be used.

The second challenge to scalable symbolic trace interpretation involves updating the measurements for heap values on method calls and returns. In the CALL-INS and RETURNFROM-INS rules, we update every symbolic value on the heap to reflect exposure to a new control scope (i.e., $\text{expose}(\Gamma|_{\text{rng}(H)}, \Sigma)$). Naïvely iterating over the entire symbolic heap on each call and return is far too slow to be practical, even for relatively short programs.

Table 1: Framework Sufficiency for Analysis Design.

		"Full"		"Recommended"	
Program	Interesting	False Dead	Bottle-necked	False Dead	Bottle-necked
antlr	1812	0	36%	330	35%
bloat	4424	0	32%	0	32%
chart	1219	34	48%	88	50%
fop	10164	0	78%	10044	43%
luindex	873	0	40%	290	51%
lusearch	661	0	56%	0	56%
pmd	830	0	66%	63	69%

To address this problem, we divide the symbolic values on the heap into *measurement update partitions* that help us update heap exposures more efficiently. We have two partition strategies: one that leverages a property of particular kinds of measurement metrics and one that is metric agnostic but more expensive.

For the Inlining Depth metric, we partition the symbolic values based on their h_{min} and h_{max} measurements. Then, on a method call, we only need to update those values whose h_{max} is the stack height before the call. Similarly, on a return, we need only update those symbolic values whose h_{min} measurements match the stack height before the return. These partitions are *prescriptive* in that using the measurements tells us exactly which symbolic values need to be updated, and fortunately, they are small enough to speed up interpretation of calls and returns drastically.

We also have a more general, heuristic approach to partition symbolic values on the heap based on how *recently* they were used (added to the heap, read from the heap, or dereferenced). For example, in our implementation of the Method Set metric we keep a collection of up to 1000 "hot" symbolic values and update their measurements individually whenever they are exposed to a new state. The remainder of symbolic values on the heap are not updated individually on every call and return. Instead, we keep a single set summarizing the recent methods that all of these cold values have been exposed to. The key invariant that we maintain is that (1) the measurements (i.e., method exposure sets) for hot values are exactly what they would have been if we had traversed the entire heap on calls and returns and (2) the measurements for the cold values, unioned with the current summary set, are what they would have been in the naïve system. With this approach, if the program dereferences a hot value, it can record the measurement directly associated with the value. If it is cold, however, we have to first apply a lazy fixup and expose the value to each of the methods in the summarized sets. The direct measurements for that value now completely reflect what its measurements should be, so we safely move it to the hot collection. If the hot collection is full, we apply the summary set to *all* non-hot values in the heap (so that their measurements are now complete), reset the cold summary to be the empty method set, and mark all values as cold. At this point, again, the invariant holds. The essence is that we keep a fixup transformer that can be applied when a cold value gets used.

Both of these approaches make field accesses more expensive, but the savings from avoiding traversing the entire heap on calls and returns more than makes up for them.

3.3 Sufficiency for Static Analysis Design

One intended use of our enforcement window measurement framework is to help determine necessary conditions for static analysis design and, in particular, help designers decide at least how much scope their analysis needs to prove a property of interest. Our combined concrete-symbolic approach is well-suited to this task because it permits us to tease apart required scope from over-approximation in any abstract analysis domain. In essence it allows us to measure,

for example, the window of code an analysis would need to examine if it was using exactly the right analysis abstraction. Even assuming such perfect reasoning, this measurement is an under-approximation for the validation scope that the hypothetical analysis would need because, as a dynamic analysis, our approach cannot measure the required scope for all possible paths in a given program. In this section, we investigate whether this under-approximation is *sufficient* in this sense: that is, to what extent a single execution discovers enough enforcement sites to determine a useful validation scope.

To do this, we first instantiated our measurement framework for the non-null dereference property to record the *location* of the closest enforcement (i.e., allocation or comparison to null) for each dereference. A dereference may have multiple such closest enforcements if it is called from different contexts. We then used the WALA framework (`http://wala.sf.net/`) to run an interprocedural static analysis that examines each dynamically observed dereference site and verifies that all static paths in the control-flow graph to that site pass through at least one of the closest dynamically observed enforcements: if so, our approach is sufficient for that site.

The results of these experiments for a subset of the DaCapo benchmarks are shown in Table 1. Here we consider a dereference site "Interesting" if (1) it is executed in our dynamically observed run, (2) it does not dereference values from uninstrumented library code (i.e., a static analysis looking at only application code would have some hope of proving the dereference safe), and (3) it is not a dereference of `this` (which in Java cannot be null). WALA has some unsoundness in its handling of reflection, leading it to claim that some executed dereferences are not reachable. We ran WALA with two different reflection policies. "Full" makes a best effort to determine reflective method targets while "Recommended" (which was recommended to us by a WALA developer) optimistically assumes programmers' casts after reflective instantiations are correct and uses these casts to determine the type of allocated objects. Because these policies are heuristic, they may falsely claim that some dynamically observed dereference sites are dead code. We give the number of these sites in the "False Dead" column.

The "Bottlenecked" column gives the percentage of interesting, statically reachable dereference sites for which all static paths to that site pass through a dynamically observed closest enforcement; that is, the closest enforcements are a *bottleneck* to reaching the use. The observation that a large percentage (30%–70%) of dereference sites are statically shown to flow through the dynamically observed enforcement sites gives us evidence that our approach finds candidate validation scopes that are likely to be useful: that is, that inferences gleaned from these these enforcement sites in one run (e.g., their typical enforcement distances) are likely to be representative for all possible runs of the program. Note that the non-minuscule bottlenecked percentages in Table 1 are significant: even under the very pessimistic assumptions that (1) we are allowed only one dynamic execution, and (2) we count only the last enforcement along that execution, our dynamic analysis frequently finds useful validation scopes. Our benchmarks range in size from ∼3,000 (antlr) to ∼25,000 (fop) methods. The bottlenecked percentage does not change much between the "Full" and "Recommended" configurations (except for fop), perhaps indicating an invariance property about enforcements across dereference sites.

4. MEASUREMENTS

In this section, we apply our measurement framework to gain insights into how enforcements actually appear in code. We have two sets of experiments that both measure the distance between a use and its closest enforcement. The first evaluates distance metrics from Sections 2 and 3 with a case study of null pointer exception bugs that tests three hypotheses: (1) that programmers find it easier to reason across short enforcement distances than long ones, (2) that fixing bugs shortens these distances, and (3) that as code bases mature, programmers respond to increasing complexity with more defensive programming. The main challenge here was the laborious process of sifting through project issue queues and software repositories to find suitable bugs and then generating inputs that both exercise the buggy sites and remain valid across multiple versions of the projects. Our second set of experiments measures the distribution of enforcement distances over the DaCapo benchmark suite to characterize the size of potential validation scopes in typical programs.

Table 2: Distances get shorter after bug fixes.

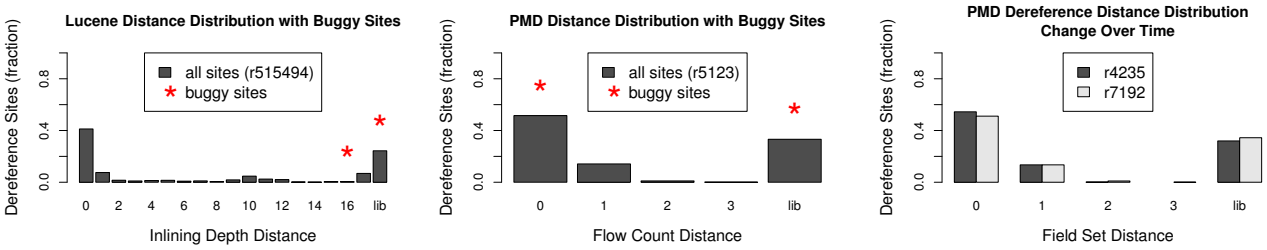
	Data Metric		Control Metric	
Issue	Flow Count	Field Set	Inlining Depth	Method Set
Lucene-825	lib → 0	lib → 0	lib → 0	lib → 1
Lucene-449	lib → 0	lib → 0	lib → 0	lib → 1
Lucene-174	lib → 0	lib → 0	lib → 0	lib → 1
Lucene-317	1 → 0	1 → 0	16 → 0	lib → lib
PMD-1425772	0 → 0	0 → 0	1 → 1	2 → 2
PMD-1529805	0 → 0	0 → 0	1 → 0	2 → 1
PMD-1552820	lib → 0	lib → 0	lib → 0	lib → 1
PMD-1728716	lib → lib	lib → lib	lib → lib	lib → lib

4.1 Case Study: Bugs and Program Evolution

This case study covers two programs in depth: PMD, a "programming mistake detector" that analyzes source code to find style violations, and Lucene, a document indexing and search tool. We perform three experiments to test a hypothesis that programmers find it easier to reason across short distances than long ones. First, we investigate how enforcement distances change after programmers fix bugs. We hypothesize that fixed bugs are likely to exhibit shorter distances since the programmer must convince herself that the bug is, in fact, fixed. Second, we look at how buggy dereference sites differ from normal sites—if longer distances are harder to reason about, we would expect to see more bugs at longer sites. Finally, we hypothesize that as programs mature and grow more complicated, programmers will need to adopt more defensive strategies and thus enforce shorter distances, so we examine how these distances change over the lifetime of projects.

Benchmark Selection. To find buggy dereference benchmarks, we were constrained by the following requirements: (1) Projects must have source repositories to get versions of the code before and after a bug fix. (2) They must have a bug database with at least 20 reported `NullPointerException` bugs so that we had a reasonable chance of triggering a buggy dereference site. (3) We limited our search for benchmarks to non-GUI programs since instrumentation slows down execution enough to make analysis of interactive programs impractical. (4) We require representative inputs over which to run our benchmarks. These constraints led us to the DaCapo suite, though we looked broadly at several open source repositories. Based on the DaCapo `small` inputs, for PMD our inputs check a file from its own source base for a variety of style violations, while for Lucene we index short portions of Shakespeare poems and then search them for the term "death."

Bug Selection. We searched each project's bug report database for instances of the word "NullPointerException." After filtering out unfixed bugs, we examined the reports to determine if they truly represented null pointer errors. For these candidates, we used the backtrace, patch date, patch author, mailing list comments, and repository logs to find the failing dereference and the source control revision numbers immediately before and after the fix. If the fix was applied across multiple commits, we used the latest revision. If the bug report spurred discovery of multiple related bugs, we



(a) All buggy sites have non-local control metric distances, but may not involve flow through the heap. Lucene exercised 2062 total sites, and PMD exercised 2522.

(b) Distances change little over time, even as the number of sites grows from 1882 to 3205.

Figure 4: Studying reported buggy dereference sites and the code evolution using enforcement distances.

only considered the original site. We removed the bugs where either revision did not exercise the buggy dereference site on the representative input. Although the site is exercised in all the remaining revisions, the bugs themselves do not manifest on the representative inputs and in some cases not all of the added code in the fixes is exercised. Overall, we obtained eight bug reports with before and after revisions on which to measure check-use distances.

Do Enforcement Distances Get Shorter After Bug Fixes? To determine whether enforcement distances get shorter after bug fixes, we annotated the buggy dereference sites for each of the bugs collected and interpreted them to collect the maximum distance at those sites before and after the fix. Table 2 shows how these distances changed after the programmer fixed the bug. Treating unknown values (e.g., from library code) as "infinite" distances, we make the following observations: (1) None of the distances get longer after a bug is fixed. The majority (five) get shorter, while three stay the same. (2) Of the distances that get shorter, most (four out of five) are "infinite" before the fix. (3) All of the distances that do get shorter go to the minimum possible distance under each metric; that is, the use and check are in the same method. For these buggy sites, the Flow Set and Flow Count distances are identical, although, as we show in Section 4.2, this is not always the case.

The nature of the bugs themselves are also quite interesting. Three of the Lucene bugs (825, 449, and 174) arise from related misuses of the `java.io.File` API to iterate through a directory (as shown inset). The developer fails to realize that `dir.list()` can return a null array if the program lacks privileges to read the directory, leading the dereference `fs.length` to raise a null pointer exception (similar to Figure 1a but with a mistaken assumption). The fixes for the three different bugs caused by this misunderstanding were also similar: the developer checks `files` for null and throws a more meaningful exception. In the fourth Lucene bug (317), a `lock` instance variable is set to null when threading is disabled, but the code calls `lock.unlock()` without checking to see if it is non-null.

```
File dir = ...;
String[] fs = dir.list();
for (i = 0; i < fs.length; i++) {}
```

For PMD, two of the bugs result from misuse of a utility function to query a node in the abstract syntax tree about its first ancestor of a given class (1425772, 1529805). In both cases, the programmer did not realize that such a parent may not exist and that the returned ancestor might be null (shown inset). Here the introduction of `enum` types in Java 5.0 broke the programmer's assumption that all nodes must have a containing class or interface. When the query returns null, the call to `p.isInterface()` throws a null pointer exception. A third PMD bug (1552820) arose from a similar query about a potentially missing child of a node. The final PMD bug (1728716) involved erroneously passing **null** to a string escape utility method. Although the bug was fixed, our analysis does not see any shortened distance because our inputs do not exercise the new check in the fix.

```
ClassOrInterface p =
  node.getFirstParentOfType(
    ClassOrInterface.class);
if (p.isInterface()) {}
```

Overall in our case study we found that enforcement distances tend to get smaller after bug fixes. This shortening helps to validate our choice of distance metrics, since we would expect a high-quality metric to show shorter enforcement windows after a bug fix. Further, the fact that most bugs involve reasoning about larger (greater than minimum) distances and most bug fixes reduce the distance to the minimum indicates that programmers are comfortable reasoning locally (within a method) but are less capable of reasoning about non-local computation. Again, this is what we would expect, but now we have empirical evidence supporting this belief, gathered by examination of software artifacts.

Do Bugs Tend to Have Long Enforcement Distances? We investigate whether buggy dereferences have longer enforcement distances by comparing the distribution of distances for all dereference sites to that for buggy sites. Figure 4a shows the distribution of all dereference sites for Lucene under the Inlining Depth control metric and for PMD under the Flow Count data metric. Buggy sites are marked with stars. For these graphs the "all sites" distribution comes from the latest before-fix revision that we analyzed—since the "all sites" distribution does not change much over time (discussed below), these graphs are representative of how buggy sites compare to all sites. We give plots of the other metrics for both benchmarks supplementally [4]—they are visibly consistent with these representative graphs.

For the control-based metrics (Inlining Depth and Method Set), the fraction of of all dereference sites that require only local reasoning (i.e., minimum distance) is significant and remarkably consistent across benchmarks—about 40% of the total 4821 dereference sites measured. Yet **none** of the buggy dereference sites involve only local reasoning. This observation further contributes empirical evidence that programmers are more comfortable reasoning locally than non-locally. The situation is not as clear-cut for data reasoning—half of the buggy sites for PMD involve a flow distance of 0 (that is, they do not involve the heap at all). These buggy sites have non-minimum measurements for both control-based metrics, possibly suggesting that these particular bugs resulted from faulty control reasoning alone. The key message from this study is that while a significant fraction of all dereference sites require only local control reasoning, buggy sites appear to be drawn from a different distribution tending towards non-local control reasoning.

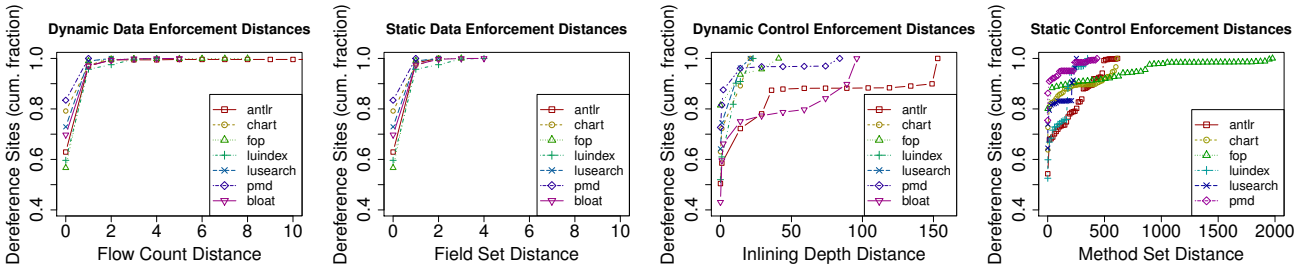


Figure 5: Distribution of dereference site measurements for DaCapo. Data enforcement distances are overwhelming short. Control distances can get very long for heap locations, suggesting non-operational heap reasoning (e.g., by encapsulation or invariants).

How Do Enforcement Distances Change Over Time? To explore how enforcement distances change over time, we compare the distribution of distances for the first revision we analyzed to a more recent one, spanning five years for PMD and 18 months for Lucene. Figure 4b shows the fraction of dereference sites with a given maximum distance for PMD under the Field Set metric at the beginning and end of the span. The distribution barely shifts to slightly longer distances. This finding is quite surprising, as the sheer increase in code size (going from 1882 to 3205 executed dereference sites and from 361 to 693 executed methods) should bias towards longer distances. The results for Lucene, and for our other metrics, are exceedingly similar. For reference, they are available supplementally [4]. For Lucene, the growth in number of measured dereferences is even larger (going from 1389 to 4176). Perhaps our metrics are capturing properties not of programs but of programmers and we should expect to see similar results for more benchmarks.

4.2 Distribution of Enforcement Distances

To understand the distribution of enforcement distances across a set of real programs, we interpreted traces over the DaCapo benchmark suite's small inputs. We omit jython, hysqldb, and xalan because of limitations in how our instrumentation handles exceptions caught in uninstrumented code and eclipse because our instrumentation causes it to deadlock. Figure 5 shows the cumulative distribution of maximum dereference distances for the Flow Count, Field Set, and Inlining Depth metrics. The y-axis shows the fraction of dereference sites with a distance less than or equal to the value on the x-axis, so for example, for luindex under the Field Set metric, 60% of sites have a maximum dereference distance of 0, while around 97% have maximum distances of 2 or less. We omit the sites with unknown distances (i.e., from library code). Interestingly for antlr on the Flow Count metric, the cumulative fraction for antlr does not reach 1.0 until a distance of 97 (so the x-axis has been cut off prematurely to elide this outlier and expose the behavior at small distances). The graphs for Flow Count (i.e., dynamic data distance) and Field Set (i.e., static data distance) are identical for distance 0, reflecting the fact that that between 55% and 85% of sites do not require reasoning about the heap. Both go to nearly 100% by a distance of 4, although the dynamic metric, as we have seen, has a very long tail. This indicates that the number of data locations about which a programmer needs to reason to ensure that a dereference will succeed is generally small but may in rare cases be large.

The small (5) number of sites in antlr that have extremely long Flow Count (97) are very interesting. The values at these sites arise from repeated execution of the pattern shown inset. These sites also exhibit the highest Inlining Depth distance observed (153) over all of our benchmarks. It appears that

```
saveField = o.field;
...// complicated recursive code
...// that may modify o.field
o.field = saveField;
```

the developer has made a deliberate decision to trade off extra data distance in order to avoid having to consider a large amount of control distance.

The situation for dynamic control (Inlining Depth metric) distances is markedly different than that for dynamic data. Although a large fraction of sites (70% to 95%) involve distances of less than 10, a significant fraction of sites show much higher distances. In the antlr benchmark, for example, around 12% of sites have distances greater than 45 and 8% have distances greater than 150. It is hard to imagine that programmers could reason operationally over such an inline depth. Recall that inlining depth speaks about an observed enforcement and its use along a call path; specifically an enforcement and a use in the same method separated by a long execution tree would still have distance 0. Instead, these large distances perhaps reflect the modeling in this metric that methods can modify *any* heap location. To examine this hypothesis further, we ran this inlining depth experiment except with heap modeling turned off (i.e., all reads from the heap are treated as unknowns and all writes to the heap are ignored), which in essence focuses the measurement to control distances of parameters. The result was that 95% of all sites had a distance of 3 or less, although both antlr and bloat had sites with maximum distances of 24 and 82 respectively. This provides some evidence for the somewhat unrealistic heap modeling hypothesis. For completeness, this plot is given supplementally [4]. In languages such as Java, type safety and encapsulation severely limit the heap locations that a given class or package can modify. An improved metric would perhaps take these features into account.

4.3 Threats to Validity

We have identified three principle threats to the validity to our conclusions: (1) *Benchmark selection:* We have chosen benchmarks that are easy to run under instrumentation and that have relatively stable interfaces (so as to allow us to use the same input over different versions of the program). This choice has led to a bias towards text processing tools. (2) *Bug selection:* We examine bugs reported in project databases, biasing our analysis towards bugs that are easier to report, which may have shorter enforcement distances. (3) *Metrics:* We have examined four of many possible different distance metrics. We discuss our reasons for choosing these metrics in Section 2, and the insights that we have obtained from the results discussed in this section has perhaps lessened this concern.

5. RELATED WORK

The closest related work is perhaps Liang et al. [16], which measures dynamically whether particular heap abstractions would have been sufficient for race and deadlock detection analyses. They focus on evaluating the abstraction function and do not perform symbolic interpretation (i.e., they instead associate facts with concrete object

identities). In contrast, our approach is abstraction-agnostic and is instead concerned with creating a framework to (a) rule out static analysis designs and (b) guess a scope (e.g., a code fragment) that may be sufficient to prove a property of interest. Livshits et al. [17] assume that bottlenecks in code enforce taint sanitization—in our work, we look to show that enforcements are bottlenecks. In contrast to work on augmenting symbolic execution with concrete information to perform directed testing or test case generation [3, 9, 18], we perform a symbolic analysis to understand source properties on a given concrete trace with an intertwined concrete-symbolic state. Dynamic invariant inference [7, 10] generalizes over observed dynamic executions to produce invariants and has been enriched with symbolic execution in DySy [5].

D'Ambros et al. [6] provide a comprehensive survey of artifact-based bug prediction metrics. Our work differs from these approaches in that we are not focused on predicting bugs per se but in understanding how enforcements are inserted to guard against faults. A large area of research studies programmers directly to see how they reason about programs (e.g., [14, 15]). With our measurements, we are not studying programmers but rather explaining empirical observations about enforcements with hypotheses about possible programmer behavior. These behaviors may be interesting to validate ethnographically.

Many have worked on null pointer error detection, both statically and dynamically. We are not specifically concerned in null pointer detection but see it as a property that naturally lends itself to the study of enforcement windows. Here, we mention a few pieces of work that make some relevant observations. Hovemeyer et al. [12] report that many null dereference bugs do not rely on heap invariants, but instead can be discovered with straightforward static data-flow analyses. Bond et al. [2] present *origin tracking*, an efficient runtime mechanism for tracing a null dereference back to the place where the null value was created.

6. CONCLUSION

We have identified two related concepts: *validation scopes* that are the code fragments needed to prove the absence of a fault and *enforcement windows* that are observed as *establish-check-use* sequences in non-faulting executions. The focus of this paper has been on creating a framework and implementation for measuring enforcement windows that enable us to inform static analysis design and to gain insights into how enforcements appear in code. A novel aspect of this framework is the application of *symbolic trace interpretation* to selectively model limitations of static reasoning in a dynamic analysis. We have given an indication that finding enforcement windows can lead to useful validation scopes. Furthermore, we have provided empirical evidence to support some widely-held beliefs about software engineering.

We chose non-null dereference enforcement windows for a case study because (a) null dereferences faults are widely-known with many techniques targeted at eliminating them, and (b) there are clear syntactic constructs that indicate *establish-check-use* sequences for dereferences. Our framework and techniques should be more broadly applicable to other enforcements, for example, downcasts, where allocation is *establish*, **instanceof** is *check* and the downcast itself is *use*. We believe our approach holds promise to help analysis designers chose effective validation scopes for a variety of interesting safety properties.

7. ACKNOWLEDGMENTS

We thank Sriram Sankaranarayanan and Manu Sridharan for insightful discussions, as well as the anonymous reviewers for their helpful comments. This research was supported in part by NSF under grants CCF-0939991 and CCF-1055066.

8. REFERENCES

[1] C. Bird, N. Nagappan, P. T. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? An empirical case study of Windows Vista. In *ICSE*, 2009.

[2] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *OOPSLA*, 2007.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[4] D. Coughlin, B.-Y. E. Chang, A. Diwan, and J. G. Siek. Measuring enforcement windows with symbolic trace interpretation: What well-behaved programs say (extended version). Technical Report CU-CS-1093-12, CU-Boulder, 2012.

[5] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

[6] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *MSR*, 2010.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2), 2001.

[8] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*, 2010.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[10] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, 2002.

[11] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, 2001.

[12] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE*, 2005.

[13] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path projection for user-centered static analysis tools. In *PASTE*, 2008.

[14] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32, 2006.

[15] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *ESEC/FSE*, 2007.

[16] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, 2010.

[17] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, 2009.

[18] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.