ExceLint: Automatically Finding Spreadsheet Formula Errors

DANIEL W. BAROWY, Williams College
EMERY D. BERGER, University of Massachusetts Amherst
BENJAMIN ZORN, Microsoft Research

Spreadsheets are one of the most widely used programming environments, and are widely deployed in domains like finance where errors can have catastrophic consequences. We present a static analysis specifically designed to find spreadsheet formula errors. Our analysis directly leverages the rectangular character of spreadsheets. It uses an information-theoretic approach to identify formulas that are especially surprising disruptions to nearby rectangular regions. We present ExceLint, an implementation of our static analysis for Microsoft Excel. We demonstrate that ExceLint is fast and effective: across a corpus of 70 spreadsheets, ExceLint takes a median of 5 seconds per spreadsheet, and it significantly outperforms the state of the art analysis.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: Spreadsheets, error detection, static analysis

ACM Reference Format:
Daniel W. Barowy, Emery D. Berger, and Benjamin Zorn. 2018. ExceLint: Automatically Finding Spreadsheet Formula Errors. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 148 (November 2018), 26 pages. https://doi.org/10.1145/3276518

1 INTRODUCTION

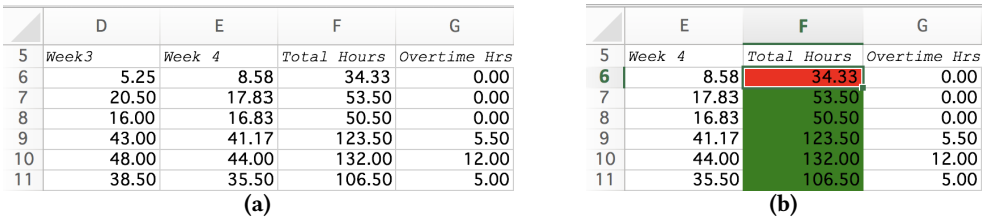
In the nearly forty years since the release of VisiCalc in 1979, spreadsheets have become the single most popular end-user programming environment, with 750 million users of Microsoft Excel alone [28]. Spreadsheets are ubiquitous in government, scientific, and financial settings [48].

Unfortunately, errors are alarmingly common in spreadsheets: a 2015 study found that more than 95% of spreadsheets contain at least one error [47]. Because spreadsheets are frequently used in critical settings, these errors have had serious consequences. For example, the infamous "London Whale" incident in 2012 led J.P. Morgan Chase to lose approximately $2 billion (USD) due in part to a spreadsheet programming error [16]. A Harvard economic analysis used to support austerity measures imposed on Greece after the 2008 worldwide financial crisis was based on a single large spreadsheet [52]. This analysis was later found to contain numerous errors; when fixed, its conclusions were reversed [37].

Spreadsheet errors are common because they are both easy to introduce and difficult to find. For example, spreadsheet user interfaces make it simple for users to copy and paste formulas or to drag

Authors' addresses: Daniel W. Barowy, Department of Computer Science, Williams College, dbarowy@cs.williams.edu; Emery D. Berger, College of Information and Computer Sciences, University of Massachusetts Amherst, emery@cs.umass.edu; Benjamin Zorn, Microsoft Research, ben.zorn@microsoft.com.

2475-1421/2018/11-ART148
https://doi.org/10.1145/3276518



	D	E	F	G
5	Week3	Week 4	Total Hours	Overtime Hrs
6	5.25	8.58	34.33	0.00
7	20.50	17.83	53.50	0.00
8	16.00	16.83	50.50	0.00
9	43.00	41.17	123.50	5.50
10	48.00	44.00	132.00	12.00
11	38.50	35.50	106.50	5.00

(a)

	E	F	G
5	Week 4	Total Hours	Overtime Hrs
6	8.58	34.33	0.00
7	17.83	53.50	0.00
8	16.83	50.50	0.00
9	41.17	123.50	5.50
10	44.00	132.00	12.00
11	35.50	106.50	5.00

(b)

Fig. 1. **ExceLint in action.** An excerpt of a buggy spreadsheet drawn from the CUSTODES corpus [17, 26]. (a) In Excel, errors are not readily apparent. (b) Output from ExceLint for a particular error: the suspected error is shown in red, and the proposed fix is shown in green. This is an actual error: the formula in `F6`, `=SUM(B6:E6)`, is inconsistent with the formulas in `F7:F11`, which omit `Week 4`.

on a cell to fill a column, but these can lead to serious errors if references are not correctly updated. Manual auditing of formulas is time consuming and does not scale to large sheets.

1.1 Contributions

Our primary motivation behind this work is to develop static analyses, based on principled statistical techniques, that automatically find errors in spreadsheets without user assistance and with high median precision and recall. This paper makes the following contributions.

- ExceLint's analysis is the first of its kind, operating without annotations or user guidance; it relies on a novel and principled information-theoretic static analysis that obviates the need for heuristic approaches like the bug pattern databases used by past work. Instead, it identifies formulas that cause surprising disruptions in the distribution of rectangular regions. As we demonstrate, such disruptions are likely to be errors.
- We implement ExceLint for Microsoft Excel and present an extensive evaluation using a commonly-used representative corpus of 70 benchmarks (not assembled by us) in addition to a case study against a professionally audited spreadsheet. When evaluated on its effectiveness at finding real formula errors, ExceLint outperforms the state of the art, CUSTODES, by a large margin. ExceLint is fast (median seconds per spreadsheet: 5), precise (median precision: 100%), and has high recall (median: 100%).

2 OVERVIEW

This section describes at a high level how ExceLint's static analysis works.

Spreadsheets strongly encourage a rectangular organization scheme. Excel's syntax makes it especially simple to use rectangular regions via so-called *range references*; these refer to groups of cells (e.g., `A1:A10`). Excel also comes with a large set of built-in functions that make operations on ranges convenient (e.g., `SUM(A1:A10)`). Excel's user interface, which is tabular, also makes selecting, copying, pasting, and otherwise manipulating data and formulas easy, as long as related cells are arranged in a rectangular fashion.

The organizational scheme of data and operations on a given worksheet is known informally as a *layout*. A *rectangular layout* is one in which related data or related operations are placed adjacent to each other in a rectangular fashion, frequently in a column. Prior work has shown that users who eschew rectangular layouts find themselves unable to perform even rudimentary data analysis tasks [11]. Consequently, spreadsheets that contain formulas are almost invariably rectangular.

ExceLint exploits the intrinsically rectangular layout of spreadsheets to identify formula errors. The analysis first constructs a model representing the rectangular layout intended by the user. Since there are many possible layouts and because user intent is impossible to know, ExceLint uses

simplicity as a proxy: the simplest layout that fits the data is most likely the intended layout. In this setting, formula errors manifest as aberrations in the rectangular layout. To determine whether such an aberration is likely to be an error, ExceLint uses the cell's position in the layout (that is, its context) to propose a "fix" to the error. If the proposed fix makes the layout simpler—specifically, by minimizing the entropy of the distribution of rectangular regions—then the cell is flagged as a suspected error.

The remainder of this section provides an overview of how each phase of ExceLint's analysis proceeds.

2.1 Reference Vectors

ExceLint compares formulas by their *shape* rather than syntactically; mere syntactic differences are often insufficient to distinguish the different computations. Consider a column of formulas in column `C` that all add numbers found in the same row of columns `A` and `B`. These formulas might be `=A1+B1`, `=A2+B2`, `=SUM(A3:B3)`, and so on. Each is syntactically different but semantically identical.

An important criterion used in ExceLint's analysis is the similarity of adjacent formulas. Nearly similar formulas often indicate an error. Large differences between formulas usually indicate entirely different computations. As a result, ExceLint needs a way to measure the "distance" between formulas, like `=A1+B1` and `=A1`.

ExceLint uses a novel vector-based representation of formulas we call *reference vectors* that enable both reference shape and formula distance comparisons. Reference vectors achieve this by unifying spatial and dependence information into a single geometric construct relative to the given formula (§3.2). Consequently, formulas that exhibit similar reference behaviors induce the same set of reference vectors.

Consider the formula `=A1+B1` located in cell `C1`: it has two referents, the cells `A1` and `B1`, so the reference vector for this formula consists of two vectors, `C1`⟶`A1` and `C1`⟶`B1`. ExceLint transforms these references into offsets in the Cartesian plane with the origin at the top left, $(-2, 0)$ and $(-1, 0)$. As a performance optimization, the analysis compresses each formula's set of vectors into a resultant vector that sums its component vectors; resultants are used instead of vector sets for formula comparisons. We call this compressed representation a *fingerprint* (§4.1.1).

ExceLint extracts reference vectors by first gathering data dependencies for every formula in the given spreadsheet. It obtains dependence information by parsing a sheet's formulas and building the program's dataflow graph [10, 19]. ExceLint can analyze all Excel functions.

2.2 Fingerprint Regions

As noted above, the syntax and user interfaces of spreadsheets strongly encourage users to organize their data into rectangular shapes. As a result, formulas that observe the same relative spatial dependence behavior are often placed in the same row or column. The second step of ExceLint's analysis identifies homogeneous, rectangular regions; we call these *fingerprint regions* (§3.3 and §4.1.1). These regions contain formulas with identical fingerprints, a proxy for identical reference behavior. Figure 2 shows a set of fingerprint regions for the spreadsheet shown in Figure 1a.

	D	E	F	G
5	*Week3*	*Week 4*	*Total Hours*	*Overtime Hrs*
6	5.25	8.58	34.33	0.00
7	20.50	17.83	53.50	0.00
8	16.00	16.83	50.50	0.00
9	43.00	41.17	123.50	5.50
10	48.00	44.00	132.00	12.00
11	38.50	35.50	106.50	5.00

Fig. 2. Fingerprint regions for the same spreadsheet shown in Figure 1a. Note that we color numeric data here the same shade of blue (column `D` and `E6`) to simplify the diagram. See §4.2.1 for details.

ExceLint computes fingerprint regions via a top-down, recursive decomposition of the spreadsheet. At each step, the algorithm finds the best rectangular split, either horizontally or vertically. This procedure is directly inspired by the ID3 decision tree algorithm [50]. The algorithm greedily partitions a space into a collection of rectangular regions. Once this decomposition is complete, the result is a set of regions guaranteed to be rectangular, homogeneous (consisting of cells with the same fingerprint), and be a low (near optimal) entropy decomposition of the plane.

2.3 Candidate Fixes and Fix Ranking

ExceLint identifies *candidate fixes* by comparing cells to all adjacent rectangular regions (§3.4). Each candidate fix is a pair composed of one or more suspect formulas and a neighboring set of formulas that exhibit different reference behavior. We call the pair a "fix" because it suggests a way to update suspect formulas such that their fingerprints match the fingerprints of their neighbors. The outcome of applying the fix is the creation of a larger rectangular region of formulas that all exhibit the same reference behavior.

The highest ranked candidate fixes pinpoint those formulas that are both similar to their neighbors and cause small drops in entropy when "fixed." Intuitively, such differences are likely to be the product of an error like failing to update a reference after pasting a formula. Because differences are small, they are easy to miss during spot checks. Large differences between pairs are usually not indicative of error; more often, they are simply neighboring regions that deliberately perform a different calculations.

2.4 Errors and Likely Fixes

Finally, after ranking, ExceLint's user interface guides users through a cell-by-cell audit of the spreadsheet, starting with the top-ranked cell (§4.2). Because broken formula behaviors are difficult to understand out of context, ExceLint visually pairs errors with their likely proposed fixes, as shown in Figure 1b.

3 EXCELINT STATIC ANALYSIS

This section describes ExceLint's static analysis algorithms in detail.

3.1 Definitions

Reference vectors: a *reference vector* is the basic unit of analysis in ExceLint. It encodes not just the data dependence between two cells in a spreadsheet, but also captures the spatial location of each def-use pair on the spreadsheet. Intuitively, a reference vector can be thought of as a set of arrows that points from a formula to each of the formula's inputs. Reference vectors let ExceLint's analysis determine whether two formulas point to the same relative offsets. In essence, two formulas are *reference-equivalent* if they induce the same vector set.

Reference vectors abstract over both the operation utilizing the vector as well as the effect of copying, or *geometrically translating*, a formula to a different location. For example, translating the formula `=SUM(A1:B1)` from cell `C1` to `C2` results in the formula `=SUM(A2:B2)` (i.e., references are updated). ExceLint encodes every reference in a spreadsheet as a reference vector, including references to other worksheets and workbooks. We describe the form of a reference vector below.

Formally, let f_1 and f_2 denote two formulas, and let v denote the function that induces a set of reference vectors from a formula.

Lemma 3.1. *f_1 and f_2 are reference-equivalent if and only if $v(f_1) = v(f_2)$.*

This property is intuitively true: no two formulas can be "the same" if they refer to different relative data offsets. In the base case, f_1 and f_1 are trivially reference-equivalent. Inductively, f_1

and f_2 (where $f_1 \neq f_2$) are reference-equivalent if there exists a translation function t such that $f_2 = t(f_1)$. Since reference vectors abstract over translation, $v(f_1) = v(f_2)$; therefore, reference equivalence also holds for the transitive closure of a given translation.

Reference vector encoding: Reference vectors have the form v = $(\Delta x, \Delta y, \Delta z, \Delta c)$ where Δx, Δy, and Δz denote numerical column, row, and worksheet offsets with respect to a given *origin*. The origin for Δx and Δy coordinates depends on their addressing mode (see below). Δz is 0 if a reference points on-sheet and 1 if it points off-sheet (to another sheet). Δc is 1 if a constant is present, 0 if it is absent, or -1 if the cell contains string data.

The entire dataflow graph of a spreadsheet is encoded in vector form. Since numbers, strings, and whitespace refer to nothing, numeric, string, and whitespace cells are encoded as degenerate *null vectors*. The Δx, Δy, and Δz components of the null vector are zero, but Δc may take on a value depending on the presence of constants or strings.

Addressing modes: Spreadsheets have two *addressing modes*, known as *relative addressing* and *absolute addressing*. For example, the reference `$A1` has an absolute horizontal and a relative vertical component while the reference `A$1` has a relative horizontal and an absolute vertical component.

In our encoding, these two modes differ with respect to their origin. In relative addressing mode, an address is an offset from a formula. In absolute addressing mode, an address is an offset from the top left corner of the spreadsheet. The horizontal and vertical components of a reference may mix addressing modes.

Addressing modes are not useful by themselves. Instead, they are annotations that help Excel's automated copy-and-paste tool, called Formula Fill, to update references for copied formulas. Copying cells using Formula Fill does not change their absolute references. Failing to correctly employ reference mode annotations causes Formula Fill to generate incorrect formulas. Separately encoding these references helps find these errors.

3.2 Computing the Vector-Based IR

The transformation from formulas to the vector-based internal representation starts by building a dataflow graph for the spreadsheet. Each cell in the spreadsheet is represented by a single vertex in the dataflow graph, and there is an edge for every functional dependency. Since spreadsheet expressions are purely functional, dependence analysis yields a DAG.

One fingerprint vector is produced for every cell in a spreadsheet, whether it is a formula, a dependency, or an unused cell. The algorithm first uses the dependency graph to identify each cell's immediate dependencies. Next, it converts each dependency to a reference vector. Finally, it summarizes the cell with a reference fingerprint.

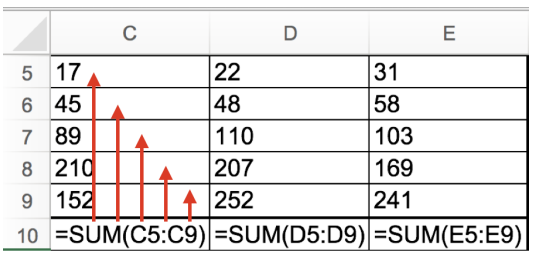


Fig. 3. **Reference vectors.** The formula in cell `C10` "points" to data in cells `C5:C9`. The cell's set of reference vectors are shown in red. One such vector, representing `C10`→`C5`, is $(0, 5, 0, 0)$.

For instance, the cell shown in Figure 3, `C10`, uses Excel's "range" syntax to concisely specify a dependence on five inputs, `C5...C9` (inclusive). As each reference is relatively addressed, the base offset for the address is the address of the formula itself, `C10`. There are no off-sheet references and the formula contains no constants, so the formula is transformed into the following set of reference vectors: $\{(0, -5, 0, 0) \ldots (0, -1, 0, 0)\}$. After summing, the fingerprint vector for the formula is $(0, -15, 0, 0)$.

A key property of fingerprint vectors is that other formulas with the same reference "shape" have the same fingerprint vectors. For example, `=SUM(D5:D9)` in cell `D10` also has the fingerprint $(0, -15, 0, 0)$.

3.3 Performing Rectangular Decomposition

As noted in the overview, spreadsheet user interfaces strongly encourage users to organize their data into rectangular shapes. Because spreadsheet operations expect these layouts, users who avoid using rectilinear layouts when building spreadsheets later discover that data in this form is extraordinarily difficult to manipulate [11]. As a result, formulas that perform the same operation (and are therefore reference-equivalent) are often placed in the same row or column.

Since slight deviations in references along a row or column strongly suggest errors, the rectangular decomposition algorithm aims to divide a spreadsheet into the user's intended rectangular regions and thus reveal deviations from them. To produce regions that faithfully capture user intent, our algorithm generates a rectangular decomposition with the following properties:

- Since users often use strings and whitespace in semantically meaningful ways, such as dividers between different computations, every cell, whether it contains a string, whitespace, a number, or a formula, must belong to exactly one rectangular region.
- Rectangular regions should be as large as possible while maintaining the property that all of the cells in that region have the same fingerprint vector.

```
EntropyTree(S)
 1  if |S| = 1
 2      return Leaf(S)
 3  else
 4      (l, t, r, b) = S
 5      x = argmin_{l≤i≤r} Entropy(S, i, true)
 6      y = argmin_{t≤y≤b} Entropy(S, i, false)
 7      p1 = (l, t, r, y) ; p2 = (l, y, r, b)
 8      e = Entropy(S, y, false)
 9      if Entropy(S, x, true) ≤ Entropy(S, y, false)
10          p1 = (l, t, x, b) ; p2 = (x, t, r, b)
11          e = Entropy(S, x, true)
12      if e = 0.0 and Values(p1) = Values(p2)
13          return Leaf(S)
14      else
15          t1 = EntropyTree(p1)
16          t2 = EntropyTree(p2)
17          return Node(t1, t2)
```

Fig. 4. EntropyTree decomposes a spreadsheet into a tree of rectangular regions by minimizing the sum total entropy vector fingerprint distributions across splits. See §3.3 for definitions.

Our rectangular decomposition algorithm performs a top-down, recursive decomposition that splits spreadsheet regions into subdivisions by minimizing the *normalized Shannon entropy* [56], an information-theoretic statistic. Specifically, it performs a recursive binary decomposition that, at each step, chooses the split that minimizes the sum of the normalized Shannon entropy of vector fingerprints in both subdivisions. We use normalized Shannon entropy since the binary partitioning process does not guarantee that entropy comparisons are made between equal-sized sets; normalization ensures that comparisons are well-behaved [12].

Normalized Shannon entropy is defined as:

$$\eta(X) = -\sum_{i=1}^{n} \frac{p(x_i) \log_b p(x_i)}{\log_b n}$$

where X is a random vector denoting cell counts for each fingerprint region, where x_i is a given fingerprint count, where $p(x_i)$ is the count represented as a probability, and where n is the total number of cells. Large values of η correspond with complex layouts, whereas small values correspond to simple ones. When there is only one region, η is defined as zero.

The procedure EntropyTree in Figure 4 presents the recursive rectangular decomposition algorithm. The algorithm returns a binary tree of regions, where a region is a 4-tuple consisting of the coordinates `(left, top, right, bottom)`. S is initially the entire spreadsheet. Each region contains only those cells with exactly the same fingerprint.

Entropy computes the normalized Shannon entropy of spreadsheet S along the split i which is an x coordinate if v = true, otherwise the coordinate is y (i.e., v controls whether a split is horizontal or vertical). $p1$ and $p2$ represent the rectangles induced by a partition. The normalized entropy of the empty set is defined as $+\infty$. Values returns the set of distinct fingerprint vectors for a given region. Finally, Leaf and Node are constructors for a leaf tree node and an inner tree node, respectively.

EntropyTree is inspired by the ID3 decision tree induction algorithm [50]. As with ID3, EntropyTree usually produces a good binary tree, although not necessarily the optimally compact one. Instead, the tree is decomposed greedily. In the worst case, the algorithm places each cell in its own subdivision. To have arrived at this worst case decomposition, the algorithm would have computed the entropy for all other rectangular decompositions first. For a grid of height h and width w, there are $\frac{h^2w^2+h^2w+hw^2+hw}{4}$ possible rectangles, so entropy is computed $O(h^2w^2)$ times.

Finally, regions for a given spreadsheet are obtained by running EntropyTree and extracting them from the leaves of the returned tree.

Adjacency coalescing: EntropyTree sometimes produces two or more adjacent regions containing the same fingerprint. Greedy decomposition does not usually produce a globally optimal decomposition; rather it chooses local minima at each step. *Coalescing* merges pairs of regions subject to two rules, producing better regions: (1) regions are adjacent, and (2) the merge is a contiguous, rectangular region of cells.

Coalescing is a fixed-point computation, merging two regions at every step, terminating when no more merges are possible. In the worst case, this algorithm takes time proportional to the total number of regions returned by EntropyTree. In practice, the algorithm terminates quickly because the binary tree is close to the ideal decomposition.

3.4 Proposed Fix Algorithm

When a user fixes a formula error, that formula's fingerprint vector changes. The purpose of the proposed fix algorithm is to explore the effects of such fixes. A *proposed fix* is an operation that mimics the effect of "correcting" a formula. The working hypothesis of the analysis is that surprising regions are likely wrong, and that unsurprising regions are likely correct. ExceLint's analysis leverages this fact to identify which cells should be fixed, and how. Those fixes that do not cause formulas to merge with existing regions are not likely to be good fixes.

Formally, a proposed fix is the tuple (s, t), where s is a (nonempty) set of *source cells* and t is a (nonempty) set of *target cells*. t must always be an existing region but s may not be; source cells may be borrowed from other regions. A proposed fix should be thought of as an operation that *replaces the fingerprints* of cells in s with the fingerprint of cells in t.

3.4.1 Entropy-Based Error Model. Not all proposed fixes are good, and some are likely bad. ExceLint's static analysis uses an error model to identify which fixes are the most promising. A good model helps users to identify errors and to assess the impact of correcting them.

We employ an entropy-based model. Intuitively, formula errors result in irregularities in the set of rectangular regions and so increase entropy *relative to the same spreadsheet without errors*. A proposed fix that reduces entropy may thus be a good fix because it moves the erroneous spreadsheet closer to the correct spreadsheet.

Since most formulas belong to large rectangular regions (§5.4), many formulas outside those regions are likely errors (§5.3). The entropy model lets the analysis explore the impact of fixing these errors—making the spreadsheet more regular—by choosing only the most promising ones which are then presented to the user.

Formally, m is a set of rectangular regions. A set of proposed fixes of size n yields a set of new spreadsheets $m'_1 \ldots m'_n$, where each m'_i is the result of one proposed fix $(s,t)_i$. The *impact* of fix the $(s,t)_i$ is defined as the difference in normalized Shannon entropy, $\delta\eta_i = \eta(m_i) - \eta(m)$.

Positive values of $\delta\eta_i$ correspond to increases in entropy and suggest that a proposed fix is bad because the spreadsheet has become more irregular. Negative values of $\delta\eta_i$ correspond to decreases in entropy and suggest that a proposed fix is good.

Somewhat counterintuitively, fixes that result in large decreases in entropy are worse than fixes that result in small decreases. A fix that changes large swaths of a spreadsheet will result in a large decrease in entropy, but this large-scale change is not necessarily a good fix for several reasons. First, we expect bugs to make up only a small proportion of a spreadsheet, so fixing them should result in *small* (but non-zero) decreases in entropy. The best fixes are those where the prevailing reference shape is a strong signal, so corrections are minor. Second, large fixes are more work. An important goal of any bug finder is to minimize user effort. Our approach therefore steers users toward those hard-to-find likely errors that minimize the effort needed to fix them.

3.4.2 Producing a Set of Fixes. The proposed fix generator then considers all fixes for every possible source s and target t region pair in the spreadsheet. A naïve pairing would likely propose more fixes than the user would want to see. In some cases, there are also more fixes than the likely number of bugs in the spreadsheet. Some fixes are not independent; for instance, it is possible to propose more than one fix utilizing the same source region. Clearly, it is not possible to perform both fixes.

As a result, the analysis suppresses certain fixes, subject to the conditions described below. These conditions are not heuristic in nature; rather, they address conditions that naturally arise when considering the kind of dependence structures that can arise when laid out in a 2D grid. The remaining fixes are scored by a fitness function, ranked from most to least promising, and then thresholded. All of the top-ranked fixes above the threshold are returned to the user.

The cutoff threshold is a user-defined parameter that represents the proportion of the worksheet that a user is willing to inspect. The default value, 5%, is based on the observed frequency of spreadsheet errors in the wild [10, 47]. Users may adjust this threshold to inspect more or fewer cells, depending on their preference.

Condition 1: Rectangularity: Fixes must produce rectangular layouts. This condition arises from the fact that Excel and other spreadsheet languages have many affordances for rectangular composition of functions.

Condition 2: Compatible Datatypes: Likely errors are those identified by fixes m_i that produce small, negative values of $\delta\eta_i$. Nonetheless, this is not a sufficient condition to identify an error. Small regions can belong to data of any type (string data, numeric data, whitespace, and other formulas). Fixes between regions of certain datatypes are not likely to produce desirable effects. For instance, while a string may be replaced with whitespace and vice-versa, neither of these proposed fixes have any effect on the computation itself. We therefore only consider fixes where both the source and target regions are formulas.

Condition 3: Inherently Irregular Computations: A common computational structure is inherently unusual in the spreadsheet domain: aggregates.

An *aggregate* consists of a formula cell f_a and a set of input cells $c_0, \ldots, c_n$ such that f_a refers exclusively to $c_0, \ldots, c_n$. This computational form is often seen in functional languages, particularly in languages with a statistical flavor. Excel comes with a large set of built-in statistical functions, so functions of this sort are invoked frequently. Examples are the built-in functions `SUM` and `AVERAGE`. f_a often sits adjacent to $c_0, \ldots, c_n$, and so f_a appears surprising when compared to $c_0, \ldots, c_n$. Without correction, f_a would frequently rank highly as a likely error. Since aggregates are usually false positives, no fix is proposed for formulas of this form.

Note that this restriction does not usually impair the ability of the analysis to find errors. The reason is that the relevant comparison is not between an aggregate formula and its inputs, but between an aggregate formula and *adjacent aggregate formulas*.

For example, the analysis can still find an off-by-one error, where an aggregate like `SUM` refers to either *one more* or *one fewer* element. Let $b_0, \ldots, b_n$ be a rectangular region, and let f_b be a formula that incorrectly aggregates those cells. Let $c_0, \ldots, c_n$ be a rectangular region adjacent to $b_0, \ldots, b_n$, and let f_c be a formula adjacent to f_b that *correctly* aggregates $c_0, \ldots, c_n$. Then if f_b refers to $b_0, \ldots, b_n, d$ (one more) or $b_0, \ldots, b_{n-1}$ (one fewer), f_b is a likely error. Because the analysis only excludes proposed fixes for aggregates that refer to $c_0, \ldots, c_n$, we still find the error since f_b will eventually be compared against f_c.

3.4.3 Ranking Proposed Fixes. After a set of candidate fixes is generated, ExceLint's analysis ranks them according to an impact score. We first formalize a notion of "fix distance," which is a measure of the similarity of the references of two rectangular regions. We then define an impact score, which allows us to find the "closest fix" that also causes small drops in entropy.

Fix distance: Among fixes with an equivalent entropy reduction, some fixes are better than others. For instance, when copying and pasting formulas, failing to update one reference is more likely than failing to update all of them, since the latter has a more noticeable effect on the computation. Therefore, a desirable criterion is to favor smaller fixes using a location-sensitive variant of vector fingerprints.

We use the following distance metric, inspired by the earth mover's distance [45]:

$$d(x, y) = \sum_{i=1}^{n} \sqrt{\sum_{j=1}^{k} (h_s(x_i)_j - h_s(y_i)_j)^2}$$

where x and y are two spreadsheets, where n is the number of cells in both x and y, where h_s is a location-sensitive fingerprint hash function, where i indexes over the same cells in both x and y, and where j indexes over the vector components of a fingerprint vector for fingerprints of length k. The intuition is that formulas with small errors are more likely to escape the notice of the programmer than large swaths of formulas with errors, thus errors of this kind are more likely left behind. Since we model formulas as clusters of references, each reference represented as a point in space, then we can measure the "work of a fix" by measuring the cumulative distance it takes to "move" a given formula's points to make an erroneous formula look like a "fixed" one. Fixes that require a lot of work are ranked lower.

Entropy reduction impact score: The desirability of a fix is determined by an *entropy reduction impact score*, S_i. S_i computes the potential improvement between the original spreadsheet, m, and the fixed spreadsheet, m_i. As a shorthand, we use d_i to refer to the distance $d(m, m_i)$.

$$S_i = \frac{n_t}{-\delta\eta_i d_i}$$

where n_t is the size of the target region, $\delta\eta_i$ is difference in entropy from m_i to m, and d is the fix distance.

Since the best fixes minimize $-\delta\eta_i$, such fixes maximize S_i. Likewise, "closer" fixes according to the distance metric also produce higher values of S_i. Finally, the score leads to a preference for fixes whose "target" is a large region. This preference ensures that the highest ranked deviations are actually rare with respect to a reference shape.

4 EXCELINT IMPLEMENTATION

ExceLint is written in C# and F# for the .NET managed language runtime, and runs as a plugin for Microsoft Excel (versions 2010-2016) using the Visual Studio Tools for Office framework. We first describe key optimizations in ExceLint's implementation (§4.1), and then discuss ExceLint's visualizations (§4.2).

4.1 Optimizations

Building an analysis framework to provide an interactive level of performance was a challenging technical problem during ExceLint's development. Users tend to have a low tolerance for tools that make them wait. This section describes performance optimizations undertaken to make ExceLint fast. Together, these optimizations produced orders of magnitude improvements in ExceLint's running time.

4.1.1 Reference fingerprints. Reference vector set comparisons are the basis for the inferences made by ExceLint's static analysis algorithm. The cost of comparing a vector set is the cost of comparing two vectors times the cost of set comparison. While set comparison can be made reasonably fast (e.g., using the union-find data structure), ExceLint utilizes an even faster approximate data structure that allows for constant-time comparisons. We call this approximate data structure a *reference fingerprint*. Fingerprint comparisons are computationally inexpensive and can be performed liberally throughout the analysis.

Definition: A vector fingerprint summarizes a formula's set of reference vectors. Let f denote a formula and v a function that induces reference vectors from a formula. A vector fingerprint is the hash function: $h(f) = \sum_{i \in v(f)} i$ where $\sum$ denotes vector sum.

When two functions x and y with disjoint reference vector sets $r(x)$ and $r(y)$ have the same fingerprint, we say that they *alias*. For example, the fingerprint $(-3, 0, 0, 0)$ is induced both by the formula `=SUM(A1:B1)` in cell `C1` and the formula `=ABS(A1)` in cell `D1`, so the two formulas alias. Therefore, Lemma 3.1 does not hold for fingerprints. Specifically, only one direction of the relation holds: while it is true that two formulas with different fingerprints are guaranteed not to be reference-equivalent, the converse is not true.

Fortunately, the likelihood of aliasing $P[h(f_1) = h(f_2) \wedge v(f_1) \neq v(f_2)]$ is small for fingerprints, and thus the property holds with high probability. Across the spreadsheet corpus used in our benchmarks (see Section 5), on average, 0.33% of fingerprints in a workbook collide (median collisions per workbook = 0.0%).

The low frequency of aliasing justifies the use of fingerprints compared to exact formula comparisons. ExceLint's analysis also correctly concludes that expressions like `=A1+A2` and `=A2+A1` have the same reference behavior; comparisons like this still hold with the approximate version.

4.1.2 Grid preprocessing optimization. One downside to the EntropyTree algorithm described in Section 3 is that it can take a long time on large spreadsheets. While spreadsheets rarely approach the maximum size supported in Microsoft Excel (16,000 columns by 1,000,000 rows), spreadsheets with hundreds of rows and thousands of columns are not unusual. EntropyTree is difficult to parallelize because binary splits rarely contain equal-sized subdivisions, meaning that parallel workloads are imbalanced.

Nonetheless, one can take advantage of an idiosyncrasy in the way that people typically construct spreadsheets to dramatically speed up this computation. People frequently use contiguous, through-spreadsheet columns or rows of a single kind of value as delimiters. For example, users often separate a set of cells from another set of cells using whitespace.

By scanning the spreadsheet for through-spreadsheet columns or rows of equal fingerprints, the optimization supplies the rectangular decomposition algorithm with smaller sub-spreadsheets which it decomposes in parallel. Regions never cross through-spreadsheet delimiters, so preprocessing a spreadsheet does not change the outcome of the analysis.

In our experiments, the effect of this optimization was dramatic: after preprocessing, performing static analysis on large spreadsheets went *from taking tens of minutes to seconds*. Scanning for splits is also inexpensive, since there are only O(width+height) possible splits. ExceLint uses all of the splits that it finds.

4.1.3 Compressed vector representation. In practice, subdividing a set of cells and computing their entropy is somewhat expensive. A cell address object in ExceLint stores not just information relating to its x and y coordinates, but also its worksheet, workbook, and full path on disk. Each object contains two 32-bit integers, and three 64-bit managed references and is therefore "big". A typical analysis compares tens or hundreds of thousands of addresses, one for each cell in an analysis. Furthermore, a fingerprint value for a given address must be repeatedly recalled or computed and then counted to compute entropy.

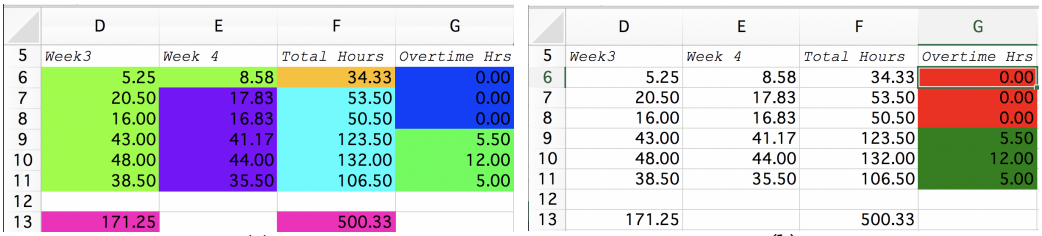
Another way of storing information about the distribution of fingerprints on a worksheet uses the following encoding, inspired by the optimization used in FlashRelate [11]. In this scheme, no more than f bits are stored for each address, where f is the number of unique fingerprints. f is often small, so the total number of bitvectors stored is also small. The insight is that the number of fingerprints is small relative to the number of cells on a spreadsheet.

For each unique fingerprint on a sheet, ExceLint stores one bitvector. Every cell on a sheet is given exactly one bit, and its position in the bitvector is determined by a traversal of the spreadsheet. A bit at a given bit position in the bitvector signifies whether the corresponding cell has that fingerprint: 1 if it does, 0 if not.

The following bijective function maps (x, y) coordinates to a bitvector index: $\text{Index}_s(x, y) = (y-1) \cdot w_s + x - 1$ where w_s is the width of worksheet s. The relation subtracts one from the result because bitvector indices range over $0 \ldots n-1$ while address coordinates range over $1 \ldots n$.

Since the rectangular decomposition algorithm needs to compute the entropy for subdivisions of a worksheet, the optimization needs a low-cost method of excluding cells. ExceLint computes masked bitvectors to accomplish this. The bitvector mask corresponds to the region of interest, where 1 represents a value inside the region and 0 represents a value outside the region. A bitwise `AND` of the fingerprint bitvector and the mask yields the appropriate bitvector. The entropy of subdivisions can then be computed, since all instances of a fingerprint appearing outside the region of interest appear as 0 in the subdivided bitvector.

With this optimization, computing entropy for a spreadsheet largely reduces to counting the number of ones present in each bitvector, which can be done in $O(b)$ time, where b is the number of bits set [58]. Since the time cost of setting bits for each bitvector is $O(b)$ and bitwise `AND` is $O(1)$, the



	D	E	F	G
5	Week3	Week 4	Total Hours	Overtime Hrs
6	5.25	8.58	34.33	0.00
7	20.50	17.83	53.50	0.00
8	16.00	16.83	50.50	0.00
9	43.00	41.17	123.50	5.50
10	48.00	44.00	132.00	12.00
11	38.50	35.50	106.50	5.00
12				
13	171.25		500.33	

(a)

	D	E	F	G
5	Week3	Week 4	Total Hours	Overtime Hrs
6	5.25	8.58	34.33	0.00
7	20.50	17.83	53.50	0.00
8	16.00	16.83	50.50	0.00
9	43.00	41.17	123.50	5.50
10	48.00	44.00	132.00	12.00
11	38.50	35.50	106.50	5.00
12				
13	171.25		500.33	

(b)

Fig. 5. **Global view**: (a) In the global view, colors are mapped to fingerprints so that users equate color with reference equivalence. For example, the block of light green cells on the left are data; other colored blocks represent distinct sets of formulas. Cells `G6:G8`, for example, are erroneous because they incorrectly compute overtime using only hours from Week 1. **Guided audit**: (b) ExceLint's guided audit tool flags `G6:G8` (red) and suggests which reference behavior should have been used (`G9:G11`, in green).

total time complexity is $O(f \cdot b)$, where f is the number of fingerprints on a worksheet. Counting this way speeds up the analysis by approximately 4×.

4.2 Visualizations

ExceLint provides two visualizations that assist users to find bugs: the *global view* (§4.2.1) and the *guided audit* (§4.2.2). Both tools are based on ExceLint's underlying static analysis.

4.2.1 Global View. The *global view* is a visualization for finding potential errors in spreadsheets. The view takes advantage of the keen human ability to quickly spot deviations in visual patterns. Another example of ExceLint's global view the the running example from Figure 1a is shown in Figure 5. The goal of the global view is to draw attention to irregularities in the spreadsheet. Each colored block represents a contiguous region containing the same formula reference behavior (i.e., where all cells have the same fingerprint vector).

While the underlying decomposition is strictly rectangular for the purposes of entropy modeling, the global view uses the same color in its visualization anywhere the same vector fingerprint is found. For example, all the numeric data in the visualization are shown using the same shade of blue, even though each cell may technically belong to a different rectangular region (see Figure 5a). This scheme encourages users to equate color with reference behavior. Whitespace and string regions are not colored in the visualization to avoid distracting the user.

The global view chooses colors specifically to maximize perceptual differences (see Figure 5a). Colors are assigned such that adjacent clusters use complementary or near-complementary colors. To maximize color differences, we use as few colors as possible. This problem corresponds exactly to the classic graph coloring problem.

The color assignment algorithm works by building a graph of all adjacent regions, then colors them using a greedy coloring heuristic called largest degree ordering [59]. This scheme does not produce the optimally minimal coloring, but it does have the benefit of running in $O(n)$ time, where n is the number of vertices in the graph.

Colors are represented internally using the Hue-Saturation-Luminosity (HSL) model, which models the space of colors as a cylinder. The cross-section of a cylinder is a circle; hue corresponds to the angle around this circle. Saturation is a number from 0 to 1 and represents a point along the circle's radius, zero being at the center. Luminosity is a number between 0 and 1 and represents a point along the length of the cylinder.

New colors are chosen as follows. The algorithm starts by choosing colors at the starting point of hue = 180°, saturation 1.0, and luminosity 0.5, which is bright blue. Subsequent colors are chosen

with the saturation and luminosity fixed, but with the hue being the value that maximizes the distance on the hue circle between the previous color and any other color. For example, the next color would be HSL(0°, 1.0, 0.5) followed by HSL(90°, 1.0, 0.5). The algorithm is also parameterized by a color restriction so that colors may be excluded for other purposes. For instance, our algorithm currently omits bright red, a color commonly associated with other errors or warnings.

4.2.2 Guided Audit. Another visualization, the *guided audit*, automates some of the human intuition that makes the global view effective. This visualization is a cell-by-cell audit of the highest-ranked proposed fixes generated in the third phase of the excelint' static analysis described in Section 3.4.1. Figure 5b shows a sample fix. The portion in red represents a set of potential errors, and the portion in green represents the set of formulas that ExceLint thinks maintains the correct behavior. This fix a good suggestion, as the formulas in `G6:G8` incorrectly omit data when computing overtime.

While we often found ourselves consulting the global view for additional context with respect to cells flagged by the guided audit, the latter is critical in an important scenario: large spreadsheets that do not fit on-screen. The guided audit solves this scalability issue by highlighting only one suspicious region at a time. The analysis also highlights the region likely to correctly observe the intended reference behavior.

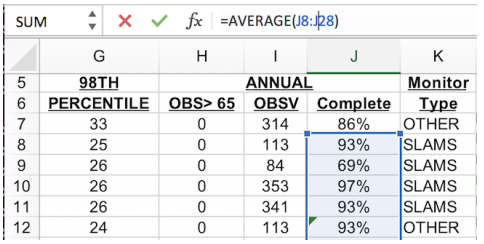
When a user clicks the "Audit" button, the guided audit highlights and centers the user's window on each error, one at a time. To obtain the next error report, the user clicks the "Next Error" button. Errors are visited according to a ranking from most to least likely. Users can stop or restart the analysis from the beginning at any time by clicking the "Start Over" button. If ExceLint's static analysis reports no bugs, the guided audit highlights nothing. Instead, it reports to the user that the analysis found no errors. For performance reasons, ExceLint runs the analysis only once; thus the analysis is not affected by corrections the user makes during an audit.

5 EVALUATION

The evaluation of ExceLint focuses on answering the following research questions. (1) Are spreadsheet layouts really rectangular? (2) How does the proposed fix tool compare against a state-of-the-art pattern-based tool used as an error finder? (3) Is ExceLint fast enough to use in practice? (4) Does it find known errors in a professionally audited spreadsheet?

5.1 Definitions

Formula error. We strictly define a *formula error* as a formula that deviates from the intended reference shape by either *including* an extra reference, *omitting* a reference, or *misreferencing* data. We also include manifestly wrong calculations in this category, such as choosing the wrong operation. A formula error that omits a reference is shown in Figure 6.



	G	H	I	J	K
5	98TH		ANNUAL		Monitor
6	PERCENTILE	OBS> 65	OBSV	Complete	Type
7	33	0	314	86%	OTHER
8	25	0	113	93%	SLAMS
9	26	0	84	69%	SLAMS
10	26	0	353	97%	SLAMS
11	26	0	341	93%	SLAMS
12	24	0	113	93%	OTHER

Fig. 6. **Example error found by ExceLint:** the formula shown (in cell `J30`) has an off-by-one error that omits a row (from `01sumdat.xls`, sheet `PM2.5`).

Inconsistent formula pairs. In our spreadsheet corpus, which we borrowed from the CUSTODES project, incorrect formulas are often found adjacent to correct formulas. This paired structure complicates counting errors because for a given pair, it is often impossible to discern which set is correct and which is incorrect without knowing the user's intent. Despite this, it is usually clear that both sets cannot be correct.

Bug duals. We call these pairs of inconsistent formula sets *bug duals*. Formally, a bug dual is a pair containing two sets of cells, (c_1, c_2). In general, we do not know which set of cells, c_1 or c_2, is correct. We do know, however, that all cells in c_1 induce one fingerprint and that all cells in c_2 induce another.

Without knowing which set in a dual is correct, we cannot count the "true number of errors." We thus arbitrarily label the smaller set of formulas "the error" and the larger set "correct." This is more likely to be a good labeling than the converse because errors are usually rare. Nonetheless, it is occasionally the case that the converse is a better labeling: the user made *systematic errors* and the entire larger region is wrong. For example, an incautious user can introduce large numbers of errors when copying formulas if they fail to update references appropriately.

Our labeling scheme more closely matches the amount of real work that a user must perform when discovering and fixing an inconsistency. In the case where ExceLint mislabels the pair—i.e., the larger region is in error—investigating the discrepancy still reveals the problem. Furthermore, the marginal effort required to fix the larger set of errors versus a smaller set of errors is small. Most of the effort in producing bug fixes is in identifying and understanding an error, not in fixing it, which is often mechanical (e.g., when using tools like Excel's "formula fill"). Counting bug duals by the size of the smaller set is thus a more accurate reflection of the actual effort needed to do an audit.

Counting errors. We therefore count errors as follows. Let a cell flagged by an error-reporting tool be called a *flagged cell*. If a flagged cell is not a formula error, we add nothing to the total error count. If a flagged cell is an error, but has no dual, we add one to the total error count. If a flagged cell is an error and has a bug dual, then we maintain a count of all the cells flagged for the given dual. The maximum number of errors added to the total error count is the number of cells flagged from either set in the dual, or the size of the smaller region, whichever number is *smaller*.

5.2 Evaluation Platform

All evaluation was performed on a developer workstation, a 3.3GHz Intel Core i9-7900X with a 480GB solid state disk, and 64GB of RAM. We used the 64-bit version of Microsoft Windows 10 Pro. We also discuss performance on a more representative platform in §5.6.

5.3 Ground Truth

In order to evaluate whether ExceLint's global view visualization finds new errors, we manually examined 70 publicly-available spreadsheets, using the ExceLint global view to provide context. This manual audit produced a set of annotations which we use for the remainder of the evaluation.

About the benchmarks: We borrow 70 spreadsheet benchmarks developed by researchers (not us) for a comparable spreadsheet tool called CUSTODES [17]. These benchmarks are drawn from the widely-used EUSES corpus [26]. These spreadsheets range in size from 99 cells to 12,121 cells (mean: 1580). The number of formulas for each benchmark ranges from 3 to 2,662 (mean: 360). Most spreadsheets are moderate-to-large in size.

The EUSES corpus collects spreadsheets used as databases, and for financial, grading, homework, inventory, and modeling purposes. EUSES is frequently used by researchers building spreadsheet tools [6, 10, 11, 17, 29, 31–33, 36, 40, 42, 44, 46, 57]. All of the categories present in EUSES are represented in the CUSTODES suite.

Note that during the development of ExceLint, we primarily utilized a small number of synthetic benchmarks generated by us, some spreadsheets from the FUSE corpus [8], and one spreadsheet (`act3_lab23_posey.xls`) from the CUSTODES corpus (because it was small and full of errors).

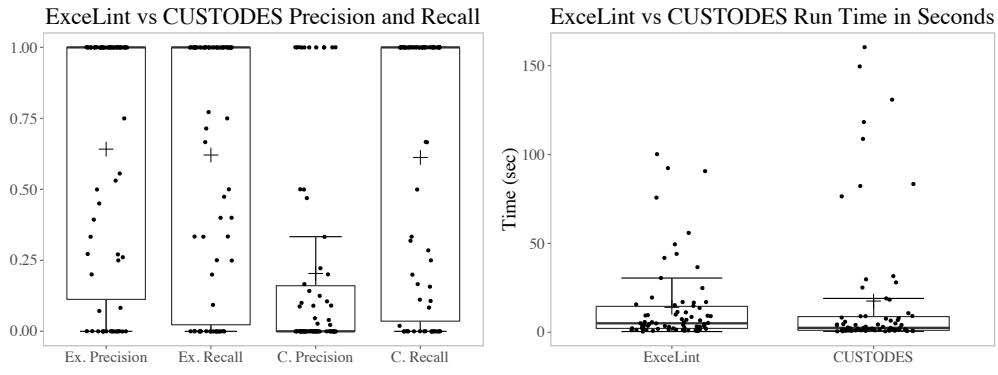


Fig. 7. (a) ExceLint's precision is categorically higher than CUSTODES; recall is comparable. ExceLint's median precision and recall on a workbook are 1. (b) Performance is similar, typically requiring a few seconds. Detailed results are shown in Figures 8, 9, 10, and 11.

Procedure: We re-annotated all of the 70 spreadsheets provided by the CUSTODES research group. Re-auditing the same corpus with a different tool helps establish whether ExceLint helps uncover errors that CUSTODES does not find; it also draws a distinction between smells and real formula errors. The annotation procedure is as follows. Each spreadsheet was opened and the ExceLint global view was displayed. Visual irregularities were inspected either by clicking on the formula and examining its references, or by using Excel's formula view. If the cell was found to be a formula error, it was labeled as an error. In cases where an error had a clear bug dual (see "Bug duals" in §5.1), both sets of cells were labeled, and a note was added that the two were duals. We then inspected the CUSTODES ground truth annotations for the same spreadsheet, following the same procedure as before, with one exception: if it was clear that a labeled smell was not a formula error, it was labeled "not a bug."

Results: For the 70 spreadsheets we annotated, the CUSTODES ground truth file indicates that 1,199 cells are smells. Our audit shows that, among the flagged cells, CUSTODES finds 102 formula errors that also happen to be smells. During our re-annotation, we found an additional 295 formula errors, for a total of 397 formula errors. We spent a substantial amount of time manually auditing these spreadsheets (roughly 34 hours, cumulatively). Since we did not perform an unassisted audit for comparison, we do not know how much time we saved versus not using the tool. Nonetheless, since an unassisted audit would require examining *all* of the cells individually, the savings are likely substantial. On average, we uncovered one formula error per 5.1 minutes, which is clearly an effective use of auditor effort.

Our methodology reports a largely distinct set of errors from the CUSTODES work. This is in part because we distinguish between suspicious cells and cells that are unambiguously wrong (see "Formula error" in §5.1). In fact, during our annotation, we also observed a large number (9,924) of unusual constructions (missing formulas, operations on non-numeric data, and suspicious calculations), many of which are labeled by CUSTODES as "smells." For example, a sum in a financial spreadsheet with an apparent "fudge factor" of `+1000` appended is highly suspect but not unambiguously wrong without being able to consult with the spreadsheet's original author. We do not report these as erroneous.

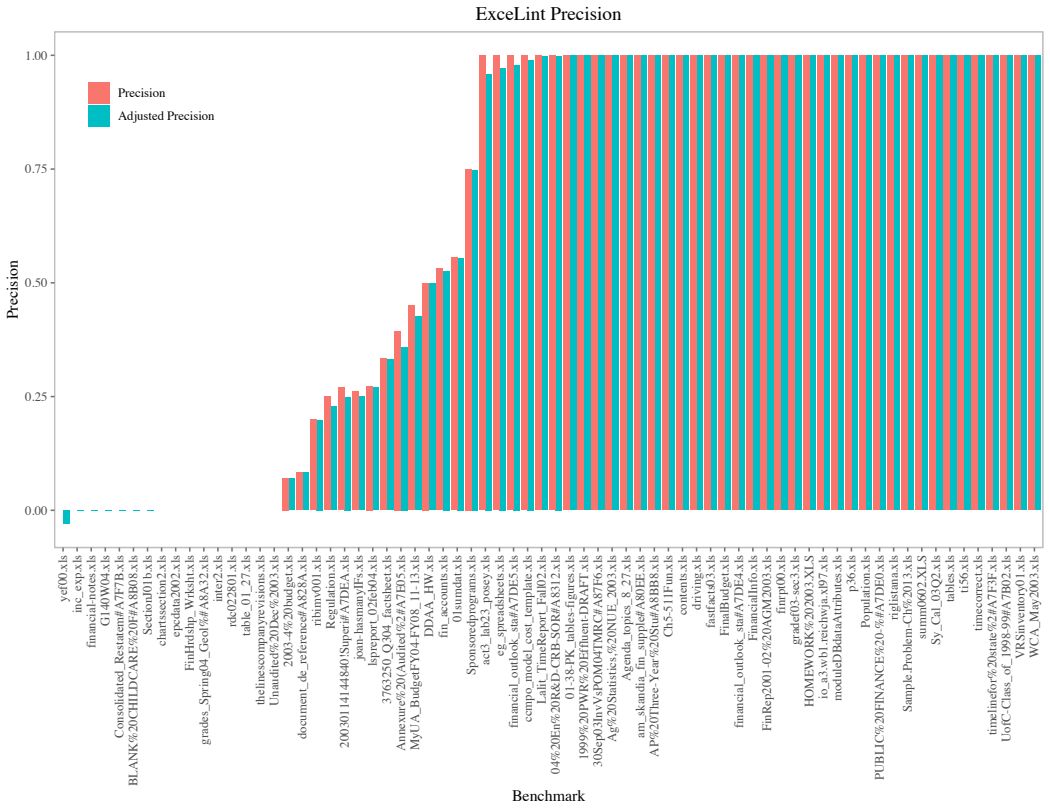


Fig. 8. ExceLint's precision is generally high across the CUSTODES benchmark suite. In most cases, adjusting the precision based on the expected number of cells flagged by a random flagger has little effect. Results are sorted by adjusted precision (see Section 5.5).

5.4 RQ1: Are Layouts Rectangular?

Since ExceLint relies strongly on the assumption that users produce spreadsheets in a rectangular fashion, it is reasonable to ask whether layouts really are rectangular. Across our benchmark suite, we found that on average 62.3% (median: 63.2%) of all fingerprint regions containing data or formulas (excluding whitespace) in a workbook are rectangular. When looking specifically at formulas, on average 86.8% (median: 90.5%) of formula regions in a workbook are also rectangular. This analysis provides strong evidence that users favor rectangular layouts, especially when formulas are involved.

5.5 RQ2: Does ExceLint Find Formula Errors?

In this section, we evaluate ExceLint's guided audit. Since it is standard practice to compare against the state-of-the-art, we also compare against an alternative tool, CUSTODES [17]. CUSTODES utilizes a pattern-based approach to find "smelly" formulas. While smells are sometimes errors, CUSTODES is not an error finder. We argue neither for nor against the merits of finding smells, which could be useful for making spreadsheets more maintainable. Nonetheless, our goal is to discover *errors*. To our knowledge, CUSTODES is both the best automated error finder available to the general public.

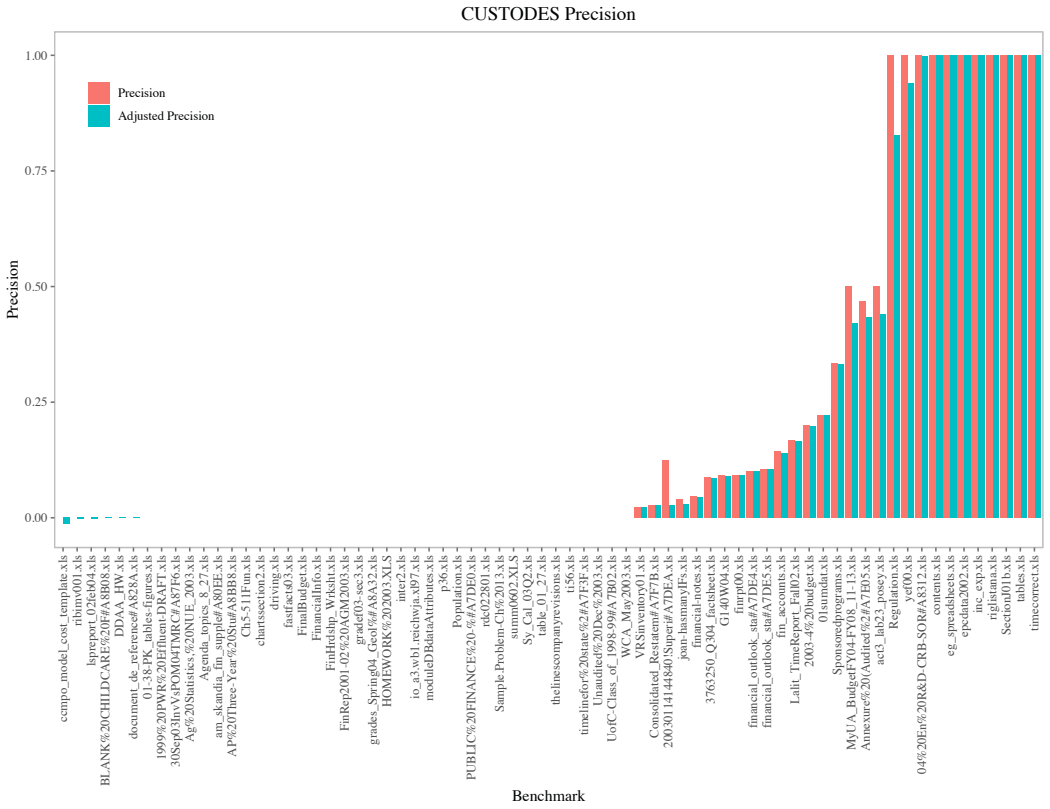


Fig. 9. CUSTODES precision is generally lower than ExceLint's. See Figure 8. Results are sorted by adjusted precision (see Section 5.5).

Procedure: A highlighted cell uncovers a real error if either (1) the flagged cell is labeled as an error in our annotated corpus or (2) it is labeled as a bug dual. We count the number of true positives using the procedure described earlier (see "Counting errors"). We use the same procedure when evaluating CUSTODES.

Definitions: Precision is defined as $TP/(TP + FP)$ where TP denotes the number of true positives and FP denotes the number of false positives. When a tool flags nothing, we define precision to be 1, since the tool makes no mistakes. When a benchmark contains no errors but the tool flags anything, we define precision to be 0 since nothing that it flags can be a real error. Recall is defined as $TP/(TP + FN)$ where FN is the number of false negatives.

Difficulty of accurately computing recall. It is conventional to report precision along with recall. Nonetheless, recall is inherently difficult to measure when using real-world spreadsheets as benchmarks. To accurately compute recall, the true number of false negatives must be known. A false negative in our context records when a tool incorrectly flags a cell as not containing an error when it actually does. The true number of errors in a spreadsheet is difficult to ascertain without a costly audit by domain experts. We compute recall using the false negative count obtained from our assisted audit; given the large number of suspicious cells we identified, we believe that a domain expert would likely classify more formulas as containing formula errors. Since we adopt

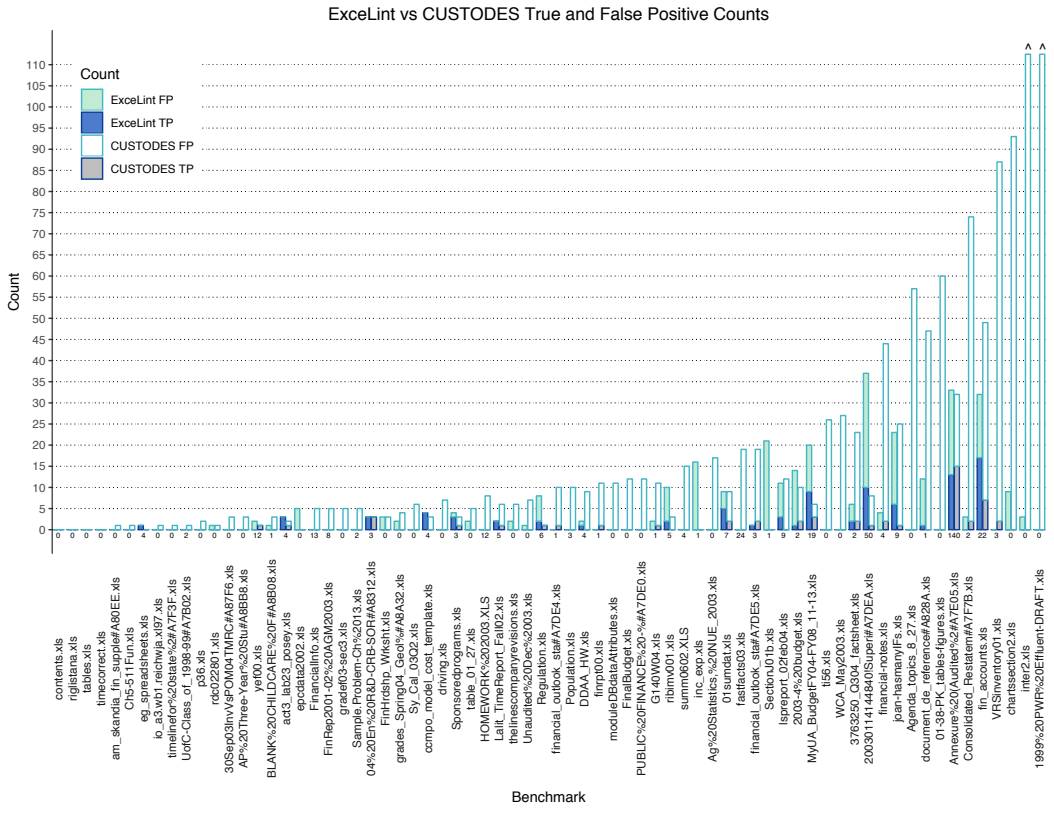


Fig. 10. ExceLint produces far fewer false positives than CUSTODES. Critically, it flags nothing when no errors are present. Each benchmark shows two stacked bars, with ExceLint on the left and CUSTODES on the right. Numbers below bars denote the ground truth number of errors. Bars are truncated at two standard deviations; ^ indicates that the bar was truncated (409 and 512 false positives, respectively).

a conservative definition for error, it is likely that the real recall figures are *lower* than what we report.

Results: Across all workbooks, ExceLint has a mean precision of 64.1% (median: 100%) and a mean recall of 62.1% (median: 100%) when finding formula errors. Note that we strongly favor high precision over high recall, based on the observation that users find low-precision tools to be untrustworthy [13].

In general, ExceLint outperforms CUSTODES. We use CUSTODES' default configuration. CUSTODES's mean precision on a workbook is 20.3% (median: 0.0%) and mean recall on a workbook is 61.2% (median: 100.0%). Figure 7a compares precision and recall for ExceLint and CUSTODES. Note that both tools are strongly affected by a small number of benchmarks that produce a large number of false positives. For both tools, only 5 benchmarks account for a large fraction of the total number of false positives, which is why we also report median values which are less affected by outliers. ExceLint's five worst benchmarks are responsible for 45.2% of the error; CUSTODES five worst are responsible for 64.3% of the error. Figure 10 shows raw true and false positive counts.

CUSTODES explicitly sacrifices precision for recall; we believe that this is the wrong tradeoff. The total number of false positives produced by each tool is illustrative. Across the entire suite,

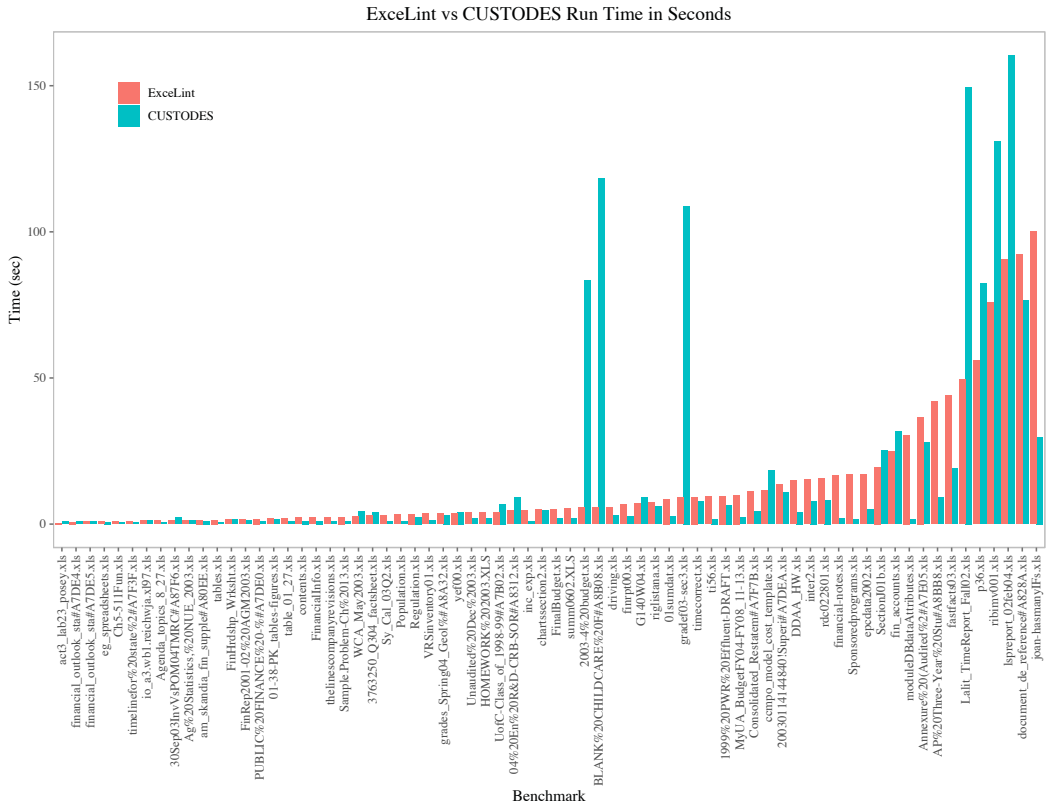


Fig. 11. ExceLint and CUSTODES have similar performance, typically requiring a few seconds to run an analysis.

ExceLint produced 89 true positives and 223 false positives; CUSTODES produced 52 true positives and 1,816 false positives. Since false positive counts are a proxy for wasted user effort, CUSTODES wastes roughly 8× more user effort than ExceLint.

Random baseline: Because it could be the case that ExceLint's high precision is the result of a large number of errors in a spreadsheet (i.e., flagging nearly anything is likely to produce a true positive, thus errors are easy to find), we also evaluate ExceLint and CUSTODES against a more aggressive baseline. The baseline is the expected precision obtained by randomly flagging cells.

We compute random baselines analytically. For small spreadsheets, sampling with replacement may produce a very different distribution than sampling without replacement. A random flagger samples without replacement; even a bad tool should not flag the same cell twice. Therefore, we compute the expected value using the hypergeometric distribution, which corrects for small sample size. Expected value is defined as $\mathbb{E}[X] = n\frac{r}{m}$ where X is a random variable representing the number of true positives, m is the total number of cells, r is the number of true reference errors in the workbook according to the ground truth, and n is the size of the sample, i.e., the number of errors requested by the user. For each tool, we fix n to be the same number of cells flagged by the tool.

We define TP_a, the adjusted number of true positives, to be $TP - \mathbb{E}[X]$. Correspondingly, we define the adjusted precision to be 1 when the tool correctly flags nothing (i.e., there are no bugs present), and $\frac{TP_a}{TP_a+FP_a}$ otherwise.

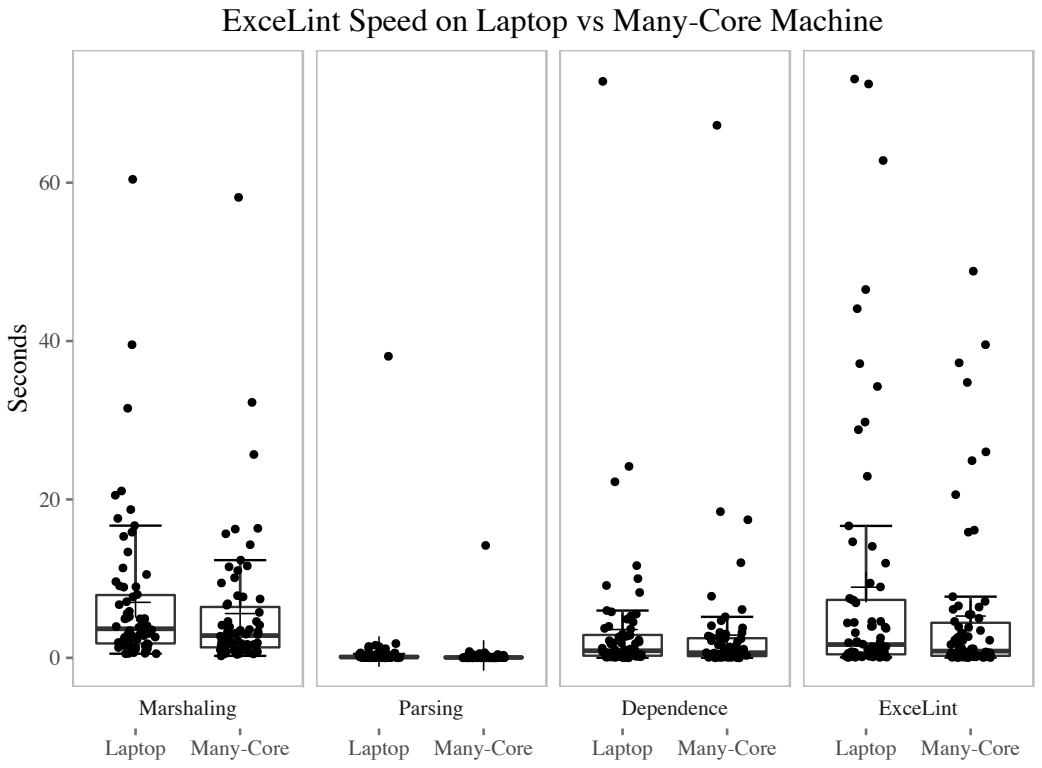


Fig. 12. Performance measurements for each phase of ExceLint's analysis across the benchmark suite using two configurations: a laptop and a multicore machine. On the multicode machine, ExceLint sees the greatest speedup in the entropy decomposition phase of the ExceLint analysis, which is multithreaded.

Results. ExceLint's mean adjusted precision is 63.7% and CUSTODES's mean adjusted precision is 19.6%. In general, this suggests that neither tool's precision is strongly dictated by a poor selection of benchmarks. The random flagger performs (marginally) better than ExceLint in only 7 of the 70 cases. The random flagger outperformed CUSTODES in 14 cases.

High precision when doing nothing. Given that a tool that flags nothing obtains high precision (but very low recall), it is fair to ask whether ExceLint boosts its precision by doing nothing. Indeed, ExceLint flags no cells in 33 of 70 cases. However, two facts dispel this concern. First, ExceLint has a high formula error recall of 62.1%. Second, in 22 cases where ExceLint does nothing, the benchmark contains no formula error. Thus, ExceLint saves users auditing effort largely when it should save them effort.

CUSTODES' low precision. Our results for CUSTODES are markedly different than those reported by the authors for the same suite (precision: 0.62). There are a number of reasons for the difference. First, the goal of this work is to precisely locate unambiguous errors. CUSTODES instead looks for suspect patterns. Unambiguous errors and suspect patterns are quite different; in our experience, suspect patterns rarely correspond to unambiguous errors. Second, our experience using the CUSTODES tool leads us to believe that it is primarily tuned to find missing formulas. While missing formulas (i.e., a hand-calculated constant instead of a formula) are indeed common, we found that they rarely

resulted in performing the wrong calculation. Finally, because of the presence of bug duals, we report fewer bugs than the CUSTODES group (which reports smells), because our count is intended to capture debugging effort.

5.6 RQ3: Is ExceLint Fast?

ExceLint analyzes spreadsheets quickly. ExceLint's mean runtime is 14.0 seconds, but the mean is strongly affected by the presence of four long running outliers. ExceLint's median runtime across all workbooks is 5 seconds. 85.2% of the analyses run under 30 seconds. Figure 7b compares ExceLint's and CUSTODES' runtimes.

Two factors have a big impact on ExceLint's performance: (1) complex dependence graphs, and (2) dense spreadsheets with little whitespace. The former is a factor because a reference vector must be produced for every reference in a workbook (§3.2). The latter is a factor because the entropy calculation cannot be effectively parallelized when the user creates "dense" layouts (§4.1.2). The longest-running spreadsheets took a long time to produce both reference vectors and to compute entropy, suggesting that they had complex formulas and dense layouts.

Figure 12 compares ExceLint's performance on two different hardware configurations: (1) running in VMWare on a typical laptop (a 2016 Macbook Pro) and (2) the multicore desktop machine used for the rest of these benchmarks. The two configurations exhibit an interesting feature: the median runtime on the laptop is the same as on the high-performance machine. The reason is that small benchmarks run quickly on both machines, since these benchmarks are dominated by single-thread performance. Larger benchmarks, on the other hand, benefit from the additional cores available on the high-performance machine. This difference is most noticeable during the entropy decomposition phase of ExceLint's analysis, which is written to take advantage of available hardware parallelism. As computing hardware is expected to primarily gain cores but not higher clock rates, ExceLint's analysis is well-positioned to take advantage of future hardware.

5.7 RQ4: Case Study: Does ExceLint Reproduce Findings of a Professional Audit?

In 2010, the economists Carmen Reinhart and Kenneth Rogoff published the paper "Growth in a Time of Debt" (GTD) that argued that after exceeding a critical debt-to-GDP ratio, countries are doomed to an economic "death spiral". The paper was highly influential among conservatives seeking to justify austerity measures. However, it was later discovered by Herndon et al. that GTD's deep flaws meant that the opposite conclusion was true. Notably, Reinhart and Rogoff utilized a spreadsheet to perform their analysis.

Herndon et al. call out one class of spreadsheet error as particularly significant [37]. In essence, the computation completely excludes five countries—Denmark, Canada, Belgium, Austria, and Australia—from the analysis.

We ran ExceLint on the Reinhart-Rogoff spreadsheet and it found this error, repeated twice. Both error reports were also found by professional auditors. Figure 13 shows one of the sets of errors. The cells in red, `E26:H26`, are inconsistent with the set of cells in green, `I26:X26`. In fact, `I26:X26` is wrong and `E26:H26` is correct, because `I26:X26` fails to refer to the entire set of figures for each country, the cells in rows 5 through 24. Nonetheless, by highlighting both regions, it is immediately clear which of the two sets of cells is wrong.

5.8 Summary

Using the ExceLint global view visualization, we uncovered 295 more errors than CUSTODES, for a total of 397 errors, when used on an existing pre-audited corpus. When using ExceLint to propose fixes, ExceLint is 44.9 percentage points more precise than the comparable state-of-the-art

	B	C	D	E	F	G	H	I	J	K	L
1	: December 5, 2009			Number of observations							Ave
2								Real GDP growth			
3				Debt/GDP				Debt/GDP			
4	Country	Coverage	Total	30 or less	30 to 60	60 to 90	90 or above	30 or less	30 to 60	60 to 90	90 or above
5	US	1791-2009		129	59	23		4.0	3.4	3.3	-1.8
6	UK	1830-2009		3	68	27	8	2.5	2.2	2.1	1.8
7	Sweden	1880-2009		79	40	11		2.9	2.9	2.7	n.a.
8	Spain	1850-2009		26	53	47	3	1.6	3.2	1.3	2.8
9	Portugal	1880-2009		42	10	39		4.8	2.5	1.4	n.a.
10	Norway	1880-2009		98	25	1		2.9	4.4	10.2	n.a.
11	New Zealand	1932-2009		9	33	17	1	2.5	2.9	3.9	3.6
12	Netherlands	1880-2009		17	50	32		4.1	2.8	2.4	2.0
13	Japan	1885-2009		47	42	11	1	4.9	3.7	3.9	0.7
14	Italy	1880-2009		26	12	39	4	5.4	4.9	1.9	0.7
15	Ireland	1949-2009		8	14	32		4.4	4.5	4.0	2.4
16	Greece	1884-2009		13	5	11	5	4.0	0.3	4.8	2.5
17	Germany	1880-2009		96	11	0		3.6	0.9	n.a.	n.a.
18	France	1880-2009		26	21	19	3	4.9	2.7	2.8	2.3
19	Finland	1914-2009		69	18	6		3.2	3.0	4.3	1.9
20	Denmark	1880-2009		57	16	17	0	3.1	1.7	2.4	n.a.
21	Canada	1925-2009		3	52	23	7	1.9	4.5	3.0	2.2
22	Belgium	1835-2009		37	60	32	31	3.0	2.6	2.1	3.3
23	Austria	1880-2009		43	32	35	0	4.3	3.0	2.3	n.a.
24	Australia	1902-2009		38	33	23	8	3.1	4.1	2.3	4.6
25											
26			2317	866	654	445	352	3.7	3.0	3.5	1.7

Fig. 13. ExceLint flags the formulas in E26:H26, which are inconsistent with formulas in I26:X26. The red and green boxes show the set of cells referenced by E26 and I26, respectively. While ExceLint marks E26:H26 as the "error" because it is the smaller set, in fact, E26:H26 are correct. This spreadsheet contains a systematic error and all formulas in I26:X26 are incorrect.

smell-based tool, CUSTODES. Finally, ExceLint is fast enough to run interactively, requiring a median of 5 seconds to run a complete analysis on an entire workbook.

6 RELATED WORK

Smells: One technique for spreadsheet error detection employs *ad hoc* pattern-based approaches. These approaches, sometimes called "smells", include patterns like "long formulas" that are thought to reflect bad practices [34]. Excel itself includes a small set of such patterns. Much like source code "linters," flagged items are not necessarily errors. While we compare directly against only one such tool—CUSTODES—other recent work from the software engineering community adopts similar approaches [17, 21, 35, 38, 41]. For example, we found another smell-based tool, FaultySheet Detective, to be unusably imprecise [3]. When run on the "standard solution" provided with its own benchmark corpus—an error-free spreadsheet—FaultySheet Detective flags 15 of the 19 formulas present, a 100% false positive rate. Both ExceLint and CUSTODES correctly flag nothing.

Type and Unit Checking: Other work on detecting errors in spreadsheets has focused on inferring units and relationships (*has-a*, *is-a*) from information like structural clues and column headers, and then checking for inconsistencies [1, 5, 7, 15, 23–25, 43]. These analyses do find real bugs in spreadsheets, but they are largely orthogonal to our approach. Many of the bugs that ExceLint finds would be considered type- and unit-safe.

Fault Localization and Testing: Relying on an error oracle, *fault localization* (also known as *root cause analysis*) traces a problematic execution back to its origin. In the context of spreadsheets, this means finding the source of an error given a manually-produced annotation, test, or specification [4, 39, 40]. Note that many localization approaches for spreadsheets are evaluated using randomly-generated errors, which are now known not generalize to real errors [49]. Localization aside, there has also been considerable work on testing tools for spreadsheets [2, 14, 27, 43, 53–55]

ExceLint is a fully-automated error finder for spreadsheets and needs no specifications, annotations, or tests of any kind. ExceLint is also evaluated using real-world spreadsheets, not randomly-generated errors.

Anomaly Detection: An alternative approach frames errors in terms of anomalies. Anomaly analysis leverages the observation from conventional programming languages that anomalous code is often wrong [18, 20, 22, 30, 51, 60]. This lets an analysis circumvent the difficulty of obtaining program correctness rules.

ExceLint bears a superficial resemblance to anomaly analysis. Both are concerned with finding unusual program fragments. However, ExceLint's approach is not purely statistical; rather, the likelihood of errors is based on the effect a fix has on the entropy of the layout of references.

7 CONCLUSION

This paper presents ExceLint, an information-theoretic static analysis that finds formula errors in spreadsheets. We show that ExceLint has high precision and recall (median: 100%). We have released ExceLint as an open source project [9] that operates as a plugin for Microsoft Excel.

8 ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1617892.

REFERENCES

[1] Robin Abraham and Martin Erwig. 2004. Header and unit inference for spreadsheets through spatial analyses. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 165–172.

[2] Rui Abreu, Simon Außerlechner, Birgit Hofer, and Franz Wotawa. 2015. Testing for Distinguishing Repair Candidates in Spreadsheets - the Mussco Approach. In *Testing Software and Systems - 27th IFIP WG 6.1 International Conference, ICTSS 2015, Sharjah and Dubai, United Arab Emirates, November 23-25, 2015, Proceedings*. 124–140. https://doi.org/10.1007/978-3-319-25945-1$_8$

[3] R. Abreu, J. Cunha, J. P. Fernandes, P. Martins, A. Perez, and J. Saraiva. 2014. Smelling Faults in Spreadsheets. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 111–120. https://doi.org/10.1109/ICSME.2014.33

[4] Rui Abreu, Birgit Hofer, Alexandre Perez, and Franz Wotawa. 2015. Using constraints to diagnose faulty spreadsheets. *Software Quality Journal* 23, 2 (2015), 297–322. https://doi.org/10.1007/s11219-014-9236-4

[5] Yanif Ahmad, Tudor Antoniu, Sharon Goldwater, and Shriram Krishnamurthi. 2003. A Type System for Statically Detecting Spreadsheet Errors. In *ASE*. IEEE Computer Society, 174–183.

[6] Abdussalam Alawini, David Maier, Kristin Tufte, Bill Howe, and Rashmi Nandikur. 2015. Towards Automated Prediction of Relationships Among Scientific Datasets. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management (SSDBM '15)*. ACM, New York, NY, USA, Article 35, 5 pages. https://doi.org/10.1145/2791347.2791358

[7] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. 2004. Validating the Unit Correctness of Spreadsheet Programs. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 439–448. http://dl.acm.org/citation.cfm?id=998675.999448

[8] Titus Barik, Kevin Lubick, Justin Smith, John Slankas, and Emerson R. Murphy-Hill. 2015. Fuse: A Reproducible, Extendable, Internet-Scale Corpus of Spreadsheets. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. 486–489. https://doi.org/10.1109/MSR.2015.70

[9] Daniel W. Barowy, Emery D. Berger, and Benjamin Zorn. 2018. ExceLint repository. https://github.com/excelint/excelint. (2018).

[10] Daniel W. Barowy, Dimitar Gochev, and Emery D. Berger. 2014. CheckCell: Data Debugging for Spreadsheets. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 507–523. https://doi.org/10.1145/2660193.2660207

[11] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 218–228. https://doi.org/10.1145/2737924.2737952

[12] Michael Batty. 1974. Spatial Entropy. *Geographical Analysis* 6, 1 (1974), 1–31. https://doi.org/10.1111/j.1538-4632.1974.tb01014.x

[13] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[14] Jeffrey Carver, Marc Fisher, II, and Gregg Rothermel. 2006. An empirical evaluation of a testing and debugging methodology for Excel. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering (ISESE '06)*. ACM, New York, NY, USA, 278–287. https://doi.org/10.1145/1159733.1159775

[15] Chris Chambers and Martin Erwig. 2010. Reasoning about spreadsheets with labels and dimensions. *J. Vis. Lang. Comput.* 21, 5 (Dec. 2010), 249–262. https://doi.org/10.1016/j.jvlc.2010.08.004

[16] J.P. Morgan Chase and Co. 2013. Report of JPMorgan Chase and Co. Management Task Force Regarding 2012 CIO Losses. (16 Jan. 2013). http://files.shareholder.com/downloads/ONE/5509659956x0x628656/4cb574a0-0bf5-4728-9582-625e4519b5ab/TaskForceReport.pdf

[17] Shing-Chi Cheung, Wanjun Chen, Yepang Liu, and Chang Xu. 2016. CUSTODES: Automatic Spreadsheet Cell Clustering and Smell Detection using Strong and Weak Features. In *Proceedings of ICSE '16*. to appear.

[18] Trishul M. Chilimbi and Vinod Ganapathy. 2006. HeapMD: Identifying Heap-based Bugs Using Anomaly Detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 219–228. https://doi.org/10.1145/1168857.1168885

[19] Keith D. Cooper and Linda Torczon. 2005. *Engineering a Compiler*. Morgan Kaufmann.

[20] Martin Dimitrov and Huiyang Zhou. 2009. Anomaly-based Bug Prediction, Isolation, and Validation: An Automated Approach for Software Debugging. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/1508244.1508252

[21] Wensheng Dou, Shing-Chi Cheung, and Jun Wei. 2014. Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 848–858.

[22] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041

[23] Martin Erwig. 2009. Software Engineering for Spreadsheets. *IEEE Softw.* 26, 5 (Sept. 2009), 25–30. https://doi.org/10.1109/MS.2009.140

[24] Martin Erwig, Robin Abraham, Irene Cooperstein, and Steve Kollmansberger. 2005. Automatic generation and maintenance of correct spreadsheets. In *ICSE (ICSE '05)*. ACM, New York, NY, USA, 136–145. https://doi.org/10.1145/1062455.1062494

[25] Martin Erwig and Margaret Burnett. 2002. Adding apples and oranges. In *Practical Aspects of Declarative Languages*. Springer, 173–191.

[26] Marc Fisher and Gregg Rothermel. 2005. The EUSES spreadsheet corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanisms. *SIGSOFT Softw. Eng. Notes* (July 2005).

[27] M. Fisher, G. Rothermel, T. Creelan, and M. Burnett. 2006. Scaling a Dataflow Testing Methodology to the Multiparadigm World of Commercial Spreadsheets. In *17th International Symposium on Software Reliability Engineering (ISSRE'06)*. IEEE, 13–22.

[28] Mary Jo Foley. 2010. About that 1 billion Microsoft Office figure ... http://www.zdnet.com/article/about-that-1-billion-microsoft-office-figure. (16 June 2010).

[29] Valentina I. Grigoreanu, Margaret M. Burnett, and George G. Robertson. 2010. A Strategy-centric Approach to the Design of End-user Debugging Tools. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 713–722. https://doi.org/10.1145/1753326.1753431

[30] Sudheendra Hangal and Monica S. Lam. 2002. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 291–301. https://doi.org/10.1145/581339.581377

[31] Felienne Hermans and Danny Dig. 2014. BumbleBee: A Refactoring Environment for Spreadsheet Formulas. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 747–750. https://doi.org/10.1145/2635868.2661673

[32] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2012. Detecting and Visualizing Inter-worksheet Smells in Spreadsheets. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 441–451. http://dl.acm.org/citation.cfm?id=2337223.2337275

[33] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2010. Automatically Extracting Class Diagrams from Spreadsheets. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 52–75. http://dl.acm.org/citation.cfm?id=1883978.1883984

[34] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2012. Detecting code smells in spreadsheet formulas. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 409–418.

[35] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2015. Detecting and refactoring code smells in spreadsheet formulas. *Empirical Software Engineering* 20, 2 (01 Apr 2015), 549–575. https://doi.org/10.1007/s10664-013-9296-2

[36] Felienne Hermans, Ben Sedee, Martin Pinzger, and Arie van Deursen. 2013. Data Clone Detection and Visualization in Spreadsheets. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 292–301. http://dl.acm.org/citation.cfm?id=2486788.2486827

[37] Thomas Herndon, Michael Ash, and Robert Pollin. 2013. *Does High Public Debt Consistently Stifle Economic Growth? A Critique of Reinhart and Rogoff.* Working Paper Series 322. Political Economy Research Institute, University of Massachusetts Amherst. http://www.peri.umass.edu/fileadmin/pdf/working$_{p}$apers/working$_{p}$apers$_{3}$01-350/WP322.pdf

[38] Birgit Hofer, Andrea Hofler, and Franz Wotawa. 2017. Combining Models for Improved Fault Localization in Spreadsheets. *IEEE Trans. Reliability* 66, 1 (2017), 38–53. https://doi.org/10.1109/TR.2016.2632151

[39] Birgit Hofer, Alexandre Perez, Rui Abreu, and Franz Wotawa. 2015. On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Autom. Softw. Eng.* 22, 1 (2015), 47–74. https://doi.org/10.1007/s10515-014-0145-3

[40] Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner. 2013. On the empirical evaluation of fault localization techniques for spreadsheets. In *Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 68–82. https://doi.org/10.1007/978-3-642-37057-1$_{6}$

[41] Dietmar Jannach, Thomas Schmitz, Birgit Hofer, and Franz Wotawa. 2014. Avoiding, finding and fixing spreadsheet errors - A survey of automated approaches for spreadsheet QA. *Journal of Systems and Software* 94 (2014), 129–150. https://doi.org/10.1016/j.jss.2014.03.058

[42] Nima Joharizadeh. 2015. Finding Bugs in Spreadsheets Using Reference Counting. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH Companion 2015)*. ACM, New York, NY, USA, 73–74. https://doi.org/10.1145/2814189.2815373

[43] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[44] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

[45] Gaspard Monge. 1781. Mémoire sur la théorie des déblais et des remblais. *Histoire de l'Académie Royale des Sciences* (1781), 666–704.

[46] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2015. Preventing Data Errors with Continuous Testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/2771783.2771792

[47] Ray Panko. 2015. What We Don't Know About Spreadsheet Errors Today: The Facts, Why We Don't Believe Them, and What We Need to Do. In *The European Spreadsheet Risks Interest Group 16th Annual Conference (EuSpRiG 2015)*. EuSpRiG.

[48] Raymond R. Panko. 1998. What we know about spreadsheet errors. *Journal of End User Computing* 10 (1998), 15–21.

[49] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 609–620. https://doi.org/10.1109/ICSE.2017.62

[50] J. R. Quinlan. 1986. Induction of Decision Trees. *MACH. LEARN* 1 (1986), 81–106.

[51] Orna Raz, Philip Koopman, and Mary Shaw. 2002. Semantic anomaly detection in online data sources. In *ICSE (ICSE '02)*. ACM, New York, NY, USA, 302–312. https://doi.org/10.1145/581339.581378

[52] Carmen M. Reinhart and Kenneth S. Rogoff. 2010. *Growth in a Time of Debt.* Working Paper 15639. National Bureau of Economic Research. http://www.nber.org/papers/w15639

[53] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. 2001. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 10, 1 (2001), 110–147.

[54] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. 1998. What you see is what you test: A methodology for testing form-based visual programs. In *ICSE 1998*. IEEE, 198–207.

[55] Thomas Schmitz, Dietmar Jannach, Birgit Hofer, Patrick W. Koch, Konstantin Schekotihin, and Franz Wotawa. 2017. A decomposition-based approach to spreadsheet testing and debugging. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017*. 117–121. https://doi.org/10.1109/

VLHCC.2017.8103458
[56] C. E. Shannon. 1948. A mathematical theory of communication. *Bell system technical journal* 27 (1948).
[57] Rishabh Singh, Benjamin Livshits, and Ben Zorn. 2017. *Melford: Using Neural Networks to Find Spreadsheet Errors.* Technical Report. https://www.microsoft.com/en-us/research/publication/melford-using-neural-networks-find-spreadsheet-errors/
[58] Peter Wegner. 1960. A Technique for Counting Ones in a Binary Computer. *Commun. ACM* 3, 5 (May 1960), 322–. https://doi.org/10.1145/367236.367286
[59] D. J. A. Welsh and M. B. Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Comput. J.* 10, 1 (1967), 85–86. https://doi.org/10.1093/comjnl/10.1.85
[60] Yichen Xie and Dawson Engler. 2002. Using Redundancies to Find Errors. In *IEEE Transactions on Software Engineering*. 51–60.