How to Build Static Checking Systems Using Orders of Magnitude Less Code

What kind of paper is this?

Fraser Brown   Andres Nötzli   Dawson Engler

Stanford University

{mlfbrown,noetzli,engler}@stanford.edu

Abstract

Modern static bug finding tools are complex. They typically consist of hundreds of thousands of lines of code, and most of them are wedded to one language (or even one compiler). This complexity makes the systems hard to understand, hard to debug, and hard to retarget to new languages, thereby dramatically limiting their scope.

This paper reduces the complexity of the checking system by addressing a fundamental assumption, the assumption that checkers must depend on a full-blown language specification and compiler front end. Instead, our program checkers are based on drastically incomplete language grammars ("micro-grammars") that describe only portions of a language relevant to a checker. As a result, our implementation is tiny—roughly 2500 lines of code, about two orders of magnitude smaller than a typical system. We hope that this dramatic increase in simplicity will allow developers to use more checkers on more systems in more languages.

We implement our approach in μchex, a language-agnostic framework for writing static bug checkers. We use it to build micro-grammar based checkers for six languages (C, the C preprocessor, C++, Java, JavaScript, and Dart) and find over 700 errors in real-world projects.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.3.4 [*Processors*]: Parsing

Keywords Micro-grammars; Parsing; Bug Finding; Static Analysis

1. Introduction

Research on bug finding has exploded in the past couple decades. While researchers have explored many approaches, the dominant, near-universal trend has been towards increasing complexity. The unsound dataflow analysis of early tools has given way to SAT solvers, theorem provers, and full-blown symbolic execution. While these new systems find interesting bugs, they also miss bugs due to their complexity. The more complex a tool is: the worse it scales; the harder it is to debug and understand; and the more assumptions it makes, limiting the programs it can check.

Our own work has followed this standard path, going from static analysis to symbolic execution. The complaints above are drawn both from our own difficulties dealing with the day-to-day cost of increasing complexity and from commiserations with other researchers. As a personal data point, the inherent difficulty in reasoning about symbolic execution—where each judgment is typically the result of thousands of previous low-level, feedback-directed decisions—meant that we once spent about six months uneasily trying to replicate *our own* results with *our own* tool with only mixed success [18]. As another, we have even struggled with the simpler approach of static analysis. During development, a commercial tool from our group switched the order in which it traversed the true and false branches of an `if` statement, leading to a disturbing change in the set of bugs the tool found (and missed). "Fixing" the problem was deemed too difficult, and the developers just rolled back the change, forfeiting the new bugs that it had found [6].

What are they referring to?

This paper is our reaction to these (and other) bitter experiences, and to the metastizing complexity throughout the field. We have taken the simplest bug finding approach—static analysis—and further reduced its complexity by about two orders of magnitude. This reduction is mostly due to one technical innovation, a new approach to parsing based on dramatically incomplete *micro-grammars*. Instead of depending on a full-blown language specification and compiler front end, our checkers are deliberately based on micro-grammars, partial grammars designed to extract checker-specific features that later passes analyze for bugs. Our checkers are often less than forty lines of code, and the parsers that they run on are less than 200. Checking an entirely new language—building its checker and associated

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872364

parser—might take merely a couple hours. We have written such checkers for C, C preprocessor, C++, Java, JavaScript, and Dart code. Furthermore, because our system uses micro-grammars instead of parsing and defining an entire language, it is tiny itself—roughly 2500 lines of code. It is not hard to understand a system that you can print on a dozen sheets of paper. As a result, it is not hard to customize, extend, rewrite, or reimplement that system. We hope that this stark simplification will allow more people to build and use static bug finders.

The main intellectual contribution of this paper is the concept of micro-grammars, along with techniques to define them in a principled way. Our main experimental results:

1. The approach works: as Table 1 shows, it is general and robust enough to find over 700 bugs in widely-used systems written in six different languages.
2. Despite ignoring most of the language, our checkers are not limited to checking simple syntactic properties. We reimplement two flow-sensitive checkers (§ 4) originally built using a traditional checking system, and show that despite being orders of magnitude simpler, they find a similar number of true bugs.
3. Micro-grammar checkers are not weak, and can help debug more sophisticated tools. When we check the same property as a state-of-the-art commercial tool and compare our results on the same code base, we find a roughly comparable number of bugs; we also find 190 bugs that the commercial tool misses. As a result, we think that our approach may be used as a safety net for more complex tools.
4. We repeatedly show (§ 5.1) that adopting an existing checker in our system to a new, previously unchecked language is a matter of writing a few lines of code (even when going from C++ to JavaScript!). Furthermore, writing a new checker from scratch is often in the tens of lines (§ 5.2). This ease is in contrast to traditional static checking systems, which, because of their detailed dependency on exact language semantics, might take months or years to retarget to a new language.
5. Micro-grammars allow us to run the exactly same checker and parser combination, unaltered, on many different languages (§ 6).

Checker	**System**	**Bugs**	**False**
Blocking while Locking	Linux (C)	124	0
Null Pointer	Linux (C)	233	12
	OpenJDK (Java)	42	3
	LLVM (C++)	10	1
	Firefox (C)	63	3
	Firefox (C++)	61	33
	Firefox (Java)	3	2
	Firefox (JavaScript)	25	6
	Node.js (JavaScript)	7	1
	All	444	61
Redundant Nested Macro Conditions	Linux (C macro)	7	0
	Firefox (C macro)	16	0
	All	23	0
Equals wo/ Hashcode	OpenJDK (Java)	73	2
	Dart SDK (Dart)	13	0
	All	93	2
Non-Observable Collection	Dart SDK (Dart)	2	0
Redundant Branches	Linux (C)	34	6
	OpenJDK (Java)	5 (8*)	1
	LLVM (C++)	2	0
	Firefox (C)	3	3
	Firefox (C++)	4	2
	All	48 (8*)	12
Redundant Conditions	Linux (C)	6	0
	OpenJDK (Java)	3	0
	Firefox (C++)	6	0
	LLVM (C++)	1	0
	All	16	0
Suspicious Loop Conditions	Linux (C)	5	2
	OpenJDK (Java)	2	0
	Firefox (C)	1	0
	Firefox (C++)	1	32‡
	LLVM (C++)	0	5
	Node.js (JavaScript)	2	0
	All	11	39
Total		761 (8*)	116
Spurious Null Check†	Dart SDK (Dart)	3	0
Incorrect Length Check†	Dart SDK (Dart)	152	0

* Generated code † Style and performance checkers
‡ Repeated patterns in related files

Table 1. Errors in all systems. We check Linux 4.4-rc7, Firefox 39.0, OpenJDK 8 b132, Node.js 0.12.7, LLVM 3.6.2 and Dart SDK 1.13.2 (including Core Libraries).

2. Overview

Traditional checking systems use front ends designed to parse a complete language grammar. These parsers reject any input that does not lead to a legal parse based on the full language specification. In contrast, our system is built on incomplete micro-grammars that intentionally ignore the vast majority of the input language.

Micro-grammar parsing differs from traditional parsing for bug finding in two ways:

I wonder if this is inherently better for html since
pages don't comply

1. A traditional parser returns an error (and perhaps tries to recover) when it fails to parse a program. In contrast, when a micro-grammar parser hits non-matching input, it simply slides forward by one token and tries again. This "sliding window" approach (similar to common regular expression matching) matches all constructs described by the micro-grammar and skips past those that the micro-grammar does not match. Ultimately, it consumes the entire input.

Whats the downside of this?

Component	Traditional	Using micro-grammars
Language	Full language grammar	Micro-grammar
Error handling	Exit or report	Advance one token, retry
Output	Complete representation	Incomplete representation

Table 2. Comparing our system to a traditional system.

2. Within a grammar rule, developers can perform fine-grained input skipping by using `wildcard` non-terminals—non-terminals that lazily match any input up to a suffix.

For example, we can write a micro-grammar that matches `if` statements but does not care about the details of expressions: "`S → if ( wildcard )`." When applied to a file with five `if` statements, the parser returns five parse trees. While a traditional parser creates a tree with (roughly) one node for every token, our parser returns a tree where lists of tokens match to each `wildcard` node. For example, parsing "`if (x == y)`" with the `if` statement micro-grammar would result in a normal `if` node and a `wildcard` node containing "`["x", "==", "y"]`". Table 2 summarizes these differences between our system and a traditional canonical one.

Implementation We implement the micro-grammar approach in µchex, a language-agnostic framework for writing static bug checkers. Like a traditional system, µchex breaks its input stream into tokens, feeds these lexed tokens to a parser, and runs a checker on the parser's output. In our system, though, checker developers can easily define their own lexers, parsers, and checkers, a process that sometimes takes less than a day (or even a couple hours). This shows one of the surprising benefits of ignoring most of a language; often, parts of languages overlap enough that they can share almost the same micro-grammar (§ 5 and § 6). Concretely, this means we can combine pieces of existing parsers and checkers to form new ones, even for syntactically different languages.

We now walk through how to check a new language from scratch. First, developers specify a lexer by supplying a list of keywords, operators, and a set of regular expressions for identifiers, literals, and comments. We use the Haskell Parsec library to make lexing easy; a typical specification is around ten lines of code [32].

Developers then build parsers recursively by defining a parser for each non-terminal and composing these parsers together to accept a micro-grammar. Adding a new non-terminal to a grammar equates to combining its parser with an existing one. For example, to create a parser for C's control flow, we first write parsers for `if` statements, `while` loops, `for` loops, etc. Each of these small parsers is around six lines of code, and we glue them together into a 156 line C control flow parser (Section 4.1.2 shows a concrete example of this process). The small parsers can be reused in other parsers; for example, our Dart control flow parser (§ 5.2.2) shares many parsers (and 75 lines of code) with the C version. For convenience, µchex ships with a library of combinable parsers.

```
S → F B         | s = f >> b
F → foo         | f = rsrvd "foo"
B → bar | baz   | b = rsrvd "bar" <|> rsrvd "baz"
```

Figure 1. The context-free grammar and its Parsec equivalent on the right.

The parsing step, like the lexing step, is built on top of the Haskell Parsec library. This compact (3475 LOC) parser combinator library supports infinite-lookahead parsers for $LL(k)$ grammars. It fits well with our approach because it makes combining parsers for different non-terminals easy. For example, Figure 1 shows how each non-terminal of a grammar can correspond to a single Parsec parser, and how those Parsec parsers can be combined.

Downside?

Finally, users build checkers using µchex's framework, which provides a set of built-in functions for common checking operations. Though µchex supports generic checking, we have tuned it especially for "belief-style" checkers [20]. Beliefs are facts implied by code. For example, "`x/y`" implies a belief that `y` is non-zero and an unlock "`unlock(l)`" implies a belief that `l` is locked. In general, if we assume that a programmer does not intend to crash their program, then a given program statement implies that they believe the statement's preconditions hold. Therefore, contradicting beliefs signify errors.

Discuss. What other?

In our system, the user implements a checking function that produces beliefs and then uses those beliefs to find bugs. µchex automatically applies the checking function to tree nodes and propagates the beliefs that it produces through the tree. It supports forward or backward flow-sensitive analyses as well as custom propagation methods. Users control how beliefs propagate through join points, and µchex provides libraries for computing domination and post-domination.

What are these?

To repeat: while we present belief-style checkers in this paper, and support them in our framework, the micro-grammar approach is not limited to this type of checking. Furthermore, while we chose to implement the system in Haskell, our approach is not limited to this language; in fact, we wrote our initial system in Python. Both versions are comparable in size and our core result—the dramatic reduction in the size of the checking system—is caused by micro-grammars, not by a specific implementation.

3. Micro-grammars

We discuss practical aspects of micro-grammars in Section 3.1 and define them more formally in Section 3.2.

3.1 Parsing with micro-grammars

Below, we discuss how we implement the two distinctive features of our approach: the wildcard non-terminals that we use to selectively ignore parts of a language and the "sliding window” strategy that advances by one token when it fails to parse.

Wildcards The simplest wildcard rule that µchex implements is `SkipTo`, which skips tokens until it reaches a suffix `P`. We construct a `SkipTo` parser with semantics corresponding to the following grammar:

```
AnyToken   → token, token ∈ language
SkipTo(P)  → P | AnyToken SkipTo(P)
```

`AnyToken` matches any token in the input, and `SkipTo` matches the suffix `P` or `AnyToken` followed by `P`. The following micro-grammar uses `SkipTo` to accept any substring that contains one “a” followed by zero or more tokens and ending in “`cd`”:

```
SkipB → a SkipTo(cd)
```

Given the input string “`abcd`”, the `SkipB` production would accept, and its wildcard non-terminal would match “`b`”. Given the input string “`abc`”, however, the `SkipB` production would not match, since “`abc`” does not fully contain `SkipB`’s suffix (“`cd`”).

Wildcard matching is lazy (as opposed to greedy), meaning that wildcards match the fewest number of tokens possible. Otherwise, any wildcard would necessarily consume the entire input. All wildcard rules (such as `SkipTo`) are parameterized by a suffix that defines the end of the wildcard matching. In our implementation, to decide whether a rule containing a wildcard applies, the parser does a lookahead and checks whether the rule suffix appears in the input. If not, that particular rule does not match. Tokens consumed by wildcard non-terminals are stored in special wildcard nodes in the parser output.

Sliding window approach When a µchex parser does not match, it slides forward by one token and retries; in other words, it always consumes the entire input, and matches input to productions whenever those productions accept. Without this mechanism, the parser might get stuck. For example, without “sliding window,” `SkipB` fails to parse “`XabcdXabcd`”, since the first token of the input is not “a”. With it, parsing “`XabcdXabcd`” with `SkipB` returns two identical parse trees (and accepts the entire input string).

Our approach semantically corresponds to adding the following rule to a micro-grammar with the top symbol `P`:

```
P' → SkipTo(P) P' | ϵ
```

Commit points The “sliding window” approach allows micro-grammars to ignore irrelevant parts of the input language. Clearly, though, badly-written micro-grammars may ignore too much, thereby skipping input that is relevant to the checker. For example, early in the project, we defined a `try` parser (incorrectly) so that its wildcard non-terminal could never find its suffix. As a result, we marched through the input, ignoring `try`s and sliding forward one token until `eof`. We solved this problem with *commit points*, warnings that sound when parsers consume too much. Users create commit points by specifying *commit tokens*, tokens that must lead to a successful match of the rule in which they appear. After users create a commit point, the parser yells each time it “slides forward” after consuming the commit token (e.g. `try`).

Diacuss

Compositional parsers µchex implements a feature called *compositional parsers* that allows users combine parsers by applying one parser to the output of another. It is usually easier to write a grammar for two different constructs separately and later compose these grammars (as opposed to intermixing them into a single micro-grammar). For example, writing a grammar for `if` conditions and the expressions that they contain is far harder than writing a single grammar for `if` conditions and a single grammar for expressions. µchex takes advantage of this by allowing users to run parsers on the `wildcard` node output of other parsers. Applying parsers to `wildcard` nodes replaces those nodes with subtrees, but has no effect on the other, non-`wildcard` nodes. We walk through an example of compositional parsers in Section 4.1.1.

Can be fine grained ove
time

3.2 Defining micro-grammars

Micro-grammars capture portions of the input that interest the checker, such as `for` loops or `try` blocks. Checkers depend on these input snippets being parsed correctly; otherwise, the checkers might be wrong, since they are operating on a mis-parse. To avoid this problem, we intuitively want the micro-grammar’s parse tree to look like a less detailed version of the original grammar’s parse tree—a version with the same structure where some subtrees are replaced by `wildcard` nodes. We formalize this intuitive definition below. Later, we give an example of a micro-grammar that violates the definition and discuss how it confounds the checker.

Suppose we are interested in writing a checker for a certain construct (an `if` statement or `for` loop, for example) that appears in a language G, where $G = (N, \Sigma, R, S)$, with N its set of non-terminals, Σ its terminals, R its productions, and S a start symbol [29]. Also, suppose that the grammar is structured in such a way that there is a non-terminal $C \in N$ that that models our construct of interest. We say that $G' = (N', \Sigma, R', S')$ is a micro-grammar of G iff the following two conditions are met:

1. The language $L(G)$ accepted by G is a subset of the language $L(G')$ accepted by G'
2. Let `MATCH` be a function that takes a non-terminal P and some input X in $L(G)$, and returns the set of all substrings of X that are matched by P. We require that there is a non-terminal $C' \in N'$ so that $\texttt{MATCH}(C, X) \subseteq \texttt{MATCH}(C', X)$ for all $X \in L(G)$.

We need the first element of the definition because, without it, our parsers would get stuck on any input that did not match any rule. This would cause them to potentially miss check-able constructs later in the source code.

The second condition describes the fact that we want our micro-grammars to have the same overall structure as the original grammar, but with some subtrees replaced by `wildcard` nodes. To make MATCH concrete, consider a grammar G for a C-like language and its non-terminal `ifStmt` that matches `if` statements:

```
IfStmt → if '(' Expr ')' '{' Body '}'
```

If G accepts the following code snippet `Sentence`:

```
int x = 4, y = 5;
if(x == y) { return x; }
x++;
if(x < y) { if (x < 0) { return 0; } return y; }
```

Then `MATCH(IfStmt, Sentence)` is a list of three substrings:

```
[ "if(x == y) { return x; }",
  "if(x < y) { if (x < 0) { return 0; } return y; }",
  "if (x < 0) { return 0; }" ]
```

In the following paragraph, we walk through an example of an incorrect micro-grammar and its effect on the checker.

Incorrect micro-grammars We want to write a micro-grammar to check `if` conditions in a language defined by the following grammar:

```
S    → if Cond Expr ';'
Cond → '(' Expr ')'
Expr → '(' Expr ')' | Expr '+' Expr | IntLiteral
```

This grammar describes `if` statements that check integer expressions. Based on this full grammar, we might be tempted to create the following micro-grammar:

```
S'    → if Cond' SkipTo(';')
Cond' → '(' SkipTo(')')
```

The problem with this micro-grammar is that the rule for `Cond'` matches the first closing parenthesis that it encounters. For example, for the input "`if ((a + b) * c) ;`", `Cond'` matches "`((a + b)`", while the original non-terminal, `Cond`, matches the entire condition: "`((a + b) * c)`". As a result, our checker will check an `if` statement that does not actually exist in the source code ("`a + b`"), potentially yielding a false positive. Furthermore, it might miss a true bug in the condition that it mis-parsed ("`((a + b) * c)`"). This problem arises because `Cond'` violates the second condition of the micro-grammar definition.

The correct micro-grammar for extracting conditions is:

```
S'      → if Cond' SkipTo(';')
Cond'   → '(' Balance
Balance → SkipTo(')' | Cond' Balance)
```

In contrast to the previous attempts, this micro-grammar balances parentheses to make sure that the non-terminal of interest (`Cond'`) matches the same substrings as the original non-terminal (`Cond`). Now, our checker only checks `if` statements as they appear in the source code.

4. How do micro-grammar based checkers compare to traditional ones?

Because our checkers only understand a fraction of the checked language, an obvious concern is that they either cannot implement more complex analyses or fail to find a compelling number of bugs. We address those concerns by:

1. Showing that µchex is not limited to detecting simple syntactic patterns by implementing two flow-sensitive checkers from our previous work [19, 20]. These checkers detect 420 errors in Linux and Firefox, two systems that have been extensively checked for over a decade.
2. Comparing one of these checkers' results to those of an advanced, widely used commercial system. Both systems find 43 of the same bugs; SystemX finds an additional 73 bugs we miss, but we find 190 bugs it misses.

These results, presented in the next two sections, demonstrate that micro-grammars are effective and useful despite their simplicity.

4.1 Reimplemented checkers

We implement a null pointer dereference checker (§ 4.1.1) and a deadlock checker (§ 4.1.2).

4.1.1 Null Pointer checker

Our Null Pointer checker uses belief analysis to detect potential null pointer dereferences. It flags when a dereferenced pointer "`*p`" (which implies a belief that `p` is non-null) is post-dominated by a check against `NULL` (which implies a belief that p could be null). While these checks are occasionally harmless, they often signify null pointer dereferences or even security vulnerabilities [13]. The following bug shows a common pattern that this checker finds—a dereference hoisted above a null check. It was copy-pasted eleven times in the same file:

```
    /* arch/ia64/sn/pci/pcibr/pcibr_reg.c */
199 union br_ptr __iomem *ptr = (union br_ptr __iomem *)
      ↪   pcibus_info->pbi_buscommon.bs_base;
200
201 if (pcibus_info) ...
```

The pointer `pcibus_info` is dereferenced (implying a belief that it is non-null) in line 199 and checked against null in line 201. To detect errors like this one, the checker does a forward analysis of each function's parse tree. It keeps track of pointers that are dereferenced and flags an error whenever a previously dereferenced pointer is checked.

Implementation The checker uses two parsers, one for control flow, the other for expressions.

The control flow parser is composed of several small parsers for different control flow constructs such as functions, `if` statements, `for` loops, etc. In total it is 156 lines of code. We reuse this parser repeatedly for other checkers in this paper (even checkers that operate on other languages in Section 6). It implements a micro-grammar with non-terminals for each control flow construct and semantics corresponding to:

```
S          → Function
Function   → Ident ('(') FnArgs SkipTo('{') Body '}'
FnArgs     → SkipTo(',' FnArgs | ')')
```

```
Body       → '{' Statements '}' | Statement
Statements → Statement Statements | ϵ
Statement  → IfStmt | WhileStmt Body | ... | Line
IfStmt     → if Balanced('(', ')') Body [Else]
Else       → else Body
... // more constructs
Line       → SkipTo(';')
```

In addition to parsing control flow it also parses straight-line statements such as "`int x = y;`" using the `Line` non-terminal. The parsers that correspond to each of these non-terminals are only three to nine lines of code. For example, the following parser code corresponds to the `IfStmt` non-terminal in the micro-grammar above:

```
ifParser = do
  position <- pRsrvd "if"
  cond <- balancedP
  thBr <- bodyParser
  elseBr <- elseParser
  return $ IfThenElse position cond thBr elseBr
```

The `ifParser` is composed of other parsers: `balancedP` for balanced parentheses and their content, `bodyParser` for the true branch body, and `elseBr` for the false branch body. This parser returns an `IfThenElse` tree node.

Shat if the order was the pther way?

A second 29-line parser extracts the structure of C expressions. We run this (compositional) parser *after* running the control flow parser (§ 3.2), using it to parse tokens that have matched to `wildcard` nodes. For example, before using the expression parser, the content of an `if` condition node might be:

```
wildcard: ["a", "==", "foo", "(", ")"]
```

After using the expression parser, its content would be:

```
binOp "==" ("a") (call "foo" [])
```

The end result of applying both parsers to the source code—first the control flow parser, next the expression parser—is a parse tree for C control flow and basic expressions.

The Null Pointer checker then operates on this tree, performing a forward, flow-sensitive traversal over each function's parse to compute a set of all pointers dereferenced (and therefore believe to be "not null") on *all* incoming paths. Each time the checker encounters a null pointer check "`p == NULL`" it looks up p in the "not null" set and emits an error if p is present.

The checker computes the "not null" set by recursively traversing the parse tree, and, at each node, pattern matching against patterns for dereferences and assignments. If a pointer is dereferenced, it is added to the "not null" set; if it is reassigned, it is removed from that set. For example, encountering a tree for the code phrase "`x = p->q`," our checker would add p to the "not null" set and remove x from the set.

The analysis splits at fork points (`if` statements, `switch` statements, etc) and, when these paths rejoin, the analysis adds new dereferences that occur on all paths and removes dereferences that were killed on at least one path.

Results We find 233 true bugs and twelve false positives in Linux, for a false positive rate of 5%; the original checker from [20] detected 102 bugs and 4 false positives (4% false positive rate). Our false positives arose mostly from correlations between the nullness of the dereferenced pointer and other variables and `for_each` macros. The checker also finds 63 bugs and 3 false positives in Firefox.

4.1.2 Deadlock checker

This checker flags instances where a potentially blocking function is called while interrupts are disabled, a spinlock is held, or a mutex is held. The first two cases can cause deadlock; the third is at least a performance bug. The following security bug in Linux is representative of the major errors that this checker finds:

```
    /* arch/frv/kernel/traps.c */
169 spin_lock_irq(&atomic_op_lock);
170 if (__get_user(z, p) == 0) {
171   if (__put_user(y, p) == 0)
172     goto done2;
173 goto error2;
174 }
175 spin_unlock_irq(&atomic_op_lock);
```

At line 169, the programmer locks `atomic_op_lock` and then immediately calls "`__get_user()`" and, perhaps, "`__put_user()`", which read and write memory from user space; these routines sleep if the user memory that they access is paged out. Because users can control residency, they can trigger this bug for a denial-of-service attack.

This checker is a twist on a deadlock checker from our prior work [19]. The original checker entered a `no-block` state when it detected a lock acquisition or interrupt disable operation. It flagged any blocking calls that it encountered in this `no-block` state. In hindsight, this checker missed errors when Linux's labyrinthine control flow and aggressive use of function pointers caused it to miss paths from acquisitions (or disables) to blocking operations.

We implement a twist on this checker as a safety net against its false negatives. Since our checker and the original check the same property in different ways, they will each find bugs that the other misses. As opposed to identifying blocking calls after locking or disabling calls, we flag variations on the following code pattern:

```
block();
unlock(l);
```

Here, "`unlock(l)`" implies the programmer believes `l` is locked on the current path, including the place where the `block` call was made.

Implementation Like the Null Pointer checker, the Deadlock checker does a forward analysis of parses for C control flow and expressions. Its main computed data structure is a set of beliefs about whether the program can block. When it encounters a blocking call, it adds a `can-block` belief to the set, and when it sees an unlocking call (or enabling call), it adds a `cannot-block` belief to the set. It clears the belief set when it encounters a `lock` or disabling call. The checker flags an error whenever the belief set contains both a `can-block` belief and a `cannot-block` belief.

	SystemX		Shared		µchex	
Type	Bugs	False	Bugs	False	Bugs	False
Intra-procedural	7	2	43	1	190	11
Inter-procedural	≤ 55	≥ 3	n/a		n/a	
Aliasing	≤ 11	≥ 0	n/a		n/a	
Total	≤ 73	≥ 5	43	1	190	11

Table 3. Comparison of bugs by type between SystemX and µchex for Linux 4.4-rc7. µchex currently does not support inter-procedural and aliasing analyses but manages to find more intra-procedural bugs. Most of the SystemX reports are not examined, thus we report the known numbers as bounds.

Results This checker finds 124 bugs in the Linux 4.4-rc7 source code and no false positives. The checker finds such a clean set of errors because it conservatively flags instances where a blocking call is made on *all* paths coming into an unlocking call, though blocking calls on *any* path are actually wrong. Compared to the results of the original implementation [19], we find a similar amount of bugs (124 instead of 123) on a newer version of the kernel, which is encouraging.

4.2 Comparison with commercial system

Micro-grammar based checkers are able to find bugs—but how do they compare to state-of-the-art static analysis tools? This section presents a case study comparing our Null Pointer checker's reports against the reports generated by a similar (but more advanced) checker implemented in a widely-used commercial tool, subsequently referred to as SystemX.[1] The company behind the tool has been developing static checkers for more than a decade, with a team that currently numbers in the low hundreds. SystemX detects bugs using an inter-procedural analysis with alias tracking and some amount of path sensitivity, while µchex does a simple intra-procedural analysis. Even though µchex is far simpler, it finds 44 out of 122 of SystemX's reports. Furthermore, and surprisingly, µchex also manages to find 190 bugs that SystemX does not. Table 3 summarizes these results.

Of course, the two tools' results are not entirely comparable. SystemX has been analyzing the kernel for over five years, and many of its most important reports have already been fixed, leaving only a residue of the total detected bugs behind. Still, we believe that our results, a direct comparison of SystemX and µchex on the same code base, show that micro-grammars are a useful approach for two reasons: 1) their results are similar to those of a more advanced system and 2) they can actually be used to improve that system. False negatives are a perennial problem for unsound checking tools. Front end based tool builders might debug their checkers by writing smaller, equivalent checkers in µchex (in thirty lines of code), or even constructing a corresponding micro-grammar based system themselves.

As a counter against potential bias in our results, we did not configure or run SystemX ourselves. The company configured their analysis for Linux, while a Linux maintainer who regularly checks the kernel with SystemX configured and built the analyzed kernel (version 4.4-rc7). He configured the checker to run with as many `CONFIG_` options enabled as viable (roughly 6900) while disabling as few as necessary (roughly 100), and built the kernel for the x86-64 architecture. We merely accessed the reports afterwards. We compared every report issued by SystemX against those found by µchex (rather than only a subset) to avoid selection bias.

Results Table 3 summarizes the result of our comparison: the Shared column describes categories and counts of bugs that both tools found, while the SystemX and µchex columns detail the distinct bugs that each tool found. In total, SystemX generated 122 bug reports (false and true), while µchex generated 245. The table tells a two-sided story: we find bugs that SystemX misses, but we also miss bugs that SystemX finds. In the next paragraphs, we first explore the reasons for our false negatives, and then discuss the mechanisms behind our true positives.

SystemX uses two analysis strategies that we do not, inter-procedural analysis and alias tracking, which account for 66 bugs; these bugs are undetectable without these two analysis strategies, and since µchex does not use them, it misses these errors. Based on our past experiences building checking tools, we believe simple inter-procedural analysis can be readily added to µchex. µchex could also implement trivial alias tracking, but non-trivial alias tracking is challenging without a full understanding of the checked language.

SystemX's inter-procedural analysis accounts for two types of µchex false negatives: bugs where the dereference appears in one function and the check in another (53 out of 55 cases) and bugs where µchex conservatively assumes that a function might reassign a pointer (two cases). The following code snippet from Linux illustrates the second type of false negative:

```
    /* drivers/staging/rts5208/rtsx.c */
446 if (chip->srb->sc_data_direction == ...)
    ...
469 else {
470   scsi_show_command(chip);
471   rtsx_invoke_transport(chip->srb, chip);
472 }
    ...
478 if (!chip->srb) ;
```

Even though "`chip->srb`" is dereferenced in line 446 and checked in line 478, µchex conservatively assumes that `scsi_show_command` could change the value of "`chip->srb`" (since `chip` is passed to `scsi_show_command`), so both the dereference and the null check may be correct. SystemX, on

[1] For confidentiality we omit the system name.

the other hand, analyzes the content of scsi_show_command, and flags an error on line 478 since the function does not alter "chip->srb."

µchex misses seven bugs that are detectable without additional analysis strategies. Three false negatives result from minor differences in how µchex and SystemX propagate beliefs through loops and gotos. Such differences are expected when comparing the results of different tools and could likely be resolved by replicating SystemX's propagation strategy (if it were public). The last four false negatives are due to our false positive suppression strategy of removing all "not null" beliefs when encountering C preprocessor directives. Without this suppression strategy, we find a less clean set of errors, but detect four more SystemX bugs.

While µchex misses a couple of bugs, it also finds 190 bugs that SystemX does not. The following error is an example:

```
    /* fs/fscache/object.c */
228 new_state = state->work(object, event);
241 if (state->work) {
242   if (unlikely(state->work == ((void *)2UL))) {
243     _leave(" [dead]");
244     return;
245   }
246   goto restart_masked;
247 }
```

In line 228, function pointer "state->work" is dereferenced. In line 241, it is checked against null (with no intervening assignments). Strangely, in the next line, it is also checked for equality against the integer literal 2UL. It is unclear why SystemX misses this error—we believe the error is simple enough that its developers would certainly want to know about their mistake.

Some of the false negatives in SystemX are caused by the fact that it only analyzes x86-64 builds: µchex finds 18 bugs in arch (in arch/alpha, arch/cris, arch/a64, and arch/x86), while the commercial tool finds only two, both in arch/x86. In general, µchex is not limited to checking a specific build configuration; it examines every function in every file of the source code. The Linux SystemX scan maintainer tries to approximate this by configuring the build with CONFIG_COMPILE_TEST, so SystemX at least checks non-x86 drivers. Otherwise, their true bug rate would be lower.

5. How hard is it to check a new language?

Because micro-grammars ignore most of the checked language, they allows us to check entirely new languages in a few lines of code, instead of the tens (or hundreds) of thousands of lines that a traditional approach requires. This section demonstrates how micro-grammars make it easy to:

1. Retarget existing checkers to a new language. In Section 5.1 we take the C Null Pointer checker from Section 4.1.1 and its associated parsers and modify them to find errors in C++, Java, and JavaScript. Table 4 summarizes the changes necessary to check these three different languages (and to find 211 bugs).
2. Create new checkers for new languages. In Section 5.2 we build: 1) a checker for the C preprocessor (CPP) that finds 23 bugs and 2) four checkers for Dart that find 15 bugs and 152 performance errors.

Language	Checking function (LOC)	Parser (LOC)
Base (C)	78	156
C++	+11	+16
Java	+1	+1
JavaScript	+2	+15

Table 4. Lines of code shared between the different versions of the Null Pointer checker (checking function and its parser). Line counts indicate changes from the language above to add support for a specific language. While line counts are a somewhat imprecise proxy for the amount of work required, the low counts are encouraging.

5.1 Adapting an existing checker to a new language

Bugs in one language often appear in others. Unfortunately, the structure of traditional checking tools makes it hard to retarget checkers to new languages. Creating a front end from scratch can take months or years. Even adapting a prebuilt front end is not easy, since front ends are complicated, and so is integrating one with a checking tool. As one data point, it took us years to add Java checking to our C static tool, both in an academic and later in an industry setting.

This section demonstrates that retargeting micro-grammar checkers can be nearly trivial by measuring how much effort it takes to port Section 4's Null Pointer checker from C to C++, Java, and JavaScript. For each language, our goal is to implement a useful checker, one that finds true bugs and few false positives, using the smallest number of modifications (akin to a minimum viable product). We discuss the adaptions and results for each target language and conclude with a brief discussion of the case study.

C++ Adapting the C Null Pointer checker to check C++ requires 27 lines of code, and the adapted checker finds 71 bugs. Without these adaptions, the original tool finds 50 of the same bugs (and 9 of the same false positives) on Firefox, but flags over one hundred additional false reports. To suppress these, the we implement support for try statements and templated identifiers within the micro-grammar. Finally, we adjust the checker to conservatively reset all "not null" beliefs when it encounters a function call that looks like it might be in the same class as the current code. Methods like these ("foo()" as opposed to "bar.foo()") can change the state of any member variable of the class; in other words, they might change the value of a pointer between the dereference and the check, invalidating our error report.

In contrast to C, C++ supports classes with operator overloading, which presents both a problem and an opportunity for µchex. We can end up with false positives if a class overloads the dereference (->) and logical negation (!) operators, since we do not know what beliefs the overloaded operators imply. On the other hand, developers have expectations about operator's behavior, and completely violating these assumptions is bad style. For example, "!" and "->" are overloaded for implementing smart pointers, pointers that ease memory management but follow traditional dereference and negation rules. µchex finds null smart pointer dereferences for free, while a traditional system would require quite a bit of modification to support checking them. We find a number of bugs involving Firefox's custom smart pointers (such as nsRefPtr and nsComPtr).

In total, the checker finds 61 bugs in Firefox and 33 false positives. Out of the false positives, 24 are caused by callbacks that change a member in the calling class. For example, "mActor->StartDestroy()" indirectly destroys mActor, which makes the subsequent null check valid. In LLVM, the checker finds 10 bugs and a single false positive. The checker finds the following bug in LLVM:

```
    /* lib/Transforms/InstCombine/InstCombineAddSub.cpp */
456 Value *Opnd0_0 = I0->getOperand(0);
457 Value *Opnd0_1 = I0->getOperand(1);
458 Value *Opnd1_0 = I1->getOperand(0);
459 Value *Opnd1_1 = I1->getOperand(1);
    ...
488 FastMathFlags Flags;
489 Flags.setUnsafeAlgebra();
490 if (I0) Flags &= I->getFastMathFlags();
491 if (I1) Flags &= I->getFastMathFlags();
```

The code checks I0 and I1 on lines 490 and 491 even though they are dereferenced on lines 456–459. In addition, the conditions do not match the dereferences in their bodies ("I->getFastMathFlags();").

Java The analogue of a null pointer dereference in C is a NullPointerException when performing operations on null object. Adapting the already-adapted C++ Null Pointer checker to check Java requires 2 lines of code; the only real change is representing the dereference operator as "." instead of "->."

The Java Null Pointer checker finds the following issue in OpenJDK:

```
/* jdk/src/share/classes/com/sun/crypto/provider/
    CipherCore.java */
int outputCapacity = output.length - outputOffset;
int minOutSize = (decrypting?
  (estOutSize - blockSize):estOutSize);
if ((output == null) ...) ...
```

Calling "output.length" implies that output cannot be null, whereas the check "output == null" implies a belief that output *can* be null. Therefore, we flag an error.

JavaScript Adapting the Java checker to JavaScript requires 17 lines of code that extend its micro-grammar with support for nested functions. Without this adaption, the JavaScript checker mis-parses functions to the point that the checker is unable to operate. To suppress false positives, we change the checker to only consider the left-hand sides of binary logical operator expressions as null checks. For example, in "a || b," we consider only a to have been checked. We do this because, in JavaScript, "expr1 || expr2" returns expr1 if it evaluates to true and expr2 otherwise (instead of returning a boolean value). This operator is commonly used to express that expr2 is a fallback, meaning that there is value to including expr2 even if it is guaranteed never to evaluate to false.

The JavaScript checker finds 25 bugs in Firefox and 7 in Node.js with 6 and one false positive, respectively. The following example shows a dereference just before a check in Firefox:

```
/* browser/devtools/styleinspector/computed-view.js */
this.sheet = rule.parentStyleSheet;

if (!rule || !this.sheet) ...
```

Discussion Our case study shows that it is possible to adapt a checker to new languages by slightly changing the checker and its micro-grammar. While these adapted checkers can be tuned further to find even more bugs, it is surprising that so few changes yield such useful results, especially compared to the amount of work that it would take to migrate a traditional checking system to a new language.

5.2 Writing a checker for a new language from scratch

We now demonstrate how easy it is to write a new checker from scratch for a new language by doing so for both Dart (§ 5.2.2) and CPP code (§ 5.2.1). These languages are very different: Dart is a new scripting language developed by Google, and CPP is an old, primitive language with many pitfalls. Nevertheless, the micro-grammar approach makes implementing checkers for both of these languages possible with very little code overhead.

5.2.1 C preprocessor (CPP)

When developers use the C preprocessor to conditionally compile code in big source files, the directives become distant and nested, making it challenging to keep track of all the active definitions ("if-def hell"). Our checker extracts all CPP conditional checks and flags logical contradictions. The following Linux example shows a representative error:

```
/* sound/pci/au88x0/au88x0_core.c */
#ifndef CHIP_AU8810
...
#ifdef CHIP_AU8810
  temp = (temp & 0xfeffffff) | ((f & 1) << 0x18);
  temp = (temp & 0xfdffffff) | ((f & 1) << 0x19);
#endif
...
#ifdef CHIP_AU8810
  temp =
    ((f & 1) << 0x18) | (temp & 0xfcfffffef) |
     ↪  FIFO_BITS;
#endif
...
#endif
```

Lines 878–1035 are guarded by a check that verifies that CHIP_AU8810 is not defined. Nested within this guarded piece of code, there are two pieces of code that are guarded by checks that verify that CHIP_AU8810 *is* defined, rendering the code within them useless. The checker finds 16 bugs in Firefox and 7 bugs in Linux with no false positives. We implement 40 lines of checker and parser to achieve these results.

We extract preprocessor directives and traverse their parse to determine whether conditions have been redundantly checked. The checker produces beliefs about whether a variable is `defined` or `undefined` and flags tautological and (often more important) contradictory checks. When the checking function encounters a check such as "`#ifdef foo`", it generates a belief `checked` for `foo`. Directives like "`#define foo`" or "`#undef foo`" remove that belief because they change `foo`. The checker conservatively resets all beliefs when encountering an "`#include`" directive, because the included file could define or undefine anything. In contrast to the previous belief-style checkers, this checker propagates beliefs path-sensitively: µchex propagates the belief that `foo` has been checked through the branches of an "`#ifdef`" but not beyond the corresponding "`#endif`".

5.2.2 Dart

The CPP language is old and relatively simple; this section shows that it is similarly easy to write a parser and checker for features of a new, much more sophisticated language. We check Dart, an optionally-typed language that compiles to JavaScript and has been used in production since 2013.

We did not conceive of the checkers in this section ourselves; in fact, none of us authors has ever written a line of Dart. Instead, we solicited a list of common errors from active Dart developers and created a checker for each one, validating its errors with experts. Clearly, even though these checkers are small and easy to build, they still find real-world errors that matter to developers.

Three of the checkers in the next paragraphs operate on the output of the same 207 line Dart control flow parser (where 75 lines are reused from the C version). One checker runs on an independent 7 line parser. We describe their implementations and the bugs that they find in the following.

Equals without hashcode This checker flags overridden Dart "==" methods that lack a corresponding `hashCode` methods. In Dart (as in Java), classes inherit an "==" (`equals`) method from the root of the of the class hierarchy. This default method only checks whether two objects are the same instance—whether they are the same exact object. Programmers can override the default "==" method with a custom implementation. Dart expects users to override the `hashCode` method when overriding "==," because equal objects should hash to the same value. Missing `hashCode` implementations may cause wrong behavior: for example, many elements that are equal according to the overridden method might end up in a different bucket in a `HashMap` because their `hashCodes` differ.

Our 28-line checking function runs on a control flow parse and does a backwards analysis of each Dart class. When it encounters a "==" method, it adds a `needs_hashCode` belief to the belief set. When it sees a `hashCode` method, it removes this belief. Finally, if it reaches the beginning of a class and the `needs_hashCode` belief remains, the checker flags an error. Below is a Dart class where the "==" function is overridden without also overriding `hashCode`:

```
/* sdk/pkg/analysis_server/lib/
   src/services/index/indexable_file.dart */
class IndexableFile implements IndexableObject {
...
@override
bool operator ==(Object object) =>
  object is IndexableFile && object.path == path;
}
```

The checker finds 13 instances of overriden "==" methods without `hashCode` and no false positives. We confirmed these errors with Dart developers. Seven of the errors that this tool found on a previous version of the analyzed code have been fixed (though independent of our reports). Our Java version of the checker finds 73 bugs and 2 false positives in OpenJDK.

Non-observable @Observable collections When writing Dart Polymer elements—Dart web elements that rely on the Polymer package—developers may use the "`@Observable`" annotation. Placed before data structures, this annotation enables a two-way binding between DOM nodes and object models. If programmers add the annotation to collections without making their contents observable (using the `toObservable` function or an `Observable` version of the collection), changes to that collection's elements in the DOM are not reflected in the data model for the collection.

The checker (26 lines of code) traverses the parsed Dart control flow, seeking annotation nodes. If it detects that a new collection object has been annotated, it verifies that that collection's value has been converted `toObservable` (or that the collection's name contains `Observable`, making it an observable collection).

We find two instances of this error in Dart code (out of twenty-two observable annotations in the code that we check):

```
/* sdk/runtime/observatory/lib/
   src/elements/flag_list.dart */
@observable List<ServiceMap> modifiedFlags =
  new List<ServiceMap>();
@observable List<ServiceMap> unmodifiedFlags =
  new List<ServiceMap>();
```

Style: spurious null check Though not an error, null checks in overridden "==" functions are not necessary, since Dart handles this case automatically. To find this style issue, we parse Dart control flow and implement a checker (9 lines of code) that finds "==" functions containing a null check. For each null check, the checker produces a warning.

Performance: incorrect length check In Dart, lists and list-backed data structures implement the `isEmpty` function, which runs in constant time. Despite this, many people check lists' "`.length`" against 0, a check that runs in linear time. This check usually results in a small performance hit but adds up when repeated. The checker consists of a separate parser (7 lines of code) and a checking function that flags everything the parser returns. It finds 152 instances of the issue in Dart code.

Discussion These results show that the micro-grammar approach is not limited to detecting low-level bugs in old languages—it can be just as readily applied to new, modern programming languages. Optimistically, our results suggest that micro-grammars could speed the adoption of new languages by enabling developers to easily write checkers for them. In fact, micro-grammars are a good fit for budding languages because they are robust to evolving language grammars. As languages mature, developers can adjust their checkers with very little overhead.

Finally, we were happily surprised by the fact that micro-grammars allowed us to find bugs not only in systems we do not understand, but also in *languages* we do not understand.

6. Does the same checker run on different languages' source code?

One of our most unexpected results is that the same checker and parser combination can find bugs in many different languages. The intuition behind this fact is that if languages share a common heritage, some portion of their grammars may be similar, even though they may otherwise differ significantly. Micro-grammars can capture the overlapping portion while ignoring the rest of the grammar.

This section explores this unexpected result by running the same checker, unaltered, charcter-for-character, on C, C++, Java, and JavaScript code. We detect programming logic errors in the form of tautological branching decisions. Such redundant or impossible operations often hint at high-level conceptual errors [20, 47, 49]. With the exception of a few stylized (and often commented) cases, programmers write code to perform useful work; if they perform an action, it is because they believe it serves a purpose. To detect these errors, we implement the following checkers:

Redundant Branches finds `if` statements with redundant branches: "`if (x) { a(); } else { a(); }`"

Redundant Conditions finds `if` statements with repeated conditions:
"`if (x) { a(); } else if (x) { b(); }`"

Suspicious Loop Condition finds `for` loop headers where the condition contradicts the increment:
"`for (i = 0; i < len; i--) ...`"

While these errors are thought of as simple, they can be some of the most difficult to diagnose [6], because they often cause code to silently compute an incorrect result.

6.1 Redundant Branches

The Redundant Branches checker flags instances where both true and false branches of an `if` statement perform syntactically identical actions. With some exceptions, checking a condition implies that the programmer must handle different cases as a result of that condition. Performing identical actions in both cases is suspicious.

The following error in LLVM (changed in version 3.7.0) is a good example:

```
  /* lib/Target/X86/AsmParser/X86AsmParser.cpp */
if (isParsingIntelSyntax()) {
  Operands.push_back(X86Operand::CreateReg(
    X86::DX, NameLoc, NameLoc));
  Operands.push_back(DefaultMemDIOperand(
    NameLoc));
} else {
  Operands.push_back(X86Operand::CreateReg(
    X86::DX, NameLoc, NameLoc));
  Operands.push_back(DefaultMemDIOperand(
    NameLoc));
}
```

Here, the LLVM assembler's parsing code checks if the input assembly uses Intel syntax, which it must handle differently (versus `AT&T` syntax). Unfortunately, both branches contain identical code, thereby leading to (potentially silent) mis-parses. Our checker, despite not understanding C++ and certainly not understanding what this program does, easily detects this bug.

The checker uses the C control flow parser. It examines each `if` node in the parse tree and if both branches are syntactically equal, it flags an error. The checker filters out `if` statements with empty consequent blocks (the surprisingly common "`if (x) ;`"). It also filters out errors that appear in chained `if` statements, because repeated branches often indicate multiple cases without a commitment to different actions in every case. This extra false positive filtration can be easily disabled.

The checker finds 48 true bugs and 12 false positives running unaltered on C, C++, Java, and JavaScript code. We find 8 additional instances in generated code in OpenJDK. Our true bugs often seem to result from copy-paste errors. Our false positives, on the other hand, mostly consist of instances where programmers use duplicate logic to document that different cases lead to the same actions. Programmers also occasionally insert these kinds of statements for future use. In practice, identical branch blocks are suspicious enough that they are commonly accompanied by a comment.

6.2 Redundant Conditions

This checker flags sequences of nested `if` statements where more than one `if` statement contains the same condition. The most straightforward errors that this checker finds mirror those in Section 6.1, where it appears that the programmer copied code but performed an incomplete rewrite. For example, the checker finds the following error in OpenJDK:

```
    /* nashorn/src/jdk/nashorn/internal/codegen/types/
        BooleanType.java */
131 } else if (to.isLong()) {
132   convert(method, OBJECT);
133   invokeStatic(method, JSType.TO_UINT32);
134 } else if (to.isLong()) {
135   convert(method, OBJECT);
136   invokeStatic(method, JSType.TO_LONG);
137 } else if ...
```

The condition "`to.isLong()`" is checked on lines 131 and 134. The code in the body of the first check, however, appears to handle `UInt32`s (instead of `Long`s). This has two ramifications: 1) `Long`s are handled as `UInt32`s and 2) `UInt32`s are not handled at all.

Like the Redundant Branches checker, this checker runs on parsed C control flow structures. For each `if` node in the program graph, it gathers all the chained `if` nodes attached to it. It flags an error if any of the conditions are not unique. It filters out duplicate cases with side-effects ("`if (x++)`").

This checker finds 16 errors in C, C++, Java, and JavaScript. The checker could potentially yield many false positives because it only does rudimentary filtering of conditions with side-effects. In practice, the filter is good enough to avoid all false positives on the systems that we check.

6.3 Suspicious Loop Condition

All four languages—C, C++, Java and JavaScript—have `for` loops, which they define similarly as containing four parts: the initialization, the condition, the afterthought, and the body. Often, developers use `for` loops to traverse a range of numbers. In this case, the condition defines either the lower or the upper bound of the range and the afterthought defines the direction of the traversal. Our Suspicious Loop Conditions checker identifies pairings of loop conditions and afterthoughts that are suspicious because the direction of change in the afterthought moves the value away from the condition. The following code is an error in OpenJDK:

```
/* jdk/test/java/io/pathNames/GeneralWin32.java */
for (char d = 'C'; d <= 'Z'; d--) {
  File df = new File(d + ":\\");
  if (df.exists()) return d;
}
```

Windows uses alphabetical identifiers for volumes. Instead of traversing these identifiers from "`C:`" to "`Z:`" in order to find an active drive, the `for` loop explores the range below "`C:`", which consists of illegal identifiers. Because of the narrow range of `char`, the loop will wrap around and terminate, silently skipping most of the intended lookups.

This checker uses the `for` loop header parser (which is also used in the C control flow parser):

```
S → for '(' SkipTo(';') SkipTo(';') SkipTo(')')
```

The checking function extracts variables that are compared with an expression (e.g. "`x < expr`") and the nature of that comparison (e.g. "<"). From the afterthought, it extracts variables that are incremented or decremented. Then, it checks that comparisons on a variable are consistent with the operation on that variable in the afterthought.

We run the unmodified checker on C, C++, Java and JavaScript code. The checker detects 11 true bugs and 39 false positives. With the exception of one of the false positives, the variable in question is an unsigned integer, so under- and overflow are defined behaviors, and the behavior may be intentional (though somewhat questionable).

7. Related Work

To the best of our knowledge, the concept of micro-grammars is new, as is the thorough, deliberate use of language incompleteness to simplify program checkers. Our empirical result that checkers can ignore the bulk of a programming language and yet be effective is viewed as "surprising" (to pick the most diplomatic word) by nearly everyone in the programming language community we have discussed it with.

Simplicity The systems community has a long tradition of arguing for and acting on simplicity. This ranges from broad principles such as KISS (and others compiled in [31]) and some forms of the end-to-end argument for moving functionality out of layers [43]. Some examples of artifacts that aimed to be more simple in useful ways than their predecessors are the RISC computer [38] and Unix [42] as a reaction to Multics [12] (and then Plan9 as a reaction to Unix complexity [40]). One way to view this paper is that we have extended the reasoning that systems should be simple to also apply to the tools that check them. In this context, many of the arguments for ease of modification, understanding, speed, and correctness made in prior work also hold.

Incompleteness Our sliding window approach for micro-grammars is similar to traditional parsers that perform error recovery [2, 26]. When a malformed input triggers a parse error in such parsers, the parser skips ahead until it finds a successful match. While our approach skips ahead and retries in a similar fashion when no rule applies, we additionally use bounded skipping within rules using wildcard nonterminals. Often error recovery in traditional parsers works well (perhaps because languages have somewhat evolved to enable it), which gives encouragement to our approach.

In the context of program checkers, the closest parallel to incomplete grammars is that some unsound static tools skip portions of code (functions, files, and directories) that they cannot parse (some tools can also skip paths in code they understand to make progress [30]). Of these tools, the closest analogue we know to our approach is the use of pre-compilation filters that remove code constructs (such as compiler-specific assembly statements) that the tool cannot parse [6]. We believe that skipping code is a common practice but often not reported because of negative connotations. While these prior approaches skip code, they combine the risks of incomplete parsing with the complexity of traditional approaches: rather than leveraging their incompleteness for simplicity, they use full-blown, complex front ends with the added challenge that they silently discard code, often in ways that are hard to reason about. In comparison, micro-grammars give orders of magnitude reduction in tool complexity, while also making the line between parsed and skipped code both explicit and more precise.

The use of incomplete grammars to parse programs can be viewed as similar to natural language processing (NLP) [9], where a key problem is to parse corpora (e.g., newspaper articles) using woefully incomplete grammars and do something useful with the result. The fact that the NLP field has obtained many interesting results using incomplete grammars perhaps makes it less of a surprise that micro-grammars can work for the more regular programming languages. NLP often benefits from the fact that many of the statements one cares about are *upward entailing* [21], i.e. missing information does not invalidate them: “Bob is 18” is true whether or not we miss that he is also African. A similar dynamic occurs with code checkers, which often only need a small set of program constructs to detect violations of their properties. For example, a lock checker may only need lock acquisition and release and control flow, but can ignore other statements in the program since they cannot invalidate the facts implied by these operations.

Finally, our deliberate strategy of skipping over code that the micro-grammar cannot parse has some similarities in spirit to failure oblivious computing [41], which discards the result of computations that caused errors (such as memory corruption or division by zero) in order to make progress.

Program Checking Using micro-grammars makes µchex and its checkers dramatically simpler than past systems for effective static bug finding. While line counts are not public, even early tools appear to be orders of magnitude more complex, including PREfix [30], ESP [16], ESC [22, 33], the leak detection analysis of Xie et al. [48], and Wagner’s security work [45, 46].

A key reason for the complexity in the past work is that from the very beginning, the standard approach to source code analysis has been to build tools on top of sophisticated language front ends. Examples for such front ends for C include `gcc`, EDG [25], `clang` [1], and CIL [34]. These front ends are complex. For example, the most widely-used commercial front end, EDG, consists of almost one million lines of code. As a consequence of using traditional front ends, even bug-finding tools known for their simplicity such as the PREfast system [3] (which performs only intra-procedural analysis and makes heavy use of annotations) are forced to be significantly larger than a micro-grammar approach. In addition, those tools can be limited by the capabilities of their front end: Despite years of effort `clang` still has issues compiling parts of the Linux kernel [10].

At the opposite end of the spectrum, there are many different approaches to dynamic bug finding. The simplest are concrete dynamic tools such as Purify [27] and Valgrind [36] taint analysis, including TaintCheck [37] and Dytan [11]. These side-step the need for a programming language specification, but instead need an analogous understanding of machine code, which empirically appears at least as difficult.

More recently, researchers have pushed dynamic analysis further by using symbolic execution [7], such as klee [8], DART [24], Woodpecker [15], DiSE [39], and CUTE [44]. Because these approaches execute code paths while reasoning about many (potentially all) possible values, they can check properties we cannot. Overall, these approaches are significantly more complex than traditional static analysis, which in turn is significantly more complex and heavier weight than our approach. As a result, while they can find some errors we cannot (the converse applies as well), they generally apply to significantly less code—one way to see this is to compare the hundreds of bugs we find to the typically order of magnitude fewer found in these other papers.

Ambitious researchers have combined static and dynamic analysis in various ways, such as CCured [35], SLAM [4, 5], Bandera [14], and FeaVer [28]. We view these approaches as somewhat orthogonal to ours, and hope that using micro-grammars could help reduce complexity in such systems.

Others have taken a language-based approach to prevent bugs, such as Vault [17] and Foster et al. [23]. Switching languages requires significant work on the part of the programmer. We view our work as relatively independent of language-based approaches, but hope that the ease of writing micro-grammar checkers for new languages might somewhat mitigate the stress of adopting them.

8. Conclusion

We challenge the long-standing assumption that static bug finding tools must be based on a complete language front end. In this paper, we find bugs despite the fact—and at times *because* of the fact—that our checkers are built on micro-grammars, incomplete fragments of a language. Because our system does not require a complete language specification, it allows developers to check new languages easily. Furthermore, the partial parses on which our checkers operate surprisingly often match constructs that appear identically in multiple languages; as a result, a checker and parser written for one language can run un- or slightly-altered on many others. Finally, since our design reduces the complexity of program checking by orders of magnitude, we hope it it will increase the number of developers writing checkers and checking systems—and the number of dangerous bugs detected.

Acknowledgments

We thank Ben Holtz, Dave Jones, Adam Bender, Keith Schwarz, and David Terei for their expertise; David Dill, Stephen Cooper, and the anonymous reviewers for their feedback; and Mary Jane Swenson for her endless assistance. This research is supported by DARPA Subcontract No. 1190029-276707.

References

[1] clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. 1986.

[3] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM.*, 2004.

[4] Thomas Ball and Sriram K Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the Seventh International SPIN Workshop on Model Checking of Software*, 2000.

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software*, 2001.

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

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6), 1975.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of Symp. on Operating Systems Design and Implementation*, 2008.

[9] Erik Cambria and Bebo White. Jumping NLP curves: A review of natural language processing research. In *IEEE Computational Intelligence Magazine*, 2014.

[10] clang. Bug 4068 - compiling the linux kernel with clang. `https://llvm.org/bugs/show_bug.cgi?id=4068`, 2009.

[11] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of International Symp. on Software Testing and Analysis*, 2007.

[12] Fernando J Corbató, Jerome H Saltzer, and Chris T Clingen. Multics: the first seven years. *Proceedings AFIPS 1972 SJCC*, 40, 1972.

[13] Jonathan Corbet. Fun with null pointers, part 1. `http://lwn.net/Articles/342330/`, 2009.

[14] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[15] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[16] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[17] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, 2001.

[18] Dawson Engler. Making finite verification of raw c code easier than writing a test case. Invited talk, International Conference on Runtime Verification, 2011.

[19] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation*, 2000.

[20] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[21] Kai Von Fintel. NPI-licensing, strawson-entailment, and context-dependency. In *Journal of Semantics*, 1999.

[22] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[23] Jeffrey S Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, 1999.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

[25] Edison Design Group. EDG C++ compiler front-end. `http://www.edg.com`.

[26] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide (2nd ed)*. 2008.

[27] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference*, 1992.

[28] Gerard J Holzmann and Margaret H Smith. Software model checking: Extracting verification models from source code. In *Invited Paper. Proceedings PSTV/FORTE99*, 1999.

[29] John E. Hopcroft, Rajeev Motwani, and Jeffery Ullman. *Introduction to Automata Theory, Languages, and Computation, Third Edition*. Pearson Education, Inc, 2007.

[30] Intrinsa. A technical introduction to PREfix/Enterprise. Technical report, Intrinsa Corporation, 1998.

[31] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 1983.

[32] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. 2002.

[33] K Rustan M Leino, Greg Nelson, and James B Saxe. ESC/Java user's manual. Technical note 2000-002, Compaq Systems Research Center, 2001.

[34] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction*, 2002.

[35] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of Symp. on Principles of Programming Languages*, 2002.

[36] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.

[37] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of Network and Distributed Systems Security Symp.*, 2005.

[38] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 21(1), 1985.

[39] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2011.

[40] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from bell labs. In *Proceedings of the Summer 1990 UKUUG Conference*, 1990.

[41] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, 2004.

[42] D.M. Ritchie, K. Thompson, Dennis M Ritchie, and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7), 1974.

[43] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), 1984.

[44] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[45] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[46] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, 2000.

[47] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[48] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the International Symp. on Foundations of Software Engineering (FSE)*, 2005.

[49] Yichen Xie and Dawson Engler. Using redundancies to find errors. *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 27(6), 2002.