Lightweight Verification of Array Indexing

Martin Kellogg  
U. of Washington, USA  
kelloggm@cs.washington.edu

Vlastimil Dort  
Charles U., Czechia  
dort@d3s.mff.cuni.cz

Suzanne Millstein  
U. of Washington, USA  
smillst@cs.washington.edu

Michael D. Ernst  
U. of Washington, USA  
mernst@cs.washington.edu

ABSTRACT

In languages like C, out-of-bounds array accesses lead to security vulnerabilities and crashes. Even in managed languages like Java, which check array bounds at run time, out-of-bounds accesses cause exceptions that terminate the program.

We present a lightweight type system that certifies, at compile time, that array accesses in the program are in-bounds. The type system consists of several cooperating hierarchies of dependent types, specialized to the domain of array bounds-checking. Programmers write type annotations at procedure boundaries, allowing modular verification at a cost that scales linearly with program size.

We implemented our type system for Java in a tool called the Index Checker. We evaluated the Index Checker on over 100,000 lines of open-source code and discovered array access errors even in well-tested, industrial projects such as Google Guava.

CCS Concepts: → Software and its engineering → Software verification: Automated static analysis: Data types and structures;

Keywords: Pluggable type systems, Index Checker, Checker Framework

ACM Reference Format:

1 INTRODUCTION

An array access a[i] is in-bounds if 0 ≤ i and i < length(a).

Unsafe array accesses are a common source of bugs. Their effects include denial of service (via crashes or otherwise), exfiltration of sensitive data, and code injection. They are the single most important cause of security vulnerabilities [41]: buffer overflows enabled the Morris Worm, SQL Slammer, Code Red, and Heartbleed, among many others, allowing hackers to, for example, steal 4.5 million medical records [21]. If all array accesses were guaranteed to be in-bounds, these attacks would be impossible. A run-time system can prevent out-of-bounds array accesses, but at the cost of halting the program, which is undesirable. Despite decades of research, preventing out-of-bounds accesses remains an urgent, difficult, open problem.

An ideal technique for avoiding out-of-bounds array accesses should satisfy the following criteria:

• prevent the vulnerability rather than merely detecting it at run-time and crashing the program.
• be efficient, with no time or space overhead at run time.
• be compatible with legacy code, which need not be rewritten.
• be sound: prevent all bounds errors, rather than finding some and letting hackers and users encounter others.
• be precise: issue few false positive warnings.
• be fast: work modularly and incrementally when the programmer makes a change.
• be deterministic with regard to output and run time; small program changes do not have large or non-local effects.
• be comprehensible: the user can understand why the analysis fails or succeeds, and the output localizes the actual error.
• be effective: finds new bugs in real-world codebases.
• be usable, without disproportionate effort or code clutter.

Many academic and industrial approaches have been put forward to address this important problem. These advances have made both theoretical and practical contributions to science and engineering. Dynamic bounds checking augments the program with run-time checks, crashing the program (i.e., by throwing an exception) instead of performing illegal operations. This widely adopted approach fails the first three criteria. Heuristic-based, compile-time bug-finding tools are useful for finding some defects, but provide no guarantee, failing the soundness criterion. Several types of sound static analyses could prevent bounds errors at compile time, though most sound tools have not been evaluated in substantive case studies. Proof assistants fail the compatibility, speed, and comprehensibility criteria: they require heroic effort to use and understand, and they require re-implementation of the program or the programming language. Automated theorem provers translate the verification problem into a satisfiability problem, then invoke a solver; they fail at comprehensibility and either speed or determinism. Bounded verification (model checking or exhaustive testing) is generally not sound. Inference approaches do not require programmer annotations but fail the speed, determinism, and comprehensibility criteria. Hybrid static–dynamic approaches trade off the criteria, but satisfy no more of them than their component approaches. Section 7 discusses related work in more detail and gives citations.

We propose to prove safety of bounds checks—equivalently, to detect all possible erroneous array accesses—via a collection of type systems. Typechecking is a nonstandard choice for this problem. In previous attempts, types were too weak to capture the rich arithmetic properties required to prove facts about array indexing, could be hard to understand, and cluttered the code. One of our contributions is to show that a carefully-designed collection of type systems—each specialized to a simple property—is an excellent fit to the problem.

A type system can satisfy all the criteria listed above. Typechecking happens at compile time. It can run within an existing compiler.
and is familiar to programmers. Our type system is sound (see section 6.1 for the usual caveats about our implementation). Every sound analysis suffers false positives, but ours issues fewer false positives than Java’s own type system does (table 3). Typechecking is modular and deterministic. Types systems specialized to a simple property do not require programmers to reason about the full complexity of dependent types. Our experiments show types are effective for array bounds checking. We hope our success inspires researchers to consider types as a practical approach in other challenging verification domains.

We have developed a set of lightweight, easy-to-understand type systems, which we implemented in a tool called the Index Checker. The Index Checker provides the strong guarantee that a program is free of out-of-bounds array accesses, without the large human effort typically required for such guarantees. The Index Checker scales to and finds serious bugs in well-tested, industrial-size codebases. The Index Checker’s type systems are simple enough for developers to reason about yet rich enough to guarantee that real programs are free of indexing errors (or to reveal subtle errors). The Index Checker verifies that programmer-written type annotations are consistent with the code; that is, at run time, the values have the given type. This provides a documentation benefit: programmers cannot forget to write documentation of necessary indexing-related properties, the documentation is guaranteed to be correct, and the types are both more formal and more concise than informal English documentation.

We implemented our type systems for Java. Our work generalizes to other languages because there is nothing about our type systems that is specific to Java. Our implementation handles arbitrary fixed-length data structures, such as arrays, strings, and user-defined classes. In fact, it found errors in collection classes defined in Google’s Guava library.

We evaluated our type system with three case studies on open-source code in everyday industrial use. The case studies show that the Index Checker scales to practical programs at reasonable programmer effort. The Index Checker found bugs in well-tested, widely-used code that were acknowledged and subsequently fixed by maintainers. Most importantly, it certified that no more array bound errors exist in checked code (modulo soundness guarantees).

The primary contributions of this paper are:

- Reducing array bounds checking to 7 kinds of reasoning.
- Modeling these kinds of reasoning as simple type systems.
- Our open-source implementation, the Index Checker, runs on Java, scales to large programs, and is distributed with the Checker Framework (https://checkerframework.org/).
- Case studies showing that the Index Checker finds bugs in real programs.
- A comparison of our type-based approach to other approaches.

2 VERIFICATION VIA COOPERATING TYPE SYSTEMS

An array access $a[i]$ is in-bounds if two properties hold: $0 \leq i$ and $i < a.length$.\footnote{Evaluation of $a[i]$ could suffer other problems: a value could be undefined (e.g., uninitializated, deallocated memory, or null); the stack could overflow if array dereference is implemented as a procedure call; etc. Our work specifically addresses array indices being within their bounds.} Sections 2.2 and 2.3 show how to establish them, thus proving an access safe. However, an analysis that computes

\begin{itemize}
  \item Constants (§2.4)
  \begin{align*}
    i = 3, \quad & a.length = 4 \\
    \end{align*}
  \item Linear inequalities (§2.7)
  \begin{align*}
    & i < j \\
  \end{align*}
  \item Minimum lengths (§2.5)
  \begin{align*}
    & a.length > 10 \\
    & \text{Lower bounds (§2.2)}
    \begin{align*}
      & i \geq 0 \\
    \end{align*}
    \item Equal lengths (§2.6)
    \begin{align*}
      & a.length = b.length \\
    \end{align*}
    \item Upper bounds (§2.3)
    \begin{align*}
      & i < a.length \\
    \end{align*}
\end{itemize}

Figure 1: Information flows between type systems. The type systems with two boxes ensure each array access is safe; the other type systems support the work of these two. A dashed line indicates flow of information from user-written annotations (see section 2.9).

only those two properties would flood the user with false positives. One of our contributions is identifying 7 kinds of knowledge (fig. 1) that are adequately precise in practice, and designing abstractions (type systems) for each. Each subsequent section gives an example of safe code that cannot be typechecked under the analyses shown so far, and shows how we enhanced our design to accommodate that code. These enhancements improve precision without affecting soundness.

2.1 Background

A type is a set of run-time values: an expression’s compile-time type is an overestimate of all its possible run-time values. Typechecking is a dataflow analysis that produces sound estimates of what a program may compute. Some of our types apply to integers, like Java’s `int`, `Integer`, etc. (sections 2.2–2.4, 2.7, and 2.8). Other types apply to fixed-length collections (sections 2.5 and 2.6). Every variable has one or more types from each type system at every point in a program.

Our implementation, the Index Checker, runs 7 cooperating type systems—those that run later can use information already computed by earlier type systems, as shown in fig. 1. Section 2.9 describes how to use rely-guarantee reasoning to permit mutual dependence between two type systems (the cycle in fig. 1).

The Index Checker uses some dependent types [43]. A dependent type is a type whose definition depends on a value, such as “an integer less than the length of array a” or “an array of length 10”. A dependent type may mention a value or the name of a variable.

A type qualifier [23] refines a type by restricting the set of values it represents, meaning a qualified type is a subtype of the same unqualified type. Essentially, it is a separate type system that can be mixed into a base type system. The Index Checker uses Java’s type annotations to represent type qualifiers in Java source code. For instance, in the variable declaration `@Positive int i`, the type is `@Positive int`, which contains fewer values than `int` and is therefore
The Index Checker warns that the first type system estimates a lower bound for each integer. This is a benefit of our simple type systems), and because of space limitations.

The type system for the lower bounds of integers is shown in Figure 2. It has three types:

1. A type for positive integers, which is useful for one-based loops.
2. A type for integers greater than or equal to -1, which is useful for one-based offsets.
3. A type for non-negative integers, which is useful for one-based indexing.

The simplest possible type system that would permit verification of (some) lower bounds would only include two types: non-negative and top. Our type system for lower bounds adds two additional types:

1. A type for positive integers, which is useful for one-based offsets and for array accesses of the form a[i-1].
2. A type for integers greater than or equal to -1, which is useful for loops that decrement the loop control variable by 1 and for index0f methods which return -1 on failure.

For example, consider the following code from one of our case studies, which uses a one-indexed variable without documenting it:

```java
/* Prints the matching item.
 * @param itemNum specifies which item to print from
 * @param items the items to print from when there are multiple matches */
void printItem(Object[] items, int itemNum) { // printItem(items[itemNum - 1]);
}
```

The Index Checker claims this may be too low. The programmer should document itemNum as a 1-based index by declaring it as @Positive int itemNum. Then, the shown code type-checks, and the Index Checker also verifies that all clients of printItem respect its contract (i.e., all clients only pass @Positive integers for itemNum).

Our type system does not support other constant lower bounds; for example, it cannot express that i ≥ 2. This design decision is intentional. Arbitrarily complex type systems require arbitrarily complex reasoning. Instead, the Index Checker uses a type system that is sufficiently expressive to verify array bounds in practice.

### 2.3 A Type System for Upper Bounds

The index in an array access must be less than the length of the array. The Index Checker infers offsets within method bodies, where they are most common. Each offset is an arbitrary expression. The Index Checker does not support other constant upper bounds; for example, it cannot express that i < 0. This design decision is intentional. Arbitrarily complex type systems require arbitrarily complex reasoning. Instead, the Index Checker uses a type system that is sufficiently expressive to verify array bounds in practice.

The type system for the upper bound of an integer $i$ is shown in Figure 3. It has two types:

1. A type for integers less than $\ell$.
2. A type for integers less than or equal to $\ell$.

These types are most common. Each offset is an arbitrary expression. The Index Checker does best-effort sound reasoning. As is typical for static analysis tools, it uses top as an estimate for non-linear arithmetic and other constructs that are challenging for static analysis.

To see why offsets are necessary, consider the following code from one of our case studies:

```java
public void concat(int[] a, List<Object> b) {
  int b.size = b.size();
  Object[] res = new Object[a.length + b.size];
  for (int i = 0; i < b.size; i++) { // res[i + a.length] = b.get(i);
    res[i + a.length] = b.get(i);
  }
  ...
}
```

The relevant type is ØLength0f(“res.length”, “a.length”). An inference rule automatically infers this type, without the need for programmer annotations, because res.length = b.size + a.length which is non-negative and i is less than b.size. The relevant type system is shown in Figure 3.

---

Lightweight Verification of Array Indexing

**ISSTA’18, July 16–21, 2018, Amsterdam, Netherlands**
The Index Checker verifies that both arrays have zero or more elements.

```
int min = array[0];
```
The Index Checker is implemented in 5,539 lines of code. The Index Checker is open-source and distributed with the Checker Framework [42] (https://checkerframework.org/), an industrial-strength, open-source tool for building Java type systems that is used at companies such as Amazon, Google, and Uber. The framework abstracts some details of the analysis implementation (e.g., by modeling the heap), and automatically supports features such as flow-sensitive local type inference, Java generics, and qualifier polymorphism.

The Index Checker is implemented in 5,539 lines of code. The Index Checker’s implementation consists of one typechecker for each type system in section 2. This structure keeps each typechecker relatively small and easy to understand. Each typechecker contains a definition of the type hierarchy, type rules, and inference rules.

The type and inference rules are implemented directly, without calling an external solver. This keeps performance fast and predictable, and can be more expressive, at the cost of an increase in implementation size.

One possible solution would be to run typecheckers multiple times until a fixed point, each time utilizing only knowledge that had already been established. However, we want to retain the speed of running each typechecker only once for each line of code.

Instead, the Index Checker uses rely-guarantee reasoning [31] to implement the mutual dependency. The analysis that computes minimum array lengths runs first, and relies on (that is, assumes the truth of) any annotation explicitly written by the programmer. The Index Checker guarantees that the explicit positive annotation will be checked later by the type system in section 2.2.

This required extending the Checker Framework, which made no distinction between user-written and inferred annotations before.

### 3 IMPLEMENTATION

A programmer invokes the Index Checker by running javac with the -processor command-line option. The Index Checker produces type errors exactly as javac does.

In addition to the type qualifiers in section 2, every type system contains a bottom type \( \perp \), which is the type of \( \texttt{null} \). \( \perp \) is needed because the Index Checker handles all of Java’s numeric types (i.e., both \( \texttt{int} \) and \( \texttt{Integer} \)). Each type system also contains a qualifier that indicates qualifier polymorphism, enabling parametric polymorphism independently over Java types, qualifiers, and full types. The Index Checker has annotations that are syntactic sugar for combinations of annotations from two type systems (table 2) to help programmers express common invariants.

The Index Checker provides 1,238 annotations for the JDK (Java’s standard library), which we wrote based on the JDK’s documentation. These annotations are trusted but not checked; a future case study could verify the JDK implementation. The Index Checker’s users can write similar annotations for other libraries.

The Index Checker is open-source and distributed with the Checker Framework [42] (https://checkerframework.org/), an industrial-strength, open-source tool for building Java type systems that is used at companies such as Amazon, Google, and Uber. The framework abstracts some details of the analysis implementation (e.g., by modeling the heap), and automatically supports features such as flow-sensitive local type inference, Java generics, and qualifier polymorphism.

The Index Checker is implemented in 5,539 lines of code. The Index Checker’s implementation consists of one typechecker for each type system in section 2. This structure keeps each typechecker relatively small and easy to understand. Each typechecker contains a definition of the type hierarchy, type rules, and inference rules.

The type and inference rules are implemented directly, without calling an external solver. This keeps performance fast and predictable, and can be more expressive, at the cost of an increase in implementation size.

### Table 2: Annotations supported by the Index Checker as syntactic sugar for multiple annotations from the type systems of section 2.

<table>
<thead>
<tr>
<th>Type Qualifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>@IndexFor(&quot;a&quot;) int i</td>
<td>( 0 \leq i &lt; a.length )</td>
</tr>
<tr>
<td>@IndexOrHigh(&quot;a&quot;) int i</td>
<td>( 0 \leq i \leq a.length )</td>
</tr>
<tr>
<td>@IndexOrLow(&quot;a&quot;) int i</td>
<td>(-1 \leq i &lt; a.length )</td>
</tr>
<tr>
<td>@LengthOF(&quot;a&quot;) int i</td>
<td>( i = a.length )</td>
</tr>
<tr>
<td>@LTEqLengthOf(&quot;a&quot;) int i</td>
<td>( i \leq a.length )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lines of code</th>
<th>Guava*</th>
<th>JFreeChart</th>
<th>Plume-lib</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotatable locations</td>
<td>10,694</td>
<td>94,233</td>
<td>14,586</td>
<td>119,503</td>
</tr>
<tr>
<td>Annotations</td>
<td>10,571</td>
<td>65,051</td>
<td>12,074</td>
<td>87,696</td>
</tr>
<tr>
<td>Annotatable locations</td>
<td>547</td>
<td>2,936</td>
<td>242</td>
<td>3,725</td>
</tr>
<tr>
<td>Bugs detected</td>
<td>5</td>
<td>64</td>
<td>20</td>
<td>89</td>
</tr>
<tr>
<td>Bugs fixed</td>
<td>5</td>
<td>11</td>
<td>20</td>
<td>36</td>
</tr>
<tr>
<td>False positives</td>
<td>114</td>
<td>350</td>
<td>43</td>
<td>507</td>
</tr>
<tr>
<td>False positives (%)</td>
<td>3.7%</td>
<td>2.8%</td>
<td>2.4%</td>
<td>2.9%</td>
</tr>
<tr>
<td>Java casts</td>
<td>219</td>
<td>2,707</td>
<td>223</td>
<td>3,151</td>
</tr>
</tbody>
</table>

* Guava packages base and primitives.

### 4 CASE STUDIES

We ran the Index Checker on three open-source Java projects (table 3) used previously to evaluate bug-finding and verification tools [7, 12, 25]. Our goal was either to verify that each was free of array indexing bugs, or to find and fix all their array indexing bugs.

We first read the developer-provided documentation and rewrote it formally, as Index Checker annotations. Then, we ran the Index Checker, investigated each warning it issued, and took one of these actions: (1) Added missing annotations to an incomplete specification and re-ran the Index Checker. (2) Fixed a bug in the code and reported it to the maintainers. (3) Determined that the code was correct, but the Index Checker was not sufficiently powerful to prove it correct. We suppressed these false positive warnings.

Each case study was performed by someone who was not familiar with the codebase being checked but familiar with the Index Checker. We spent most of our time understanding subtle and/or undocumented code and improving documentation or fixing errors. Reading the entire codebase was not necessary. Overall, the case studies demonstrate three results:

(1) The Index Checker scales to sizable programs.
(2) The Index Checker finds real bugs even in well-tested programs.
(3) The Index Checker issues fewer false positives than Java’s type system. Java programmers write casts to suppress false warnings from Java’s type system. The Index Checker handles casts soundly, issuing a warning at each annotated type downcast. Sections 4.4 and 4.5 discuss false positives and annotation burden.

#### 4.1 Guava Case Study

Guava [26] is Google’s general-purpose core libraries for Java. Guava is considered extremely reliable: it is used extensively in production at Google and elsewhere, and its test suite is larger than its code. We annotated two packages, \texttt{com.google.common.base}
and com.google.common.primitives. Most of the uses of fixed-length sequences in base are strings, either directly or through the CharSequence interface. primitives uses arrays and defines custom fixed- and mutable-length collections. Much of the code is duplicated for each of Java’s eight primitive types.

We found 5 bugs in Guava, all of which were instances of the same programming mistake. The bug involves factory methods for immutable collections, which begin with code such as:

```java
public static ImmutableIntArray of(int first, int... rest) {
    int[] array = new int[rest.length+1];
    ... some code ...
    return range; // range is a new ImmutableIntArray
}
```

This code uses unchecked integer addition to compute the length of a new array. If the rest array has a length equal to Integer.MAX_VALUE, the addition overflows, and the method attempts to allocate an array of negative size. Guava’s maintainers classified this bug as priority one and accepted our patch that documents the maximum allowed array length and checks this requirement at run time.

The Guava package primitives consists almost entirely of classes working with or representing sequences of Java primitive types. These classes require many annotations in their public interfaces, accounting for the relatively large number of annotations Guava required. For example, of the 160 total @IndexOrHigh annotations we wrote in the Guava case study, 114 were in this package on parameters of methods that take a pair of indices specifying a range in a sequence, such as:

```java
List<Long> subList(@IndexOrHigh("this") int fromIndex,
                   @IndexOrHigh("this") int toIndex) { ... }
```

Table 4: Annotation density: number of annotations per line of code. Annotations with larger denominators are rarer. @SearchIndexFor only appears in the JDK. The annotations appear in the same order as in section 2.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Guava</th>
<th>JFreeChart</th>
<th>Plume-lib</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>@NonNegative</td>
<td>1/139</td>
<td>1/50</td>
<td>1/228</td>
<td>1/59</td>
</tr>
<tr>
<td>@GETNegativeOrOne</td>
<td>1/972</td>
<td>1/1193</td>
<td>1/2917</td>
<td>1/1258</td>
</tr>
<tr>
<td>@Positive</td>
<td>1/446</td>
<td>1/2005</td>
<td>1/1216</td>
<td>1/1440</td>
</tr>
<tr>
<td>@LengthOf</td>
<td>1/891</td>
<td>1/7249</td>
<td>1/912</td>
<td>1/2915</td>
</tr>
<tr>
<td>@HasSubsequence</td>
<td>1/972</td>
<td>0</td>
<td>0</td>
<td>1/10865</td>
</tr>
<tr>
<td>@InRange</td>
<td>1/563</td>
<td>1/688</td>
<td>0</td>
<td>1/766</td>
</tr>
<tr>
<td>@Total</td>
<td>1/10694</td>
<td>1/13462</td>
<td>0</td>
<td>1/14939</td>
</tr>
<tr>
<td>@ArrayLen</td>
<td>1/5347</td>
<td>1/819</td>
<td>1/503</td>
<td>1/819</td>
</tr>
<tr>
<td>@MinLen</td>
<td>1/191</td>
<td>1/1812</td>
<td>1/972</td>
<td>1/972</td>
</tr>
<tr>
<td>@SameLen</td>
<td>1/1528</td>
<td>1/1428</td>
<td>1/858</td>
<td>1/1328</td>
</tr>
<tr>
<td>@LessThan</td>
<td>1/289</td>
<td>1/18847</td>
<td>1/2084</td>
<td>1/2439</td>
</tr>
<tr>
<td>@SearchIndexFor</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>@IndexFor</td>
<td>1/281</td>
<td>1/202</td>
<td>1/521</td>
<td>1/224</td>
</tr>
<tr>
<td>@IndexOrLow</td>
<td>1/167</td>
<td>1/47117</td>
<td>1/3647</td>
<td>1/1707</td>
</tr>
<tr>
<td>@IndexOrHigh</td>
<td>1/67</td>
<td>1/4712</td>
<td>1/810</td>
<td>1/604</td>
</tr>
<tr>
<td>@LTEqLengthOf</td>
<td>1/891</td>
<td>1/9423</td>
<td>1/1458</td>
<td>1/5196</td>
</tr>
<tr>
<td>@LengthOf0</td>
<td>0</td>
<td>1/2692</td>
<td>0</td>
<td>1/3415</td>
</tr>
<tr>
<td>All annotations</td>
<td>1/20</td>
<td>1/32</td>
<td>1/60</td>
<td>1/32</td>
</tr>
</tbody>
</table>

4.2 JFreeChart Case Study

JFreeChart [24] is used by Java application developers to include graphs and charts in their programs. It uses arrays and fixed-size structures extensively to represent data it draws onto charts.

The Index Checker found 64 bugs in JFreeChart, all of which would lead to crashes. Of these bugs, 24 were in code and 40 were in documentation. Of the code bugs, 14 were failures to check arguments to public methods before indexing; 3 were inconsistencies between JFreeChart’s time classes and the JDK’s calendar class; and 2 resembled the bug in fig. 5. The other 7 code bugs all had different causes. The other 40 bugs involved undocumented assumptions made by code. Clients could pass values permitted by the documentation and cause a crash. We fixed these bugs by modifying the documentation to reflect the actual assumptions.

The maintainers accepted our first two patches (fixing 11 bugs), then went dormant. As of this writing, there has only been one commit, and no fixed bugs or accepted pull requests, in over three months. Our third patch is pending, and we will file the remainder as our patches are accepted.

4.3 Plume-lib Case Study

Plume-lib [16] is a library of Java utility methods. Like Guava, it is well-tested: its JUnit tests contain 2,693 lines of code and 1,136 assert statements.

We found 20 bugs, all of which were fixed by the developers.

We fixed 9 code defects. One involved a table header that was printed even when the table itself was empty. Three were crashes due to unchecked, externally supplied indices; we added code that checks the user’s input and prints a user-friendly error message rather than a stack trace. One was an access to an array that might have zero length. One was a crash that would occur only when duplicate arrays were passed to a method. One was a case where an array’s length was checked, but then it was dereferenced unsafely anyway. Two were in routines that should have accepted ragged arrays, but used the length of the first subarray for all subarrays.

In 11 cases, methods were missing documentation about their requirements and assumptions; without the documentation we added, a user might have supplied illegal input and caused a crash.

4.4 Causes of False Positives

Every sound type system rejects some safe programs that never perform an undesired operation at run time. If the programmer is
Lightweight Verification of Array Indexing

confident the code is safe (due to manual, or other, verification), the programmer can suppress the warning. When using the Index Checker, this is expressed using Java’s @SuppressWarnings syntax.

The Index Checker issues 507 false positive warnings in our case study. This number may seem high, but it is less than the Java type system. Programmers wrote 3,151 casts to suppress false positives from the Java type system. (One could measure the percentage of reports that are false positives, but this would be primarily a measure of code quality rather than tool quality. For any program, once its bugs are fixed, the percentage of reports that are false positives is by definition 100%. All our subject programs had high quality to begin with. The most effective way to use a typechecker is throughout development, not after testing and deployment.)

This section now discusses the three most common reasons for the Index Checker to issue a false positive in our case studies.

1. The Index Checker is restricted to immutable length data structures (see section 6.2). When code relies on interoperation between mutable-length data structures and arrays, the Index Checker may issue false positives. For example, consider the following method from PlumeLib:

```java
public <T> T[] concat(List<T> a, T[] b) {
    T[] result = (T[]) new Object[a.size() + b.length];
    for (int i = 0; i < a.size(); i++)
        result[i] = a.get(i); // false positive
    ...
}
```

result’s length is greater than a’s size, so i must be an index for result, but the Index Checker conservatively assumes that a’s size might have changed before i is used to access result. Interfaces for custom collections with implementations that are backed by either an array or a list are also common. All interactions between arrays and lists required 56 warning suppressions.

2. The JFreeChart library (see section 6.2). When code relies on interoperation between mutable-length data structures and arrays, the Index Checker may issue false positives. For example, consider the following method from PlumeLib:

```java
public class DefaultPolarItemRenderer {
    /** The plot the renderer is assigned to. */
    private PolarPlot plot;
    public LegendItem getLegendItem(...) {
        XYZDataset dataset = plot.getDataset(plot.getIndexOf(this)); // f. pos
        ...
    }
}
```

Because this is a field in plot, getIndexOf cannot return -1 here even though its documentation indicates that it could. We suppressed 46 warnings caused by this pattern. Reasoning about an invariant like this is beyond the capabilities of the Index Checker.

3. The JFreeChart library defines custom data structures that are backed by ragged arrays, called datasets. Every method to access a dataset requires two parameters: an index into the array of “series” (that is, other arrays), and an index into the corresponding series. For example, the definition of get2 in the XYZDataset is:

```java
public Number get2(int series, int item);
```

The Index Checker handles most uses of datasets correctly, but a limitation of the Checker Framework causes 40 false positives when a programmer casts a generic dataset to a more specific subclass of dataset, like XYZDataset. The Checker Framework conservatively assumes that this cast invalidates indexes dependent on behaviors because the new type may not support those behaviors.

4.5 Annotation Burden and Benefit

Table 4 shows how often each Index Checker annotation needed to be written. Verification is a difficult problem and provides strong guarantees, so some programmer effort is expected. The annotation burden is less than for Java generics, for 2 of the 3 subject programs; that is, the programs contain fewer Index Checker annotations than Java generic type arguments. The annotations are also much smaller than the programs’ test suites; this comparison is relevant because testing is another way to find errors, though in each of these programs the testing missed errors.

The annotations are more concise and precise than English, so writing them reduces the size of documentation. Since they are typechecked, they are more reliable than English documentation, which may be out of date. So the annotation count is less a measure of programmer burden than a measure of documentation benefit.

5 COMPARISON TO OTHER APPROACHES

Array indexing bugs are an important problem in practice, so many tools have been built to detect them. We compared the Index Checker to three tools, representing three different approaches to the problem.

FindBugs is bug-finding tool widely deployed in industry that uses heuristic-based pattern-matching and static analysis [2]. It emphasizes its low false-positive rate and ease of use; unlike the other tools, it does not aim to be sound. We used FindBugs v. 3.0.1 with all 9 bug patterns related to array or string indexing.

KeY [1] verifies Java Modeling Language (JML) [8, 33] specifications, which can express full functional correctness properties, using an automated theorem prover. We used KeY v. 2.6.3 with Z3 [13] v. 4.5.1. We translated Index Checker annotations to JML, and otherwise instructed KeY to verify the weakest possible contracts.

Clousot checks Code Contracts5 on .NET methods [20]. It works by abstract interpretation. We used the Code Contracts v. 1.9.10714.2 extension to Microsoft Visual Studio Enterprise 2015 v. 14.0.25431.01 Update 3 with Microsoft .NET Framework v. 4.7.02556, with arithmetic and bounds checks enabled and the SubPolyhedra [32] abstract domain. We disabled contract inference to prevent Clousot from detecting that parts of some test cases are unreachable. We hand-translated code from Java to C# and Index Checker annotations to Code Contracts. We used equivalent classes and methods from the .NET Framework in place of JDK classes if there was a clear correspondence; otherwise, we wrote stub classes to mimic them. For example, we translated new String(bytes, 0, pos) to Encoding.ASCII.GetString(bytes, 0, pos).

We first tried to run each tool on the case study programs from section 4 (section 5.1). We also ran each tool on an example of each bug found by the Index Checker that was accepted by developers (section 5.2). Finally, we compared Clousot to the Index Checker using the test suites of the two tools (sections 5.3 and 5.4).

We categorized all the bugs that the Index Checker detected and whole-program analysis. The KeY distribution includes a tool to generate stubs for unavailable code, but the generated stubs omitted some JDK dependencies for each of our benchmarks. KeY also does not support features of Java used in real-world code: generics, floating-point numbers, several Java 7 features, and all Java 8 features.

We did not run Clousot on the case study programs because hand-translating each case study from Java to C# would have been prohibitively time-consuming.

5.1 Case Study Programs

FindBugs neither found any bugs nor issued any false positives on the case study programs from section 4. KeY failed to run on all of our case study programs. KeY is a whole-program analysis. The KeY distribution includes a tool to generate stubs for unavailable code, but the generated stubs omitted some JDK dependencies for each of our benchmarks. KeY also does not support features of Java used in real-world code: generics, floating-point numbers, several Java 7 features, and all Java 8 features.

We did not run Clousot on the case study programs because hand-translating each case study from Java to C# would have been prohibitively time-consuming.

5.2 Developer-accepted Bugs

We categorized all the bugs that the Index Checker detected and the program’s developers fixed, into 18 categories. We created a minimized example of each, a few lines of code long. We also wrote corrected versions of these minimized examples. We then ran all four tools on these 36 code snippets (table 5).

Table 5: Effectiveness of 4 tools in finding real indexing bugs in their default configurations.

<table>
<thead>
<tr>
<th></th>
<th>FindBugs</th>
<th>KeY</th>
<th>Clousot</th>
<th>Index Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positives</td>
<td>0 / 18</td>
<td>9 / 18</td>
<td>16 / 18</td>
<td>18 / 18</td>
</tr>
<tr>
<td>False Negatives</td>
<td>18 / 18</td>
<td>1 / 18</td>
<td>2 / 18</td>
<td>0 / 18</td>
</tr>
</tbody>
</table>

Although KeY aims to be sound, its default configuration has a false negative: it verified as correct the buggy code from Guava. By enabling the most faithful Java integer semantics, KeY can detect the bug. As expected, KeY rejected the other buggy test cases. However, in 8 cases it gave the identical verification failure on the fixed code, indicating that the verification failure had nothing to do with the bug. In only 2 cases did KeY verify the correct code without additional input. In 7 cases, KeY did not verify the correct code, but gave a different verification failure than it did on the buggy code. We found ourselves unable to interpret KeY’s output to locate the bug, but we count these 7 cases as true positives in table 5 because a KeY expert might be able to do so.

Clousot is effective—it detected most of the bugs. In its default configuration, Clousot failed to detect two bugs, instead reporting that they were correct code. One involved conversion from an array of bytes to a string:

```java
public static void doSession(InputStream stream, byte[] buf) {
    Contract.Requires(buf != null); int pos = stream.read(buf);
    string actual = Encoding.ASCII.GetString(buf, 0, pos);
    ...
}
```

The InputStream.read method returns -1 when the end of the file is reached. Client code should check it before using it as an index.

The other bug is the Guava bug involving overflow shown in section 4.1. Clousot has a command-line option for unsoundly checking overflow, which is disabled by default and is not available from the Visual Studio extension. With this command-line option enabled, Clousot finds the Guava bug.

Based on the results in table 5, we dropped FindBugs and KeY from further comparisons and focused on Clousot.

5.3 The Index Checker’s Test Suite

We ran Clousot on the Index Checker’s test suite, which is a set of Java files containing correct and incorrect code with expected warnings. The Index Checker passes this suite with no false positive warnings. The tests are mainly real-world code encountered in our case studies, not corner cases designed to highlight the Index Checker’s particular strengths.

We translated the test suite into C#. We skipped tests that use JDK classes without a direct equivalent in .NET, tests that use bottom types, and tests that use polymorphic qualifiers (which cannot be expressed by Code Contracts). We also skipped tests that check whether particular Java features are handled correctly. The resulting test suite consists of 163 C# files containing 4,608 lines of code.

One of the tests revealed a bug in Clousot. Clousot incorrectly reported that the asserted condition is false, but it is always true:

```java
int a = -1; int d = 2; int u = a / d; Contract.Assert(u >= -1);
```

Clousot issued 56 false positives. The most common causes for Clousot to issue a false positive were: computing indices by division, max, min, and bitwise-and operations (20 warnings), inferring that arrays equal by \(\equiv\) have the same length (10 warnings), and handling an array that contains all values of an enum type (5 warnings).

5.4 Clousot’s Test Suite

Clousot’s test suite has similar structure to the Index Checker’s, with C# files and expected warnings. We translated the parts of the suite that check array accesses to Java, and the associated code contracts to Index Checker annotations. This test suite contained 26 files with 2,633 lines of code. We did not write type qualifiers for contracts or assertions that could not be expressed in our type systems.

The Index Checker issued 136 warnings, of which 78 were true positives (expected warnings) and 58 false positives. Eleven of these warnings were because the code contracts could not be translated into the Index Checker’s type systems. For example, the Index Checker does not have an annotation to express that the return value of a method is equal to the value of a field. The most common reasons for false positives were: the Index Checker failed to refine types of local variables after a check or in a loop (18 warnings), code that increments an independent index variable in a loop over an array (8 warnings), using a multiply of an index as a size of a newly allocated array (6 warnings). Clousot’s test suite includes many linear inequalities that are more complex than those handled by the type system in section 2.7. This is probably because complex linear inequalities are a strength of Clousot’s SubPolyhedra domain. The Index Checker uses a cheaper analysis that issues few false positive warnings on the real-world code in our case studies, suggesting that Clousot’s test suite may be uncharacteristic of real-world code.

6 DISCUSSION

6.1 Limitations and Threats to Validity

Like any code analysis tool, the Index Checker only gives guarantees for code that it checks. The guarantee excludes native code, unchecked libraries like the JDK, and dynamically generated code. The Checker Framework handles reflective method calls soundly [4].
Type casts do not affect soundness: the Checker Framework issues a warning at every annotated type downcast. The Index Checker makes no guarantees about mutable-length data structures such as Java Lists (see section 6.2). The Index Checker makes no guarantees in the presence of overflow, though its best-effort analysis found some such errors in Guava (section 4.1).

Like any sound static analysis, the Index Checker cannot verify all correct code and produces some false positive warnings. A programmer must apply some other type of reasoning to such code; if the code is indeed safe, the programmer must suppress the warning. The Index Checker trusts that when a programmer suppresses a warning: (1) the code is safe, and (2) its annotations are correct.

Our results, while encouraging, may not generalize. The Index Checker might suffer more false positives or be harder to use if our subject programs are uncharacteristic. Over a dozen people have used the Index Checker, but its usability by programmers has not been established.

Our case studies demonstrate the Index Checker’s bug-finding power on well-tested, deployed code. Our case studies are a worst-case scenario for the tool’s usefulness. It would be both more useful and easier to use the Index Checker from the inception of a project. This would validate the program’s design and prevent bugs from ever entering the code. We chose our case studies to be array-heavy code; other code might not require as many annotations.

It is possible that our type system or the Index Checker implementation might contain errors. We have mitigated this danger by having multiple authors review every type rule and every line of code, and with a large test suite of 403 test cases and 9,904 lines of test code.

### 6.2 Fixed-length vs. Mutable-length Collections

Our type systems work for fixed-length collections, whose size does not change after the object is constructed. The Index Checker’s annotations can be written on type declarations and uses, which enables support both for JDK classes such as String and for user-defined classes. This works even for classes that do not extend Collection nor have one as a field, as long as the class’s abstraction represents more than one item. To specify that a class contains an array or collection field that acts as a delegate, the programmer writes @SomeLEN annotations (section 2.6).

Handling mutable-length collections is interesting future work that requires three techniques. The major part is tracking of indexing and lengths, which is described and implemented in this paper. The second part is handling operations that change the size of a data structure, such as add and remove. This is not difficult, though it requires specifications about side effects or their absence. The Checker Framework invalidates dataflow facts about expressions at all possible reassignments, including non-pure method calls. The third part is precisely tracking all aliasing, so that when a list’s length changes, the lengths of all (and only) aliased lists are also changed. This is challenging, but not specific to array indexing. New implementations of alias analyses could be substituted in as the community develops them. We do not know whether any existing analysis would be sufficiently precise for our needs.

### 6.3 Types vs. Expressions

The most closely related work, Clousot, checks specifications written as arbitrary C# expressions at statement boundaries. By contrast, Index Checker specifications are type qualifiers written on type uses. These differ in expressiveness, conciseness, and solver completeness.

Expressions are in general much more expressive than types. However, they are not strictly so. Unlike Code Contracts, types support polymorphism and variance. As an example, consider the following method fragment from the JFreeChart case study:

```java
Map<@NonNegative Integer, CategoryDataset> datasets;
List<@NonNegative Integer> getDatasetIndices(DatasetRenderingOrder order) {
    List<@NonNegative Integer> result = new ArrayList();
    for (Map.Entry<@NonNegative Integer, CategoryDataset> entry : datasets.entrySet()) {
        if (entry.getValue() != null)
            result.add(entry.getKey());
    } ...
    return result;
}
```

Clousot can use a quantifier to specify that the returned list should have non-negative elements:

```
Contract.ForAll( Contract.Result <List<int>>(), j -> j >= 0 )
```

However, Clousot cannot check that only non-negative numbers are added to the list (because the add method doesn’t have a precondition requiring that the argument is non-negative), and cannot prove this property. By contrast, the Index Checker can verify the whole method, using the fact that the values added to the list are keys from the dataset map which are non-negative, and similarly, wherever this method is called, use the fact that any integer retrieved from the list is non-negative.

Types are often more compact and easier to read. Expressing properties in the underlying programming language is convenient for tooling, but often verbose. Compare declaring `i as @IndexFor("a")` versus writing `Contract.Requires(i >= 0 & i < a.Length)`. Or, consider annotating an interface handling non-negative values:

```java
public interface Values {
    public @NonNegative int getItemCount();
    public Number getValue(@NonNegative int index);
}
```

The corresponding Code Contracts requires defining a ghost abstract class explicitly implementing the interface. The programmer must stub out each method just to write the specifications, which dramatically increases the size of the code: instead of modifying a few lines by adding annotations, the interface must be copied and then annotated. Combined with the more verbose syntax of Code Contracts (i.e. `Contract.Ensures(Contract.Result<int> i => i >= 0) instead of @NonNegative int`), the required changes to the code are almost an order of magnitude larger. Using the underlying language is sometimes convenient, but can significantly clutter the code.

A significant difference is that a programmer can predict if the Index Checker will verify a program’s correctness. When a programmer states an argument for why each array access in a program is legal, if the argument are expressible in the Index Checker’s language, then the Index Checker will verify the program is correct.
The design of a set of types that is sufficiently expressive, yet efficient to check, is one of our contributions.

7 RELATED WORK

The most common approach to array bound errors is dynamic checking, which crashes the program rather than permitting an unsafe operation. Most modern languages build this into the runtime system, and tools such as Purify [28] and Valgrind [40] do it for unsafe languages like C. These checks incur significant time or space overhead, despite research on reducing their cost [6, 45]. Other research aims to integrate bounds checks into unsafe languages [9, 15, 30, 38, 39, 49, 52]. Fundamentally, the goal of these approaches is to safely crash the program when an out-of-bounds array access would occur at run time. By contrast, the Index Checker imposes no run-time cost, and it prevents the program from crashing due to array bounds mistakes.

The classic dynamic approach of allocating, maintaining, and checking shadow bits can be performed statically via a dataflow analysis [17, 18]. An early example [48] was notable in not using the standard approach of unsound heuristics, but instead creating an infinite chain of approximations to the general loop invariant, using the “weakest liberal precondition”.

Bug-finders such as FindBugs [2] and Coverity [5] are easy to use and useful in finding some errors. Unlike the Index Checker, their heuristics are too weak to find all errors so they offer no guarantees.

Extended Static Checking [14, 22, 35], KeY [1], and Dafny [34] translate verification conditions into the language of powerful satisfiability engines or automated theorem provers, such as Z3 [13]. This is the dominant paradigm in bounds verification and in some other types of program analysis. Unlike the Index Checker, these tools suffer brittleness or instability: a small, meaning-preserving change within a method implementation may change the tool’s output from “verified” to “failed” or “timeout”, or might lead to different diagnostics in unrelated parts of the program [29, 36]. Scalability and usability are also challenges.

Wei et al. [50] evaluated different program analyses, including representations for integers and heap abstractions. They and we both found that polygons are expensive and not very helpful; simpler analyses can suffice.

Other researchers have applied dependent type systems to the problem of array bounds. With Xi and Pfening’s dependent type system for a subset of SML, [51], programmers write nearly arbitrary arithmetic linear inequalities, a type elaboration phase propagates them to unannotated expressions and collects a set of inequalities for the entire program, and then a solver for linear inequalities (such as Fourier variable elimination or the Omega Test [44]) is applied.

It was evaluated on 8 procedures, and the annotation overhead was 17% of lines or 31% of characters. Liquid types [46] use the same type system but provide better inference, reducing the annotation overhead by combining type inference with predicate abstraction. The Dsolv tool found a bug in the Bitv library, then verified the array safety of 58 of its 65 routines, requiring 65 lines of annotations to verify 30 array access operations. No information is given about the unverified routines. The Index Checker’s type system is more limited but works without an external solver. It has been designed to scale to real code with few false positives. ESPX [27] uses a dataflow analysis to find buffer overflows in C programs. It scales to large programs, but is unsound even on its own benchmarks. A type system with only upper and lower bounds [47] found 16 errors in one program. It is simpler than the Index Checker, but less general and has a higher false positive rate.

Abstract interpretation is as expressive as type systems [10], though in practice the two approaches lead to analyses that feel very different. Clousot or cccheck [20] has a number of similarities to the Index Checker: it is automated, checks programmer-written specifications, combines a set of interdependent analyses, works modularly, and performs some inference. Section 6.3 notes that Clousot’s abstract domains are richer than the Index Checker’s. Clousot was inspired by the limitations of theorem provers that we noted above.

The key challenge in designing a program analysis is selecting sufficiently precise abstractions that are still efficient. Clousot’s authors found that octagons and intervals were too imprecise, buckets exacerbated non-determinism, and polyhedra were too inefficient. They settled on disjunctions of intervals, upper bounds, pentagons [37], linear inequalities, and SubPolyhedra [32] (a novel abstraction that is as expressive as Polyhedra, but has more limited inference). Clousot is non-deterministic, since it must apply a timeout to its long-running analysis. The Clousot researchers do not discuss case studies in which they examined Clousot’s output, nor any bugs it revealed [19, 20, 32, 37]. Clousot issues a false positive at over 10% of array bounds [32]. We have found our choice of abstractions is simpler, more concise, more precise, faster, and effective in practice, and our implementation is more sound. Clousot made great strides, and the Index Checker makes significant further advances toward its vision.

8 CONCLUSION

The Index Checker occupies a new point in the design space of static analysis tools for array-bounds violations. It is fast and incremental. Unlike other scalable and simple tools, the Index Checker can find complex bugs and is sound—it provides a proof of correctness. Its technical approach is cooperating type systems, which are familiar to any developer writing in a statically typed language. We hope that this blend of the power of verification with the ease of use of a bug-finding tool will allow more developers to avoid array bounds violations.

Acknowledgments. Joe Santino, Wesley Cox, Grant Hughes, and Kevin Klein helped design and implement an early version of an index-checking type system. Abhishek Sangameswaran performed a case study with an early version of our tool. Jiaxin Xu and Werner Dietl created the Checker Framework’s interval analysis. Sam Elliott, Jared Roesch, Martin Schäf, Sandy Kaplan, John Toman, David Tarditi, Calvin Loncaric, Pavel Panchevka, Pavel Parízek, and the anonymous referees gave comments on this paper. This work was partially supported by Czech Science Foundation project 17-12465S.

