Referee: A Pattern-Guided Approach for Auto Design in Compiler-Based Analyzers

Fang Lv*, Hao Li*, Lei Wang*, Ying Liu*, Huimin Cui*, Jingling Xue†, and Xiaobing Feng*
*SKL Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China
†School of Computer Science and Engineering,University of New South Wales, Australia
flv@ict.ac.cn, lihao2018@ict.ac.cn, wlei@ict.ac.cn
liuying2007@ict.ac.cn, huimin.cui@gmail.com, jingling@cse.unsw.edu.au, fxb@ict.ac.cn

***Abstract*—Coding rules become more critical for security-oriented softwares, which prefer compilers as their base platforms due to simultaneous demands not only in a mature grammatical analysis, but also in compilation and optimization techniques. However, engineering such a compiler-based analyzer, exploring proper launch points before integrating hundreds of rules one by one in the frontend of compilers, is a completely manual decision-making process with heavy redundant efforts exhausted. To improve this, we introduce a novel pattern-guided approach, named Referee, to facilitate this process. Referee improves the manual approach significantly by making three advances: (1) our pattern-guided approach can significantly reduce the amount of redundant manual efforts required, (2) a twin-graph aided broadcasting process is developed to enable rule patterns to be characterized with partially developed rules, and (3) a reliable recommendation mechanism is used to pinpoint the launch point for a new rule based on the accumulated experience from handling earlier rules.**

We have implemented Referee in GCC 8.2 with 163 rules from SPACE-C and MISRA-C standards. Referee achieves an accuracy of 89.9% on recommendation of launch points for new rules to our GCC-based analyzer automatically when trained with 70% of all the rules. Decreasing the training data size to 60% and 50% still yields an accuracy of 87.7% and 81.5%, respectively. Therefore, Referee can significantly reduce the amount of manual efforts that would otherwise be required, with a careful selection of seeding rule patterns, providing an interesting and fruitful avenue for further research.

***Index Terms*—pattern-based, machine-learning, coding rules, analyzer, compiler**

I. Introduction

Coding rules define a consistent syntactic style, fostering not only readability, but also safety and maintainability [1]–[6]. They are becoming increasingly critical especially for security-oriented softwares. Many coding rules prefer to be constructed in compilers due to their higher demands in the maturity of grammatical analysis [4]. Therefore, a compiler-based analyzer is capable of carrying out safety inspection while compiling and optimizing simultaneously. However, engineering such a compiler-based analyzer, exploring the complicated compiler architectures and performing enhancement, is a time-consuming process which urges for innovation. The existing manual-mode development spends considerable time exploring proper launch points (the calling sites of inspection modules for rules) before integrating hundreds of rules one by one in the frontend of compilers. More importantly, the rules keep on updating, e.g., 169 rules in 2008 and additional 159 rules in 2012 are supplemented in MISRA-C [5]. This becomes an essential, and heavily burdened work which hurdles the widespreading of the techniques deeply, but few mitigation exists.

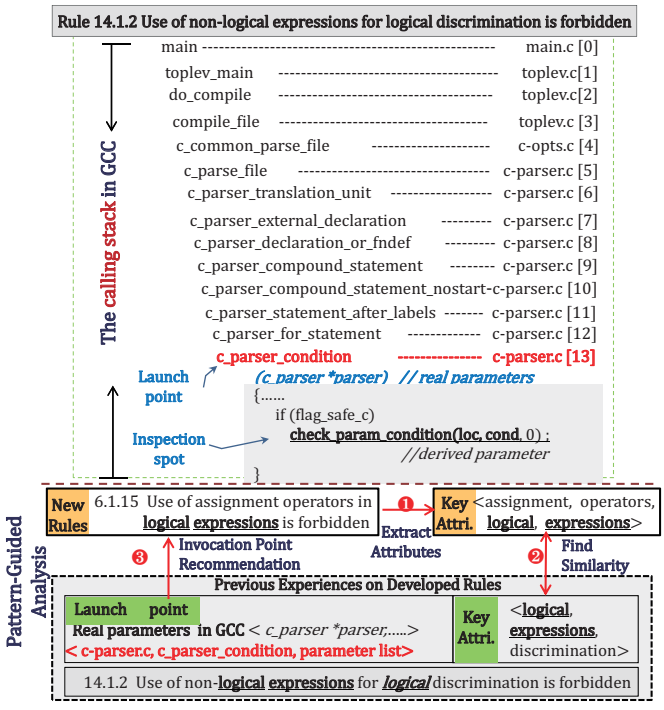

Fig. 1. Deep calling stack in the GCC-base analyzer and our pattern-guided design approach. Redundant manual efforts on locating launch points can be saved via exploiting previous experiences on similar-looking rules.

Fig. 1 demonstrates this tediousness as to figuring out the proper calling site to implement a rule's inspection, e.g., searching in the 14-layer deep calling stacks for Rule 14.1.2, compared to usually only a few dozens of lines in its inspection codes. Our experiences in the GCC-based analyzer for SPACE-C [6] show that it takes longer than 6 months to complete the whole code design for 163 rules of SPACE-C. During this process, some efforts have to be consumed on pinpointing the

This work is supported in part by the National Key R&D Program of China (2017YFB0202901), CCF-Tencent Open Reseacrch Fund grant, and the National Natural Science Foundation of China (61802368, 61521092, 61432016, 61432018, 61332009, 61702485, and 61872043).

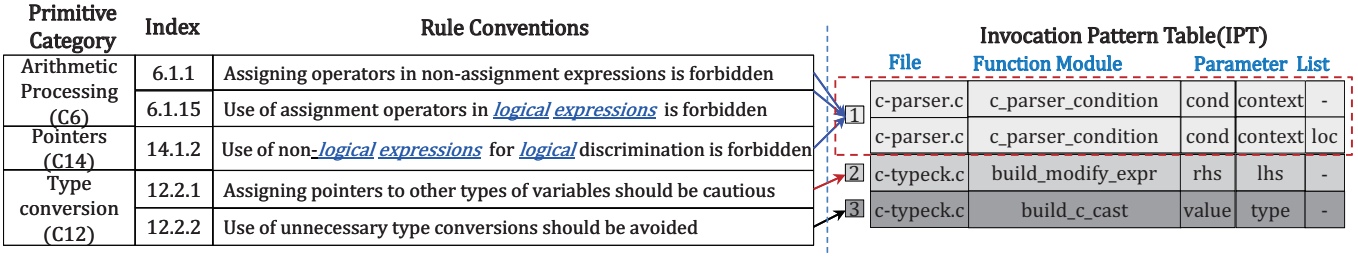

Fig. 2. Examples from SPACE-C user guideline. Rules indexed with their primary categories are demonstrated on the left, while information about their launch points are recorded in the invocation pattern table (IPT) on the right.

launch points for highly similar rules redundantly. This sheds some lights on us that a more intellectual way, i.e., guiding design under certain rule patterns, can save redundant manual efforts and enhance the existing development.

In a rule description, some key words, i.e., rule patterns, can work as hints for their launch points. Fig. 1 demonstrates the pattern-based idea. In pre-design, Rule 14.1.2 is carried out at the inspection spot underlined inside the 14^{th} layer of the calling stack. Here all necessary data it requires can be derived from the launch point's parameter lists in blue. For another Rule 6.1.15 in the lower part of the figure, due to the commonness it has with Rule 14.1.2 in the key attributes of "`expressions`" and "`logical`", it can be recommended directly with the same launch point at function *c_parser_condition* of *c-parser.c*. This means once the similarity (similar words) in coding rules can be uncovered, some experiences from the previous design process can be applied to "*similar looking*" rules. In this way, previous experiences can be applied to new rules naturally. Thus, it is needless to struggle for every launch point any more and the redundant manual locating process can be avoided. We call this method *Referee*, and the similarities in coding rules *rule pattern*.

Referee is based on pattern recognition which facilitates the design of compiler-based analyzers. Unlike the previous manually coding process which designs rules one-by-one, our goal is to ease the burden by automatic recommendations for new rules based on a few implemented rules. We achieve this by first applying popular machine-learning algorithms about patterns in the design process. Referee includes two processes, a training process to recognize similarities in rule patterns and their calling sites, and a recommendation process to apply knowledge on new rules automatically. First, it portrays invocation patterns and deduces similarity information from launch points of partial developed rules. After broadcasting key information to rules by a twin-graph aided method, it then navigates the training on rule characteristics, i.e., rule patterns. Finally, a relation atlas between two kinds of patterns is established, every rule pattern pointing to an invocation pattern (a launch point). In the later recommendation stage, it applies pattern characterization to new rules, and recommends proper launch points according to similarity analyses. In this way, the redundant manually locating overheads can be cut down. Once it fails, Referee improves itself via new patterns supplements and automatic updates.

Our experimental results with SPACE-C and MISRA-C user guidelines show that a design guided by similar patterns can outperform the state-of-the-art one-by-one manual method. Trained on randomly selected seeds of invocation patterns, Referee can recommend successfully for 81.5% to 89.9% new rules, and the corresponding manual efforts can be saved. When enhancing the coverage on invocation patterns and rule features, the accuracy in recommendation can promote synchronously, from a lower rate of 81.5% to 87.7%. Such a pattern-guided methodology represents an encouraging direction for experts' experience and industry standards to be highly efficiently integrated into general-purpose code analyzers.

This paper makes the following contributions:

- Referee, to the best of our knowledge, is among the first efforts to facilitate the manual design process for compiler-based analyzers with machine-learning. This is a preliminary step towards automatic design in compilers, and also an encouraging direction to integrate the expert designers' experiences in the compiler-based analyzers.
- A novel twin-graph aided method for pattern recognition has proposed. It overcomes the insufficiencies in coding rules by digging decisive information from known calling sites, and navigating the characterizing for rule patterns.
- Experiments on rules from SPACE-C and MISRA-C demonstrate the feasibility of Referee. Training with 50% - 70% random rules can yield up to around 90% accuracy in recommendations for launch points of new rules. The accuracy can further improve with broader coverage on pattern seeds. These results also certify the extensibility of Referee to a wider range of user guides.

II. Motivation

In this section, we first describe the existing complicities in manual design (Section II-A). We then examine the challenges faced for an intelligent design (Section II-B). Finally, we introduce our method of Referee, which accumulates knowledge about patterns from developed rules, and applies on new rules by an example (Section II-C).

A. The Manual Design

Current design for an analyzer coupled with compilers is a completely manual decision-making process. Fig. 2 demonstrates 5 rule examples from SPACE-C user guideline and their implementations. On the left, each rule is indexed with

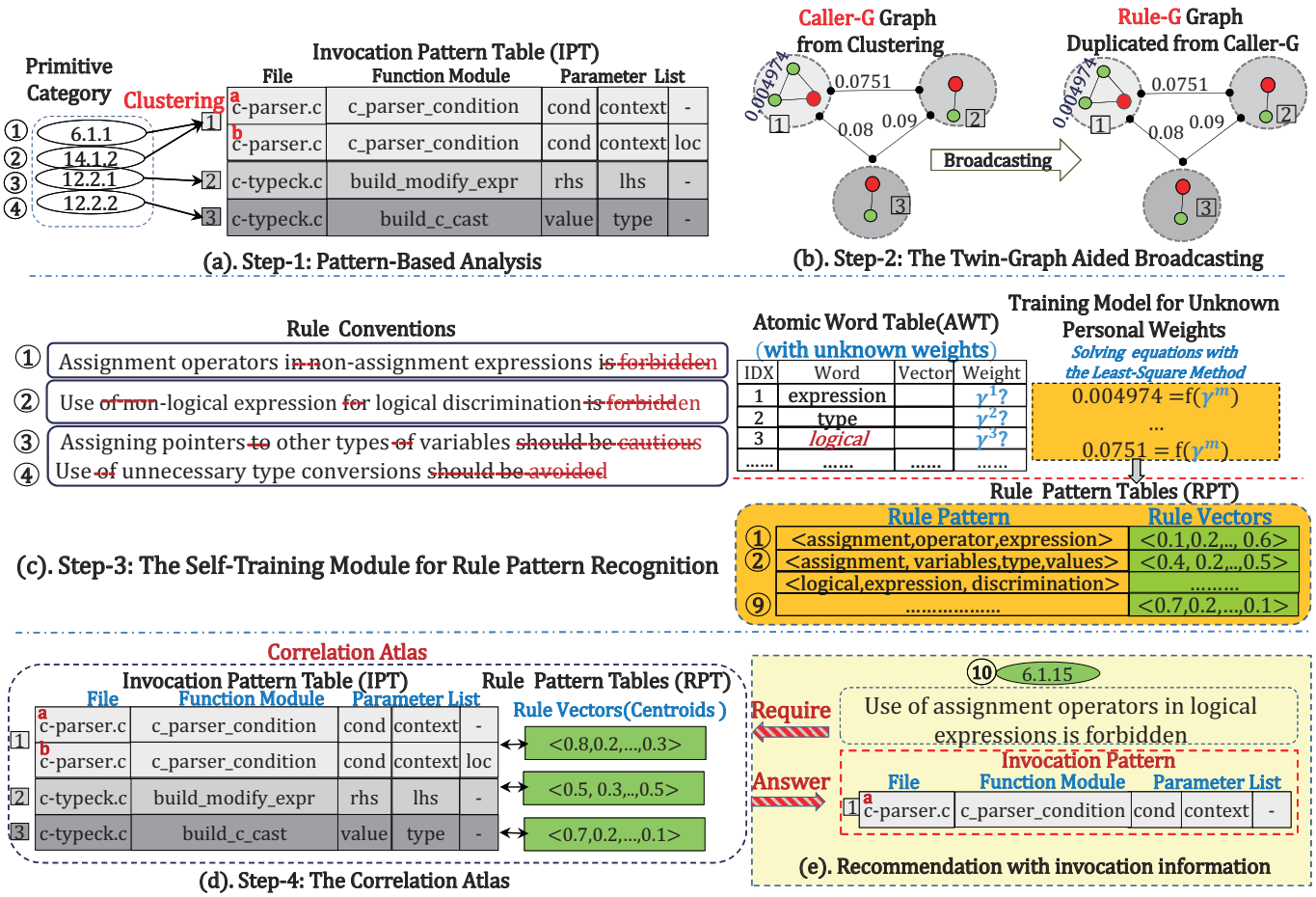

Fig. 3. The automatic self-training and recommendation process of Referee for relieving redundant manual efforts.

M.N.X. *M* represents the primitive category in the guideline, e.g., 6 for Arithmetic Processing (abbr. C6), 12 for Type Conversion (abbr. C12) and 14 for Pointers (abbr. C14). *N* is the strictness of the rules (1 is prohibited usage and 2 is recommended usage), and *X* is the indexed sequence inside the category. The following is the detailed coding rules. On the right, the launch points where these rules should be carried out are recorded in the invocation pattern table (*IPT*). Each item is composed of the hosting file, the function module, and necessary parameters. Inspection codes for these rules have been completed one-by-one by designers in the compiler.

Fig. 2 presents the interleaved and disordered relation between rules and launch points in real implementations. On one side, rules from the same primary category are dispersed due to diversified requirements on information, such as Rule 12.2.1 and 12.2.2 from C12 are designed in the 2^{nd} and the 3^{rd} launch points respectively with different parameters and the callers' name. Conversely, a launch point on the right always serves multiple categories of rules irregularly, e.g., the 1^{st} launch point contains Rule 6.1.15 from C6 and Rule 14.1.2 from C14. These irregularities illustrate the difficulties in the exploration in compilers. Untangling this complexity depends entirely on manual analysis at present.

Despite these irregularities, we have discovered some opportunities about patterns in the figure. Rule 6.1.15 and Rule 14.1.2 in the same launch point of *c_parser_condition* in *c-parser.c*, the 1^{st} point in IPT table(with slightly differences in parameters), have the same words of "`expressions`" and "`logical`". This commonness enlightens us that we can save efforts by making good use of similar-pattern recognition.

B. Challenges

Existing coding rules are still deficient in decisive features or patterns for indicating their launch points. Contrary to natural language environment, or formal specification of a software system which is typically described by dozens or hundreds of properties [7], rules are normally no more than 20 words. High frequency words such as "`forbidden`", "`cautious`", and some prepositions such as "`of`" lack guiding significance in design. Despite these noisy words, there is little meaningful information retained which neither leads to distinct depiction for rules'characteristics, nor reflects where can carry out the inspection for them. We are in need of more clues.

The same launch points where many rules reside in present some inspiration. Rules in the same launch points can be classified naturally as one category, e.g., Rule 6.1.15 and Rule

14.1.2, no matter how different they are. Under such grant, rules are assumed having the consistent similarity trends with their launch points. Then, the similarity analyses from launch points, in turn, can guide the characterizing for rule regulations, which are defined as *rule patterns*. The differences in words of rules can be balanced by a set of *personalized weights*. Ultimately, the major task of Referee turns into training rule patterns, especially personalized weights, under the guidance of available similarity distances between launch points. The accumulated knowledge about both rules and launch points can benefit new coming rules correspondingly. This is the *basic principle* of Referee.

C. Our Referee Solution

Referee aims to relieve manual efforts via accumulated experiences on rule similarities. Not directly dealing with rules, it digs out decisive information from the launch points of implemented rules in the analyzer. The resulted similarity trends should be consistent in their rules and thus can be used to guide the training on similarities of coding rules. Finally we can obtain a group of rule patterns for characterizing rules. The relation between rule patterns and their invocation patterns form a basic knowledge library for new rules in later development. We demonstrate this process of Referee by an example in this subsection.

As shown in Fig. 3, Referee is a two-level self-training process which includes the pattern-guided self-training process in Fig. 3 (a)-(d), and the recommendation process in Fig. 3(e).

We take 4 out of 5 rules from Fig. 2 as training seeds. In Step-1 of Fig. 3(a), their actual launch points are collected and parsed into invocation patterns of <*file name*, *function module*, *parameters*>with CoreNLP [8] and maintained in *IPT*. We use a group of initial personalized weights for vectorization of these patterns so that the similarity distances between every two invocation patterns can be evaluated. The similarity analyses lead to 3 clusters with *kmeans* [9] at the end of the step. Moreover, slightly differences in parameters result in further breakup, e.g., the 1^{st} category breaks into two sub-categories of *a* and *b*. More details about patterns and similarity evaluations are introduced in Section III-B.

In Step-2 of Fig. 3(b), necessary data is broadcasted from invocation patterns to rule patterns. We use two fully-connected graphs to complete the duplication between two types of patterns. The caller graph *(Caller-G)* on the left and the rule graph *(Rule-G)* on the right work as the carriers for two types of patterns. Caller-G is made of invocation patterns denoted with green vertexes. Three circles are the generated cluster categories from Step-1. This step copies clusters, and transfers the similarity distances on every edge, e.g., 0.004974 inside the 1^{st} category or 0.0751 between the 1^{st} and the 2^{nd} categories, from Caller-G to Rule-G. After duplication, Rule-G is almost the same with Caller-G except the rule pattern representations on every vertex. Thus, Referee is ready for the training for rule pattern, especially necessary personalized weights.

Step-3 in Fig. 3(c) demonstrates the core step of self-training for rule patterns. Similar to invocation patterns, each rule pattern is formed by parsing coding rules into atomic words with CoreNLP. Under the premise that rule patterns share the same similarity with invocation patterns, as the orange box on the right shows, we obtain a set of equations which are composed of the available similarity distances and rule patterns containing unknown personalized weights in blue. As introduced in previous Section II-B, the major task of Referee is actually to settle down the personalized weights for rule patterns. With the Nonlinear Least Squares Method (*NLS*), equations are resolved and a group of personalized weights can be obtained which lead to quantized vectors for rule patterns. At the end of this step, Rule Pattern Table (*RPT*) is turned out which records the real-number rule pattern vectors as shown in 3(d).

As the result of the self-training, an important relation atlas between two types of patterns is set up in Step-4 in Fig. 3(d). Each rule pattern in RPT is linked with an item of invocation pattern in IPT. This means that for a new rule, once it can be matched successfully to a known rule pattern in RPT, it can be recommended with a launch point correspondingly.

Fig. 3(e) illustrates the fast scaling for new rules. Take Rule 6.1.15 for example. Referee characterizes its rule pattern with the corresponding weight values and requires similarity analyses with all rule patterns in RPT. After that, it is answered with the 1^{st} major category in IPT, and then with a more suitable sub-category denoted with *a* as its launch point.

III. Approach

Motivated by machine-learning aided optimization [10], [11], Referee is designed as a two-level learning framework. In this section, we first introduce the framework of Referee. Then we detail the design for the self-training process. At the end, we discuss the limitation waiting for further improvement.

A. The Framework

The two layers of Referee are illustrated in Fig. 4. The upper layer is a pattern-guided self-training process, which includes three modules, the pattern-based analysis module on available launch points, the twin-graph aided broadcasting module and the core self-training module for rule pattern recognition. Referee starts from invocation pattern analysis on launch points of partial rules which have been implemented in the compiler, outputs a new classification after clustering. Then, the analysis results are propagated to the corresponding coding rules for training on rule patterns via a twin-graph duplication module. In the core self-training module, Referee deduces a group of rule patterns for characterizing coding rules. Thus, an important mapping atlas is established between rule patterns and invocation patterns, in which each rule pattern is linked to an invocation pattern.

The lower layer of the recommendation process is designed for scalability of Referee for new rules. For a new rule, its rule pattern is first characterized, and then matched in the relation atlas. Once successfully matched, the corresponding invocation pattern can be recommended and thus the corresponding manual

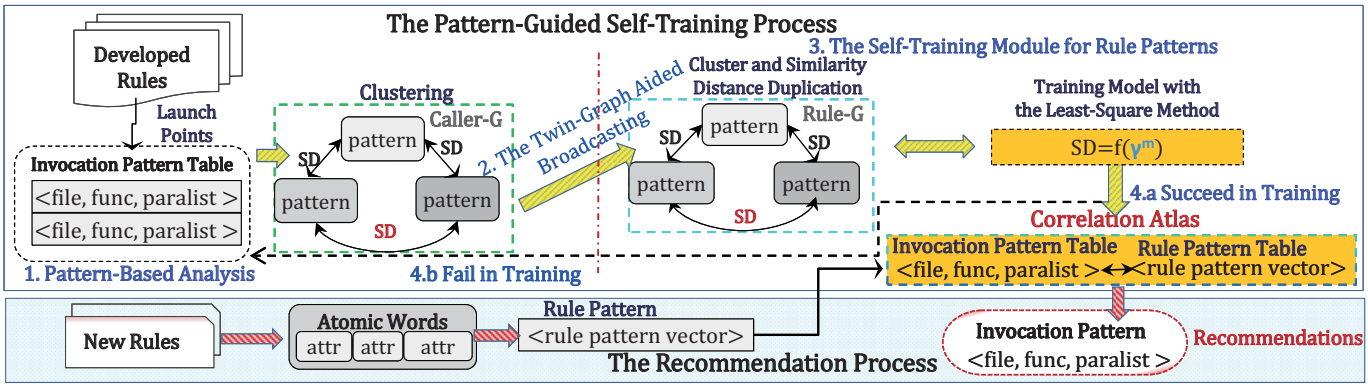

Fig. 4. The two-level framework of Referee.

locating process can be avoided. In this way, the burden on manual exploration can be relieved significantly.

B. The Pattern-Guided Self-Training Process

As a self-training mechanism, the training seeds in Referee should cover a variety types of invocation patterns from multiple different points, not just the large quantity. This can get more diversified rule patterns involved in the training. Moreover, although Referee is based on partial developed rules, we recommend that as many words from the whole context of user guide are utilized for more thorough analyses.

The whole pattern-guided self-training process is based on two types of patterns, the invocation pattern and the rule pattern. An invocation pattern is usually represented with atomic words of no more than *WORDS*(≤ 20) including the file name, the function module, and the required parameters, which are extracted from an available launch point. Similarly, a rule pattern is composed of atomic words from a rule regulation. Basically, analyses on these patterns are similar to that on sentences. Therefore, after vectoring the patterns into real-numbers, Referee can perform similarity analyses for patterns by utilizing mature research achievements from Natural Language Processing (NLP) [12], [13] in our work.

As shown in Fig. 4, Step-1 of Referee starts analyses on invocation patterns. For quantitative analyses on similarities, it needs to vectorize each invocation pattern into real-number. After combining all words with a group of initial personalized weights, the real-number vector for each invocation pattern can be formed. Thus, the similarity distances between every two pattern vectors can be evaluated and all invocation patterns can be clustered into several major categories with *kmeans*.

1) Atomic Words and Personalized Weights: By methods of Continuous Bag-of-Words (CBOW) [12] and word2vec [13], each word is turned into a LEN-*D* real-number vector *wordv* <v^1,v^2,...,v^k , ...,v^{LEN}>, where v^k is a real number on the k^{th} dimension. A few examples are given in Table. I(a), e.g., the $wordv$ of "cond" is *<0.241,...,0.165>*. *LEN* is a tunable boundary for all vectors, which is initialized to 10 in our work due to the short length of launch points and rule regulations.

Each atomic word has a personalized weight *pw* as discussed in Section II-C. It is determined collaboratively by the basic weight *b_weight*, and the special weight *s_weight* as shown in (1). The basic weight *b_weight*, is decided explicitly by the occurrences of the word, *occ*, together with an auxiliary factor α (initialized to 0.001) in the whole SPACE-C context as in (2) [14]. Another auxiliary factor, special weight *s_weight*, is used for a more accurate regulation on the different significance of each word in different rules.

$$\text{pw} = b\_weight * s\_weight \tag{1}$$

$$\text{b\_weight} = \alpha/(\alpha + occ) \tag{2}$$

Referee uses a series of initial *pw* to start the similarity analyses between invocation patterns, which in turn are used to guide the training for another group of *pw* (mainly *s_weight*) used on rule patterns. These *s_weight* values are the key features for characterizing rules.

2) Pattern Vectorization: With atomic word vectors and their personalized weights, a pattern vector can be deduced [14]. A pattern is composed of word vectors of $wordv^1$, $wordv^2$,..., $wordv^m$,..., $wordv^{WORDS}$*(1 ≤ m ≤ WORDS)*, while $wordv^m$ is the vector of the m^{th} atomic word and $wordv^m(k)$ is a real number v^k on the k^{th} dimension. We use pw^{wordv^m} as its personlized weight. Then, the pattern vector is decribed with *pv* <pv^1, pv^2,..., pv^k,..., pv^{LEN} >, while element pv^k ((1 ≤ k ≤ LEN)) is calculated by integrating all v^k from the pattern's composing words and their personalized weights as in (3).

$$\text{pv}^k = \Big(\sum_{m=1}^{WORDS} (wordv^m(k)) * pw^{wordv^m}\Big)/WORDS \tag{3}$$

Table. I(a) demonstrates the quantization process for invocation patterns of Rule 14.1.2 which is described with words <*c-parser.c, c_parser_condition, cond, context*>. After initializing *s_weight* with 0.6, 0.3, and 0.1 for file name, function name, and parameters respectively, a series of *pw*(denoted with β^m) are obtained. The quantized pattern vector for Rule 14.1.2 then can be generated, e.g, *<0.0194, 0.0546, -0.0961,..., -0.0749>*.

TABLE I
EXAMPLE WITH RULE 14.1.2 AND RULE 6.1.1 FOR QUANTIZATION OF PATTERN VECTORS, DEDUCTION OF PERSONALIZED WEIGHTS (pw) AND SIMILARITY DISTANCES (SD).

(a). Quantization process for invocation pattern pv of Rule 14.1.2 during Step-1

Example	Type	word[1]	word[2]	word[3]	word[4]
Rule 14.1.2	words	c-parser.c	c_parser_condition	cond	context
	pw	β^1=1.909	β^2 =0.470	β^3=0.256	β^4=-3.32
	wordv	<0.545,.,0.268>	<-0.046,.,-0.052>	<0.241,.,0.165>	<0.096,.,0.0275>
	pv_1	$<(\sum_{m=1}^{WORDS} wordv^m(1)\times\beta^m)/\text{WORDS},\ldots,(\sum_{m=1}^{WORDS} wordv^m(10)\times\beta^m)/\text{WORDS}>$ = <0.0194, 0.0546, -0.0961,..., -0.0749>			
Rule 6.1.1	pv_2	<0.3608, 0.0708 -0.1287,..., -0.1766>			
SD	$\sqrt{\sum_{m=1}^{10}(pv_1^m - pv_2^m)^2}$ $= \sqrt{((0.0194-0.3608)^2+(0.0546-0.0708)^2\ldots)}$ = 0.0705				

(b). Training for γ and rule pattern pv of Rule 14.1.2 with SD=0.0705 during Step-3

Example	Type	word[1]	pw	word[3]	pw
Rule 14.1.2	words	discrimination	γ^1=0.1495	expressions	γ^2=-1.2393
	wordv	<0.0357, 0.0291,...,-0.0187>		<-0.0225,0.0401,...,-0.0352>	
	pv_1	$<(\sum_{m=1}^{WORDS}\mathbf{word_1^m}(1)\times\gamma_1^m)/\mathbf{WORDS},\ldots,(\sum_{m=1}^{WORDS}\mathbf{word_1^m}(10)\times\gamma_1^m)/\mathbf{WORDS}>$ = < 0.0203, -0.0317, -0.0029,..., 0.0281 >			
Rule 6.1.1	pv_2	$<(\sum_{m=1}^{WORDS}\mathbf{word_2^m}(1)\times\gamma_2^m)/\mathbf{WORDS},\ldots,(\sum_{m=1}^{WORDS}\mathbf{word_2^m}(10)\times\gamma_2^m)/\mathbf{WORDS}>$			

3) Similarity Distance: The similarity distance *SD* between two patterns, either invocation or rule ones, pv_i and pv_j, is calculated with the square root of the sum of all dimension distance squares between per dimension as (4) shows [15]. The lower the distance is, the more similar two patterns are.

$$\text{SD[i,j]} = \sqrt{\sum_{k=1}^{LEN}(pv_i^k - pv_j^k)^2} \tag{4}$$

In Table. I(a), with pv_1 and pv_2 for Rule 14.1.2 and Rule 6.1.1, *SD* of 0.0705 can be generated, which is close enough to cluster them into the 1^{st} category in Fig. 3(b).

4) Clustering: At the end of Step-1, the invocation patterns are clustered into *CLUSTERS* major categories with *kmeans*, as the categories indexed from 1 to 3 in Step-1 of Fig. 3(a). A major category is furtherly broken into sub-categories in order to distinguish those patterns which are slightly different in parameters, e,g., denoted with α and β in IPT of Fig. 3(a). This can make the later recommendation information more practical.

C. The Twin-Graph Aided Broadcasting

Step-2 in Fig. 4 is responsible for data duplication, mainly similarity distances duplication. Referee uses two fully graphs for this information broadcasting. One graph, Caller-G, is built on invocation patterns, and the other, Rule-G, is on rule patterns. The duplication includes almost everything except different pattern representations on every vertex. Thus, Referee is ready for the training for rule patterns.

A rule may have more than one invocation point when containing multiple detection contents. We separate these rules into discrete vertexes to maintain the relation between rules and launch points as one-to-one.

D. The Self-Training Module for Rule Pattern

Step-3 demonstrates the core stage of the self-training process for personalized weights of rule patterns. The core idea of Referee is that rule patterns share the same similarity trends with invocation patterns. A similarity distance from two invocation patterns is also the input for training of the corresponding rule patterns. Table. I(b) demonstrates this process. With SD of 0.0705 between pv_1 and pv_2, a set of equations containing unknown pw values(denoted with γ_i^m, mainly containing unknown *s_weight*) can be set up. With the *NLS* method, a group of γ_i^m values, *wordv*, and *pv* for each rule vector can be deduced in succession, e.g., pv_1 for Rule 14.1.2 is <0.0203, -0.0317, -0.0029,..., 0.0281>. Thus, we can character any new rule with this information.

At the end of the step, the correlation atlas between two types of pattern is established during Step-4.a as shown in Fig. 4. Referee adopts the rule patterns of centroids as the representatives for each category as examples in Fig. 3(d). Thus, new rules can be processed only by examination with a few centroids.

Once Referee fails to solve equations, Step-4.b of re-training will re-start from Step-1 with another group of β values.

E. Recommendation

For a new rule $Rule_t$, aided with personalized weights, it is first characterized for the rule pattern, and then the similarity distances with known centroids are measured for determination on its major category. If necessary, further analyses are performed inside the major category with all vertexes to determine the sub-category. The recommendation is completed with a specified invocation pattern(a launch point).

However, we observed that once the similarity distances between a new rule and all centroids are too high, it is dangerous to manage to recommend. In case of wrong recommendation, we use an empirical value λ as a criteria to remind of artificial judgement. As shown in (5), it warns that all SDs between the new rule $Rule_t$ and centroids are exceeding the boundary and artificial interventions should be involved.

$$\text{MIN}_{m=1}^{CLUSTERS}(SD[Rule_t, Centroid^m]) \geq \lambda \tag{5}$$

F. Core Algorithms for Self-Training

Algorithm 1 and 2 illustrate the kernel algorithms for the self-training process. Algorithm 1 demonstrates three major modules in the pattern-guided self-training process of Referee. Algorithm 2 of Training_γweight demonstrates the self-training process in detail. In this stage, a group of equations are set up according to (3) and (4), and γ values can be resolved with *NLS*. Once failed, Referee restarts from Step-1 with another group of β. The trials are limited under TRY_MAX ($\leq$ 100)

Algorithm 1: Self-Training

Input: $Caller-G, Rule-G$;
Output: $Corelation\_Atlas$;
#define $\boldsymbol{TRY\_MAX}$ = 100;//*Maximum training trials*
Initialize $\boldsymbol{\beta weight} = 0.6, 0.3, 0.1$;
bool $\boldsymbol{Tune\_\beta weight}$ = true;
while $((\boldsymbol{Tune\_\beta weight})\&\&(TRY\_MAX > 0))$ **do**
  //1. Pattern-Based Analysis with Kmeans
  $<Caller\_SD, Clusters, C\_Centroid>=$ $\boldsymbol{Analysis}(Caller\text{-}G, \beta weight, Invocation\_Patterns)$;
  //2. The Twin-Graph Aided Broadcasting
  $<Rule\_SD, Clusters, R\_Centroid>=\boldsymbol{Duplicate}$ $<Caller\_SD, Clusters, C\_Centroid>$;
  //3. Rule Pattern Recognition
  $\boldsymbol{Tune\_\beta weight} =$ $\boldsymbol{Training\_\gamma weight}(R\_Centroid, Rule\_SD)$;
  if *(*$\boldsymbol{!Tune\_\beta weight}$*)* **then**
    $\boldsymbol{Adjust\_\beta weight}()$; *//4. Re-Training*
    TRY_MAX−1;
if *(*$\boldsymbol{!TRY\_MAX}$*)* **then**
  $\boldsymbol{warning}$("manual surveillance!")
else return $\boldsymbol{Corelation\_Atlas}$ $(Invocation\_Patterns, Rule\_Patterns)$

Algorithm 2: Training_γweight

Input: $Centroids, SDs$;
Output: $Tune\_\gamma weight, Rule\_Patterns$;
bool $\boldsymbol{Success}$ = false;
//$\boldsymbol{\gamma weight}$ *Initialization*
Initialize $\boldsymbol{\gamma weight} = 0.6, 0.4$;
bool $\boldsymbol{Tune\_\beta weight}$ = false;

// Set up equations for training on γweight,
// Solving equations with the Nonlinear Least Squares Method.
foreach *cluster* ∈ *Clusters* **do**
  foreach *edge* ∈ *Cluster* **do**
    $<Vertex^i, Vertex^j>=$ $\boldsymbol{GetVertex}(edge, cluster)$;
    $\boldsymbol{Set\_Equations}(Vertex^i, Vertex^j)$;
//Nonlinear Least Squares Method, NLS
$Success = \boldsymbol{Solving\_Equations\_with\_NLS}($ $Rule\_Patterns, \gamma weight)$;

if *(!Success)* **then**
  $\boldsymbol{Tune\_\beta weight}$ = true;

return *(Tune_βweight, γweight, Rule_Patterns)*

times in case of infinite loops, although we usually succeed after few attempts.

G. Limitations

It should be pointed out that, as a self-tuning system, Referee is good at recommending for those new rules whose patterns have been covered by previous experiences. However, it is still possible to fail. The ability of Referee depends on training seeds. Rarely used words in rules can confuse Referee and make it fail to match and recommend. At this time, the only manual effort needed is to add the new invocation pattern in the seeds. Subsequently, Referee can restart the training and enrich itself automatically. Anyway, compared to traditional all-manual mode, Referee is still helpful. Experiments in Section IV certificate its feasibility. More research efforts on deep smarter recommendations are still under-going.

IV. Evaluations

Referee has been implemented to facilitate the design of our GCC8.2-based analyzer. In this section, we validates Referee in reducing redundant manual efforts without generating wrong recommendations by answering the following research questions (RQ):

RQ1: Is Referee feasible and reliable for compiler-based analyzers?

RQ2: Is Referee scalable to other compatible rules else?

RQ3: Compared to manual analyses, is Referee worthwhile and what is the upper line in cutting down manual efforts?

A. Methodology

Referee is evaluated on a rule set from SPACE-C and MISRA-C standards, which contains 163 rules including 22 intersections. Thus, the efficiency of Referee can be validated via the recommendation results for these rules. *RQ1* can be answered by the ratio of successful recommendations. The scalability of *RQ2* is answered through experiments on intersected rules. *RQ3* is investigated via further comparison between Referee and manual exploration.

The rule set is analyzed and designed manually at 69 dispersive launch points in the real design, which forms 69 categories naturally. Among them, about 128 rules have more or less common words in their decisive character words, while the rest 35 rules are characterized by some rarely used words. We evaluate Referee via contrasts of the coincidence between the manually real-designed launch points and the recommended points.

First, we establish a standard for evaluation. All the available 69 real-designed launch points work as the standard answers. Starting from these invocation points, Referee can form 69 category clusters naturally including centroids inside each cluster. It then builds a Caller-G, and then deduces similarity distances between every two rules. We use SDs values between a rule patten and its centroid, $SD^{standard}$, as the standard distance for comparisons in this section.

Referee adopts a portion of rules as training seeds, and tries to recommend for the rest rules, i.e., the validation set. The proportions of training seed rules can lead to different

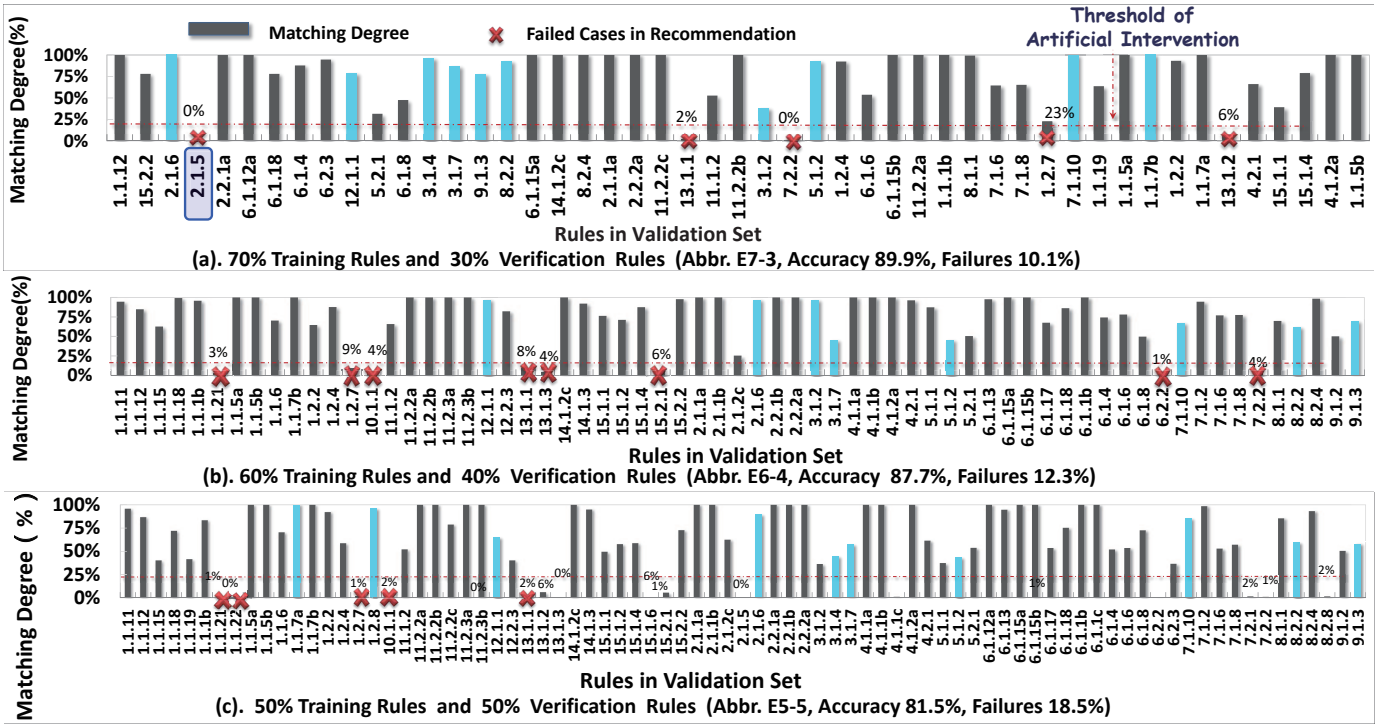

Fig. 5. Validation experiments with randomly selected training seeds and tunable proportion between the training seeds and the validation set.

Words \ Experiments	E5-5	E6-4	E7-3
Rare Words	13.1.1 enumeration	13.1.1 enumeration	2.1.5 code
	13.1.1 element	13.1.1 element	2.1.5 front
	7.2.2 mode	15.2.1 non-standard	13.1.2 consistent
	1.2.7 #pragma	15.2.1 characters	13.1.1 enumeration
	1.1.21 incompleted	6.2.2 paid	13.1.1 element

Fig. 6. Rarely used words in the decisive characters of rule patterns which affect the recommendation.

recommendation results for the validation set. We use SD^{valid} as the new similarity distance between the recommended invocation point and the centroid in the standard category it should belong to. Thus, evaluations on the feasibility and reliability of Referee can be made from two aspects, the matching degree (%) and the accuracy (%). As in (6), the matching degree evaluates the accuracy in recommendation for a single rule via contrasts between $SD^{standard}$ and SD^{valid}. Higher value stands for higher coincidence between the recommended sites and the manual design sites. The accuracy (%) counts the matching degree of the whole validation set, and higher rate means higher feasibility and reliability of Referee. Then the failures (%) count the ratio in the validation set which Referee fails to recommend correctly and require artificial interventions.

$$\text{Matching Degree} = (SD^{valid})/(SD^{standard}) * 100\% \quad (6)$$

B. Experimental Setup

Platform and Compiler: Our working platform is a 20-core server with two 2.2GHz Intel(R) Xeon(R) E5-2630 CPU, Ubuntu 14.04.5 and 256G memory. Referee has been implemented in GCC (version 8.2) due to our further requirements on compiler optimizations.

Benchmarks: Referee provides full supports for test cases from SPACE-C user guide, which is partially compatible with MISRA-C (22 rules).

C. Feasibility and Reliability of Referee

Fig. 5 certifies the feasibility and the reliability of Referee. Experiments are performed with randomly selected training rules. By shrinking the proportion of training seeds from 70%, 60%, to 50%, while increasing the validation set from 30% to 50% correspondingly, experiments show that Referee obtains acceptable accuracy in recommendations on partial developed rules. For clarity, we use E7-3, E6-4 and E5-5 for short in later sub-sections.

Each figure contains the recommendation results for the validation set. Grey bars stand for the matching degree for SPACE-C rules. Blue bars stand for the matching degrees for rules from MISRA-C, which are also intersected rules in these two user guides. In these figures, most rules present higher matching degree values, which mean the recommended calling sites are coincide with the standard launch points. Experiments in Fig. 5(a)(E7-3) demonstrate that once training with enough seeds of 70% rules, Referee can work with an accuracy rate up to 89.9% in recommendation. We also investigate the relation between the accuracy of Referee and the training set in the latter two experiments in Fig. 5. As shown in Fig. 5(b)(E6-4) and (c)(E5-5), by gradually reducing seed rules to 60% and

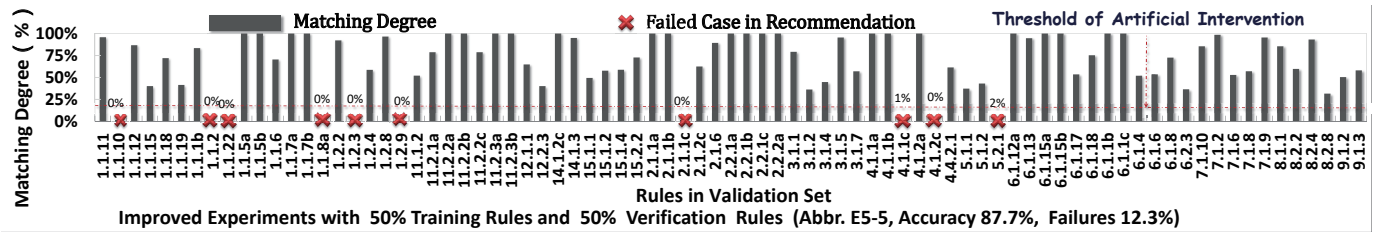

Fig. 7. Experiments on E5-5 with improved converge on training seed patterns.

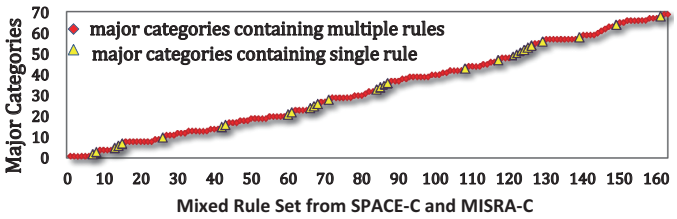

Fig. 8. 69 discrete launch points in the real-design incur heavy manual efforts. Rarely used words in rules lead to special categories only containing single rule, which may incur failures potentially on recommendations.

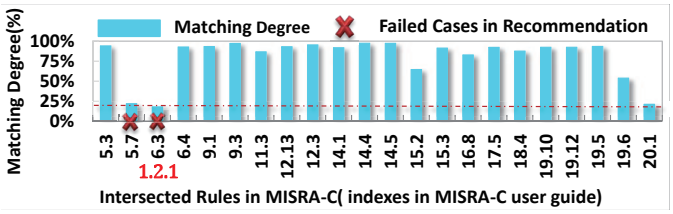

Fig. 9. Training with SPACE-C exclusive rules and verify with intersected Rules of MISRA-C (Accuracy 90.9%, Failures 9.1%)

50% of SPACE-C, Referee still can achieve accuracy of 87.7% and 81.5%, with slightly but still acceptable decline.

Thismeans that the mechanism of Referee is a hopeful measure to save manual exploration efforts with enough sample seeds. Bars denoted with "x" stand for failures in the recommendations. As discussed in Section III-G, Referee still may fail in some cases, e.g., the 35 rules out of 163 containing scarcely used words in decisive characters cited in Fig. 6. In case of wrong recommendation, Referee adopts a high λ on experience as a threshold reminding of artificial intervention, which can result in an extremely low matching degree. We set it $\leq$0.25 as denoted with the red threshold line in Fig. 5, e.g., the matching degree is 0% for Rule 2.1.5. For these cases, Referee will remind the designer of artificial intervention instead of clustering them blindly to any existing categories. In our experiments, the proportion of such failures is relatively low(E7-3 10.1%, E6-4 12.3% and E5-5 18.5%).

We further observe the potential of Referee with improved converge on seeds' patterns. We refine E5-5 due to its lowest accuracy. By absorbing more invocation patterns in the training seeds which have rare words in their corresponding rules, the accuracy increases from 81.5% to 87.7% (failure cases decrease from 15 to 10) in Fig. 7, which approaches the accuracy of 87.7% in E6-4 in Fig. 5(b). We can see that although scarcely used words do impede the accuracy of Referee, it still can be improved once with higher coverage on seed rules' patterns.

D. The Scalability of Referee

The scalability of the mechanism of Referee can be verified via blue bars in Fig. 5. When training with randomly selected seeds from the mixed rule set, most of the intersected rules get success recommendations. Further validations on intersected rules are demonstrated in Fig. 9, when training with 141 exclusive rules from SPACE-C, 20 out of 22 intersected rules(90.9%) achieve coincide results with the standard answers except 2 rules (9.1%) which contain unique words in rule descriptions, e.g., "`signedness`" in Rule 6.3 of MISRA-C(or 1.2.1 of SPACE-C). These results show that Referee is capable of applying in other C-language code rules. In order to be more helpful, it is suggested that researchers or practitioners choose rules which are clearly distinct. This can contribute to a much knowledgeable atlas, and then Referee can play a more pratical role for the design of new rules. It is under-going for Referee to co-operate in the development of compilers, e.g., Loongson Compiler.

E. The Upper Limit of Referee

The upper limit of Referee in saving efforts depends on reasonable coverage on seeds' types. As shown in Fig. 8, our analyzer has 69 discrete launch points which lead to 69 categories. The actual design process requires tedious exploration efforts due to frequently re-position for rules. In this instance, Referee is necessary and helpful.

$$\text{UpLimit(Referee)} = (RULES - CLUSTERS)/RULES \quad (7)$$

All the above analyses demonstrate that Referee has a upper line which, ideally, relies on the number of real categories. As (7) shows, Referee can effectively save efforts for the rest of rules after fully covering CLUSTERS types of invocation patterns. In our work, it means saving about 94 rules only with 69 pieces of manually implemented seed rules(43% of all) which fully cover the 69 categories of invocation patterns. This can reduce the R&D cycle to half.

However, we observed that invocation patterns or rules which contain rare words can pull down the capability of Referee. Further statistics on yellow icons in Fig. 8 show that about 36 out of 69 categories contain single rule due to their scarcely used words. These invocation patterns and rules can only play

one role, either as a training seed which can not be validated, or a validated rule which fail to find mature experiences from training. Either way can pull down the accuracy of Referee. Experiments in Fig. 5 have certificated that they are the main source of recommendation failures because of missing matched categories. Once combined coding rules with researches in formal description [16], Referee can further weaken the side-effects of rarely used words. Thus, Referee is such a promising method to in-cooperate the expert designer's experiences in the analyzer design more efficiently.

V. Related Work

Referee has been motivated by machine-learning aided programming, which falls into the joint area of code analyzers and automatic compilation.

Static Code Analyzers: Industry coding conventions can enhance the safety and robustness of codes. Most of them focus on frequently used languages such as Java [1], [2] or C [5], [6]. Correspondingly, code analyzers are under ongoing demand in order to unearth potential flaws in coding efficiently [4], [17]–[22]. There are many open-source analyzers such as Checkstyle [17], FindBugs [19] or some commercial tools such as Coverity [4] famous for C and Java language inspection. Tools such as Astrée [23], PVS-Studio Analyzer [24] focus on runtime errors, or certain types of problems such as about arrays, and RuboCop [21] in code format. Many of these techniques support cross-language coding conventions and extensive APIs by users [4], [25].

Machine-Learning in Analyzer: Machine-learning in static code analyzers is relatively scarce. NATURALIZE [26] is a general learning framework which solves the coding convention inference via learning on code styles. The study in [27] has quantified the relation between rule violations and actual faults with empirical data for the MISRA C 2004 standard.

Machine-Learning in Compilation: Machine-learning in compilation has been an active research area for several years. A number of studies have emerged for relieving manual efforts by automatic compilation due to the complexity of compiling techniques. They initiated from iterative-compilation [28]–[30] or auto-tuning [31] on performance. They centered around specific optimizations such as unrolling or register allocation, and then stepped into more mature areas including optimisation space [32]–[34] for performance prediction [35]–[37], and performance maximisation [37], improvement on learning algorithms [38], feature engineering [39] with systematic representations, or reduction on learning costs [30].

Of late, some researches penetrated into wider application with machine-learning algorithm. The study in [40] has extended adaptive automatic program transformation to more architectures for data-parallel languages. New compiler technique combined with structured genetic tuning algorithms is proposed for variable-accuracy algorithms with better trade-off between time and accuracy in compiler [41]. CLgen is proposed as a machine learning based generator for OpenCL programs for better performance [42]. Newly emerging investigation have been devoted in code size reduction [43], [44].

There are some thorough surveys about automatic compiling techniques. Research in [45] details the machine-learning-based compilation techniques of the latest stage. It has performed in-depth analyses for different learning methods in performance predication and learning costs, and investigated potential challenges in research directions. Research in [46] presents deep thoughts in learnable probabilistic models on code patterns at the intersection of machine learning, programming languages, and software engineering recently. It also gives unique discussion on cross-cutting and application-specific challenges and opportunities.

Machine-Learning in Detection and Verification: Machine-learning has been used for bug detection, such as TAC for static use-after-free(UAF) detection [11], software defect prediction [47], memory leaks detecion [48], buffer overflow detection [49], violations of temporal safety detection by compile-time transformation [50], etc.

Studies in program verification and source code transformation have inspired our work. Feature engineering [39] of auto-tuning are the base for characterizing or description in natural language. Related work in natural language generation includes characterizing specific fault patterns for programming error prevension [51], code summarization via a high level natural language description for better software maintenance and code categorization, code auto-generation such as producing programmers' style benchmarks for verification [10], [52], automatic program repair systems based on systematic analysis of key characteristics [53].Studies about code transformation also make language processing more convient and efficient. [54], [55]Machine-learning has been integrated flexibly in areas such as software maintenances, e.g., code smells combined with deep learning for software refactoring instead of manually identifying certain code structures [56].

Our work is on the way to automatic design for code analyzers. Referee combines automatic feature analysis together with the design of static analyzers, but with key differences in feature patterns and the learning model. We are still going on to combine studies such as declarative specifications or pattern characterizing [16], [51] in so that Referee can benefit from formal formats in rules, and can be spreaded into wider area.

VI. Conclusion

We present Referee, a pattern-guided design approach for compiler-based analyzers, and demonstrate the feasibility with high accuracy in guiding the location of launch points for new rules. Referee can work efficiently via 1) carefully selected training seeds, 2) pattern-based analysis and 3) a machine-learning aided rule pattern recognition. It is an effective method to integrate the expert designer's experiments in the analyzer.

Acknowledgments

We would like to particularly acknowledge Professor Wei Huo of Institute of Information Engineering, CAS, China and the anonymous reviewers who helped us improve the quality of our paper in its final version.

REFERENCES

[1] Alibaba, "Alibaba java coding guidelines pmd implements and ide plugin," 2017. [Online]. Available: https://github.com/alibaba/p3c/

[2] Google, "Google java style guide," 2017. [Online]. Available: https://checkstyle.org/styleguides/google-java-style-20170228.html

[3] Sun, "Code conventions for the java programming language: Contents," 1997. [Online]. Available: https://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

[4] Coverity, "Coverity static application security testing." 2019. [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html,VisitedMarch9

[5] MISRA, "Misra 2012," 2012. [Online]. Available: https://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/Default.aspx.

[6] SPACE, "Gjb 5369 2005 space model software language c safe subset." 2005. [Online]. Available: https://download.csdn.net/download/zhangqian_zhangqian/10225409

[7] S. Apel, D. Beyer, V. Mordan, V. Mutilin, and A. Stahlbauer, "On-the-fly decomposition of specifications in software model checking," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 349–361. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950349

[8] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," 01 2014.

[9] Kmeans, "Kmeans," 2019. [Online]. Available: https://www.mathworks.com/help/stats/kmeans.html,VisitedMarch9

[10] G. Ye, Z. Tang, D. Fang, Z. Zhu, Y. Feng, P. Xu, X. Chen, and Z. Wang, "Yet another text captcha solver: A generative adversarial network based approach," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'18. Toronto, ON, Canada: ACM, 2018, pp. 332–348.

[11] H. Yan, Y. Sui, S. Chen, and J. Xue, "Machine-learning-guided typestate analysis for static use-after-free detection," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017. New York, NY, USA: ACM, 2017, pp. 42–54. [Online]. Available: http://doi.acm.org/10.1145/3134600.3134620

[12] CBOW, "Continuous bag of words," 2019. [Online]. Available: https://iksinc.online/tag/continuous-bag-of-words-cbow/,VisitedMarch9

[13] word2vec, "Coverity static application security testing." 2019. [Online]. Available: https://en.wikipedia.org/wiki/Word2vec,VisitedMarch9

[14] S. Arora, Y. Liang, and T. Ma, "A simple but tough-to-beat baseline for sentence embeddings," 2016.

[15] Euclidean, "Euclidean distance," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_distance,VisitedMarch9

[16] F. Molina, C. Cornejo, R. Degiovanni, G. Regis, P. F. Castro, N. Aguirre, and M. F. Frias, "An evolutionary approach to translate operational specifications into declarative specifications," in *Formal Methods: Foundations and Applications*, L. Ribeiro and T. Lecomte, Eds. Cham: Springer International Publishing, 2016, pp. 145–160.

[17] CheckStyle, "Checkstyle," 2019. [Online]. Available: http://checkstyle.sourceforge.net/

[18] Checkmarx, "Checkmarx," 2019. [Online]. Available: http://www.binqsoft.com/solutions/checkmarx?audience=309202,VisitedMarch9

[19] FindBugs, "Findbugs^{TM} - find bugs in java programs," 2019. [Online]. Available: http://findbugs.sourceforge.net/,VisitedMarch9

[20] P. Jtest, "Parasoft jtest," 2019. [Online]. Available: https://www.eswlab.com/products/parasoft/jtest/,VisitedMarch9

[21] RuboCop, "Rubocop," 2019. [Online]. Available: https://www.rubocop.org/en/stable/,VisitedAug12

[22] D. Hou and H. J. Hoover, "Using scl to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.

[23] ASAnalyzer, "Astrée static analyzer," 2019. [Online]. Available: http://www.astree.ens.fr/,VisitedMarch9

[24] P.-S. Analyzer, "Pvs-studio analyzer," 2019. [Online]. Available: https://www.viva64.com/en/pvs-studio/,VisitedMarch9

[25] PMD, "Pmd," 2019. [Online]. Available: https://pmd.github.io/pmd-6.10.0/index.html,VisitedMarch9

[26] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[27] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *2008 IEEE International Conference on Software Maintenance*, Sep. 2008, pp. 277–286.

[28] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimisation space," 1998.

[29] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 237–. [Online]. Available: http://dl.acm.org/citation.cfm?id=517554.825767

[30] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *International Symposium on Code Generation and Optimization (CGO'06)*, March 2006, pp. 11 pp.–305.

[31] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," 11 2008, p. 4.

[32] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 204–215. [Online]. Available: http://dl.acm.org/citation.cfm?id=776261.776284

[33] J. D. Thomson, "Using machine learning to automate compiler optimisation," Tech. Rep., 2008.

[34] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *Fourth IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2006. New York, New York, USA: IEEE Computer Society, 2006, pp. 319–332.

[35] J. Cavazos and J. E. B. Moss, "Inducing heuristics to decide whether to schedule," in *IN PROCEEDINGS OF THE ACM SIGPLAN'04 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION*. ACM Press, 2004, pp. 183–194.

[36] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *In Artificial Intelligence: Methodology, Systems, Applications*. Springer Verlag, 2002, pp. 41–50.

[37] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. P. O'Boyle, "Portable compiler optimisation across embedded programs and microarchitectures using machine learning," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 78–88. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669124

[38] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 147–162. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384628

[39] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *Artificial Intelligence: Methodology, Systems, and Applications*, D. Scott, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 41–50.

[40] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 455–466. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628087

[41] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *International Symposium on Code Generation and Optimization (CGO 2011)*, April 2011, pp. 85–96.

[42] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2017, pp. 86–99.

[43] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press,

2019, pp. 149–163. [Online]. Available: http://dl.acm.org/citation.cfm?id=3314872.3314892

[44] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/2597809.2597811

[45] Z. Wang and M. O'Boyle, "Machine learning in compiler optimisation," *CoRR*, vol. abs/1805.03441, 2018. [Online]. Available: http://arxiv.org/abs/1805.03441

[46] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 81:1–81:37, Jul. 2018. [Online]. Available: http://doi.acm.org/10.1145/3212695

[47] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884804

[48] S. Lee, C. Jung, and S. Pande, "Detecting memory leaks through introspective dynamic behavior modelling using machine learning," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 814–824. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568307

[49] L. Li, C. Cifuentes, and N. Keynes, "Practical and effective symbolic analysis for buffer overflow detection," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882338

[50] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for c," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. New York, NY, USA: ACM, 2010, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1806651.1806657

[51] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 486–497. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238170

[52] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 379–390. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737969

[53] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 727–739. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106253

[54] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight transformation and fact extraction with the srcml toolkit," in *2011 11th IEEE Working Conference on Source Code Analysis and Manipulation*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2011, pp. 173–184. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SCAM.2011.19

[55] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[56] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238166