Patching Locking Bugs Statically with Crayons

JUAN CRUZ-CARLON and MAHSA VARSHOSAZ, IT University of Copenhagen, Denmark
CLAIRE LE GOUES, Carnegie Mellon University School of Computer Science, USA
ANDRZEJ WASOWSKI, IT University of Copenhagen, Denmark

The Linux Kernel is a world-class operating system controlling most of our computing infrastructure: mobile devices, Internet routers and services, and most of the supercomputers. Linux is also an example of low-level software with no comprehensive regression test suite (for good reasons). The kernel's tremendous societal importance imposes strict stability and correctness requirements. These properties make Linux a challenging and relevant target for static automated program repair (APR).

Over the past decade, a significant progress has been made in dynamic APR. However, dynamic APR techniques do not translate naturally to systems without tests. We present a static APR technique addressing sequential *locking API misuse* bugs in the Linux Kernel. We attack the key challenge of static APR, namely, the lack of detailed program specification, by combining static analysis with machine learning to complement the information presented by the static analyzer. In experiments on historical real-world bugs in the kernel, we were able to automatically re-produce or propose equivalent patches in 85% of the human-made patches, and automatically rank them among the top three candidates for 64% of the cases and among the top five for 74%.

CCS Concepts: • **Software and its engineering → Software defect analysis;**

Additional Key Words and Phrases: Automated repair, static program repair, api misuse

ACM Reference format:
Juan Cruz-Carlon, Mahsa Varshosaz, Claire Le Goues, and Andrzej Wasowski. 2023. Patching Locking Bugs Statically with Crayons. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 56 (April 2023), 28 pages.
https://doi.org/10.1145/3548684

1 INTRODUCTION

Automatic Program Repair (APR) aims to repair *buggy* programs by automatically constructing source-level patches [1]. A large proportion of the techniques proposed over the past decade use test cases to guide repair. That is, the correctness of a program for APR purposes is defined as passing tests in a provided automated test suite (e.g., References [2–10]). At a high level, these types of techniques begin by applying a fault localization mechanism to the provided tests and buggy program to identify likely locations for repair, and then generate or synthesize candidate patches

This work is supported by Independent Research Fund Denmark under Grant Agreement 8022-00326B and by National Science Foundation, Grant No. CCF-1750116.
Authors' addresses: J. Cruz-Carlon, M. Varshosaz, and A. Wasowski, IT University of Copenhagen, Copenhagen, Denmark; emails: {alcr, mahv, wasowski}@itu.dk; C. Le Goues, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, USA.

1049-331X/2023/04-ART56
https://doi.org/10.1145/3548684

for these locations seeking those that cause all tests to pass (called plausible patches) [2, 8]. Careful algorithm design typically aims to increase the probability that a plausible patch will generalize reasonably to other test sets [11]. The tests, and the ability to run them efficiently and automatically, are thus key to the expressive power, quality, applicability, and efficiency of these methods.

However, efficient and fully automated test suites are not always available. Tests can be inherently hard to write or automate for certain classes of systems (e.g., long-lived systems with long accumulated legacy code), and even if automated at high cost, the benefits can be limited. This category of software includes systems-level software and embedded control software in cyber-physical systems and robotics. A good example is the Linux Kernel (the subject system of this article): Since a very substantial part of its codebase interacts directly with hardware, it contains very few automated tests. As thousands of hardware components cannot possibly coexist on a single computer system, it would not be possible to execute a test suite for all of them and would be practically impossible to maintain (the list of supported hardware is changing rapidly). Moreover, tests for core kernel components must be run virtually, since the test infrastructure requires an operating system to run. Even if feasible, such a setup would still leave many properties untestable or difficult to diagnose. For these reasons, the kernel project famously and effectively relies on code review for accepting new contributions, and on specialized tools supporting manual bug triaging. Operating system kernels and drivers are also ripe targets for static analysis tools [12], which are well-positioned to find property violations that are often critical in such systems (e.g., resource leaks, concurrency errors) but particularly difficult to test for deterministically.

How do we build APR tools for systems with hard-to-find bugs, and limited access to test execution? One promising answer is *static* APR [13, 14]. Static program repair relies on static analyzers to identify bugs and to localize, construct, and reason about candidate patches. The main challenge with using static analyzers to guide program repair, however, is that they reason about programs via abstraction. As a result, they typically ignore finer grained details of program state, as well as overall notions of correctness, in the interest of tracking only sparse general properties. In addition to posing practical challenges for patch construction (which must contend with actual source code), this poses additional challenges to patch correctness. Put simply, many scalable static analyzers explicitly disregard concerns about whole-program functional correctness, and thus "overfitting" patches can be particularly easy to construct. For instance, consider a double-lock bug finder applicable to Linux Kernel code. Such an analyzer can detect locking errors in real-world code (two consecutive lock operations on the same lock in the same thread). However, trivially, such a detector will also happily declare a program correct if all locks were removed! Such a repair is almost never correct, as it introduces race conditions, even though the analyzer in question is satisfied. Instead, additional reasoning is required.

Our insight is that, following the philosophy of certain classic APR techniques [15], it can be possible to learn just enough additional correctness specifications from other places in the same code base. Our patch ranking tool uses both statically recovered information (memory regions and effects) and syntactic information (types and identifiers) as features.

In this article, we attempt to realize this idea for locking API misuse bugs. We develop a specialized repair technique based on *coloring* the program control flow graph to synthesize a set of repair candidates. We apply information retrieval techniques to analyze the other places in the code that use the incriminated lock and learn an estimation of the critical resource being protected by the lock. Finally, we use this estimator to rank different repair proposals with respect to how well they protect the critical resource. Overall, the contributions of this work are:

- A formal characterization of possible API misuses for a two-function resource reservation API (such as lock–unlock),

```
1    int a=0;
2 +  mutex_lock(&b->l);
3    while (a == 0) {
4 -    mutex_lock(&b->l);
5      if (b->head != Null) {
6        a=c+a; ...
7      }
8      f(a)
9    }
10   mutex_unlock(&b->l);
11   return a;
```

```
1    int a=0;
2    while (a == 0) {
3      mutex_lock(&b->l);
4      if (b->head != Null) {
5        a=c+a; ...
6      }
7      f(a)
8 +    mutex_unlock(&b->l);
9    }
10 - mutex_unlock(&b->l);
11   return a;
```

Fig. 1. A double-lock bug example with two ways to fix it.

- A static APR method and implementation that synthesizes repairs for locking API misuses, by graph coloring and ranking,
- A method to approximate the criticality of a line of code (i.e., likelihood of the line being in a critical section) given a lock object, which can be applied to learn about code criticality in an error trace reported by a static analyzer,
- An evaluation on the actual Linux Kernel codebase demonstrating that the APR method synthesizes and prioritizes patches that are identical to those created by humans.

The technique does not require manual interventions or human written specifications, and works directly on the Linux Kernel code. The key significance of these contributions lies in demonstrating how even an extremely coarse grained, fast, and imprecise static analyzer can be used as a basis for an APR procedure by supplementing it with (also static) learning capabilities.

The technique is implemented in a prototype tool, **Crayons**. We evaluated **Crayons** on several versions of the Linux Kernel code. We were able to automatically re-produce or propose equivalent patches for 85% of the human-made patches, and automatically rank 63% of them among the top three candidates and 74% among the top five. Moreover, the experiments show that the contribution of the static analysis-based features is more significant than that of syntactic features (program text) for effective ranking of patches.

While we restrict attention to the Linux Kernel project, we note that this project is of critical importance and scale (over 20 million lines of code, over 4,000 contributors, and over 400 contributing companies). From a methodological perspective, the kernel contains a large diversity of systems code, more than a collection of smaller evaluation subjects could cover. At the same time it imposes serious scalability and maturity requirements on our tools. So, while the techniques are not limited to the kernel code, the kernel does appear as a challenging benchmark: If we can effectively repair bugs here, then we should be able to extend the tool to handle other projects.

2 OVERVIEW

Figure 1 presents a simple illustrative example of a double-lock error inspired by a historical bug from the Linux Kernel commit history (`ca9fe1588427`).[1] We are using the Unix diff notation, so the buggy version is shown by the lines without plus prefixes. Consider the version before patching, on the left-hand side. Execution begins by entering the while loop in Line 3, and then takes the lock (`b->l`) for the first time on Line 4. The lock is held for the entire iteration. At the second iteration, a double-lock occurs in Line 4; the thread thus hangs indefinitely, as mutex locks are non re-entrant.

[1]Commit hashes in this article can be used to access the kernel history as follows: https://github.com/torvalds/linux/commit/ca9fe1588427.

The original code was fixed by the patch shown in the left-hand side of Figure 1, moving the lock *acquisition* before the loop. An alternative repair, shown in the right-hand side of the figure, moves the lock *release* inside the loop. The choice between these two repairs depends on which resources in the program are protected by the lock, i.e., which lines constitute a critical section. In the example, if the lock `b->l` protects only the variable `b`, then the unlock call can be moved to a point in the loop after which `b` is not manipulated anymore (the right-hand side). If the lock protects a data structure iterated on by the loop, of which `b` is just a single link, then the left-hand code is preferable, as it makes the entire traversal atomic. Although it is tempting to conservatively select a larger critical section, a smaller critical section allows interleaving of loop iterations with another thread accessing the resource. Which of these is preferred is application specific.

We use this example throughout this section for illustration of the key ideas behind our technique for static repair of lock API misuses. We begin by describing EBA, the static analyzer on which we build our repair technique (Section 2.1) before providing a problem statement (Section 2.2) and an overview of our approach (Section 2.3). We elaborate technical details of the approach in subsequent sections.

2.1 EBA Bug Finder

We build our static bug repair tool on an existing static analyzer, EBA. EBA is a static finder for resource manipulation bugs, i.e., bugs that can be specified as misuses of an API captured by a finite state monitor of a resource that is represented as a pointer in a C program [16]. EBA is fast but relatively imprecise; it approximates the efficiency of linters while adding a lightweight semantic abstraction in the analysis. This allows it to scale to the Linux Kernel code base. EBA reasons about a program using its control-flow graph annotated with computational effects and aliasing information. A *region* is an abstraction of a memory location in which program values are stored. If two variables are (potentially) aliased, then they are represented by the same abstract region. A computation is summarized by the *effects* it may produce, e.g., read or write a memory location, acquire or release a lock. An effect is typically parameterized by a memory region, so that the tool knows which variable is read, which lock is acquired, and so on. Let $\mathrm{r}(i)$ be the region of variable i. Then, we can summarize the entire program in Figure 1 (repaired or not) by the following unordered set of effects:

$$\begin{aligned}&\{\mathtt{read}_{\mathrm{r}(\mathtt{a})}, \mathtt{write}_{\mathrm{r}(\mathtt{a})}, \mathtt{read}_{\mathrm{r}(\mathtt{b})}, \mathtt{read}_{\mathrm{r}(\mathtt{b->l})}, \mathtt{read}_{\mathrm{r}(\mathtt{\&b->l})},\\ &\ \mathtt{read}_{\mathrm{r}(\mathtt{b->head})}, \mathtt{read}_{\mathrm{r}(\mathtt{\&b->head})}, \mathtt{lock}_{\mathrm{r}(\mathtt{\&b->l})},\\ &\ \mathtt{unlock}_{\mathrm{r}(\mathtt{\&b->l})}, \mathtt{read}_{\mathrm{r}(\mathtt{c})}, \mathtt{read}_{\mathrm{r}(\mathtt{f})}, \dots\}\end{aligned} \tag{1}$$

EBA can track other resources and APIs beyond locks, which brings the potential of generalizing our static repair method to other kinds of bugs; however, in this article, we only consider locking problems. EBA calculates summaries per node in the control flow graph, so the sets tend to be small, with the exception of function call sites (where the set summarizes the entire function body). Table 1 shows example **Control Flow Graph (CFG)** nodes generated from the left-hand side of Figure 1.

As we describe in subsequent sections, our technique requires labeled datasets for a learning component. We therefore extend EBA to print CFGs accordingly (specifically mapping colors to CFG nodes, as described in Section 2). We implement this via a monitor automaton applied while traversing a CFG; extending our approach to different types of bugs therefore requires only providing a suitable such monitor.

Table 1. An Annotated CFG Entry for the Example of Figure 1

Line#	Source code	Effect[Regions]	Variable name/type
1	int a=0	read[reg23],write[reg23]	a/int, const
2	mutex_lock(&b->l);	!?f136, read[reg152,reg161], lock[r731]	mutex_lock/function, b/struct debug_list*
3	while (a == 0)	read[reg23]	a/int
5	if (b->head != Null)	read[reg152]	b/struct debug_list*
6	a=c+a;	read[reg23,reg349], write[reg349]	a/int, c/int
8	f(a)	!?f4598, read[reg23], write[reg23]	a/int, f/function
10	mutex_unlock(&b->l);	!?f67, read[reg152,reg161],unlock[r731]	mutex_unlock/function, b/struct debug_list*

Four features are maintained in a CFG: effects, regions, variable names, and types. The identifiers in square brackets refer to regions assigned by EBA.

2.2 Problem Statement

We focus on generating repairs for sequential lock manipulation bugs in C programs. We assume no tests, but generate the repair entirely statically, assuming that a static analyzer has detected the bug and that we have access to the analyzer's internal representation of the program to understand how locks are used in the code base. In particular, we assume a report issued by the EBA static analyzer identifying such a bug. To produce acceptable patches for lock API misuse defects, a static APR tool needs to "understand" which resources are being protected—a piece of information that we do not have a priori. C has no explicit annotations defining critical resources. *This lack of information guiding the repair strategy is inherent to static program repair. Static analyzers track specific generic properties and do not reason about complete program specifications. An APR tool based solely on abstractions available in a particular static analyzer operates with very sparse information.* While a generated candidate repair might satisfy the static analyzer, the tool is left in the dark regarding whether other errors have been introduced, especially in the absence of comprehensive test suites.

We consider two types of errors in static repair: (i) errors *tracked* by the static analyzer and (ii) errors *opaque* to the analyzer. Atomicity violation is the example of the latter, because critical sections are not explicit in C code. An example of the former is an *unlock-without-lock*.

The goal of this article is to generate patches for these statically detected *tracked* bugs, minimizing the possibility of introducing *opaque* bugs. We do this by combining a static reasoning approach to determine patch candidates, supplemented with information retrieval techniques, to infer knowledge about the correct uses of locks in the program under repair.

2.3 The Approach

Our solution consists of two high-level components: static reasoning to generate patches that can tackle the API misuse bug in question, and an information retrieval-based analysis to provide information on critical sections based on available usage data.

Patch synthesis. We designed our patch generation model to be *unbiased*, making no assumption on what should be protected, and *simple*; translating to the model and back from it must be as simple as possible. We achieve this by translating lock manipulation API calls and lock states to colors on vertices of the CFG and expressing bugs as ill-colored edges. These errors are fixed by manipulating the graph and the coloring until no errors are present or until a maximum repair

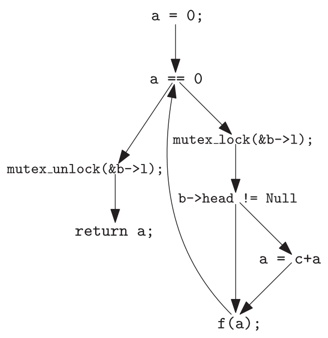


Fig. 2. The CFG of the code in Figure 1.

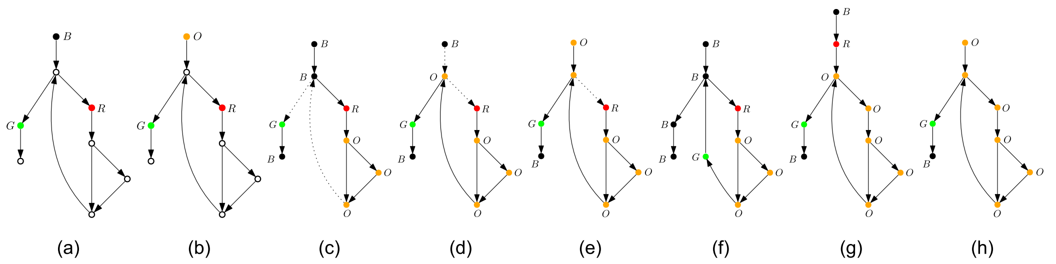


Fig. 3. The two initial colorings (a, b), their three induced colorings (c–e), and their corresponding repairs (f–h) for the code snippet in Figure 1. Edges linking vertices with inconsistent colors are dotted.

threshold is reached. Let us discuss the example of Figure 1 to develop the key intuitions. We describe the technical details in Section 3.

Figure 2 shows the statement CFG of the code snippet in Figure 1. We decorate it with four colors: *red* (R), *green* (G), *orange* (O), and *black* (B), representing the lock state along a potential execution. We color all the vertices in which the lock &b->l is being taken with *red*, and all the vertices in which it is being released with *green*. Since the entry vertex in the CFG is not a lock operation, it can only be colored with one of the other colors: *black* or *orange*, meaning that the vertex is outside, respectively, inside, a critical section. This gives rise to the two *initial colorings* for our example, shown in Figures 3(a) and 3(b).

The next step is to *induce* total colorings, where all vertices have a color assigned, following two rules: If a vertex is *red* (*green*), then all its uncolored successors are colored *orange* (*black*). If a vertex is colored *orange* (*black*), then we just propagate the same color to successors. Figures 3(c)–3(e) show all the induced colorings for the initial colorings of Figures 3(a) and 3(b).

Every *orange* vertex is reachable with the lock held, and every *black* vertex is reachable with the lock released (assuming all paths are feasible in the program). Consequently, Figure 3(a) specifies that the CFG is entered with the lock free, and Figure 3(b) assumes that the CFG is entered with the lock held. Similarly, the existence of the back edge in Figures 3(c) and 3(d) implies that in a set of possible executions, the branching vertex can be reached both with the lock released and held.

In Figure 3(c), the back edge is *ill-colored*, because the control flow reaches the same vertex with the lock taken and free. Independently, the other dotted edge in the figure is also ill-colored, this time because the vertex labeled G can be reached in a state in which the lock is not taken (a black to green shift). All the ill-colored edges in Figures 3(d) and 3(e) are dotted as well.

To repair the back ($O \rightarrow B$) edge in Figure 3(c), we remove it, insert a *green* vertex, and add the $O \rightarrow G, G \rightarrow B$ edges (i.e., insert a lock release at the end of the loop); to repair the left dotted edge, we can *remove* the green color (see Figure 3(f)). With an analogous analysis, we can derive the repairs shown in Figures 3(g) and 3(h). We do not consider addition nor removal of orange or black vertices, because they do not modify the coloring of their successors.

To report a repair to the developer, every graph and coloring manipulation performed is translated to lock operations. For example, the edits in Figure 3(f) are reported to the developer as: *(i) Remove unlock operation at x (ii) Insert lock operation between u and v*. Where x, u, and v are CFG nodes identifiers.

Patch ranking. The patch generator returns many patch proposals for each buggy function. We subsequently *rank* them to produce the order in which they are to be presented to the developer.

We use a simple logistic regression model to choose among possible repair patches (possible critical sections in our example), aiming at minimizing the opaque errors. The model approximates criticality for lines of code. It is trained on code from the same project that shows no locking errors, but where the same resources (i.e., the same variables) are manipulated. If we observe that certain resources are protected with the same lock elsewhere, then we gain confidence that lines manipulating them in the problematic code should also be protected. We estimate the level of confidence into a repair as a real number, depending on how well the regression model trained on other uses of code agrees with the coloring produced by the patch generator. Then, we rank the patch proposals according to these estimates.

To learn a model of criticality—that is, that predicts whether a line belongs to a critical section or not—we use EBA to generate a dataset of CFGs. The CFGs are generated from correct code, with no tracked errors, that uses the lock participating in the bug under repair. Each CFG contains *features* that are either syntactic (e.g., variable names) or statically recovered information (e.g., memory regions) about statements in lines of code. The ground truth for parameter fitting is obtained by coloring the vertices in the CFG following the policy explained above.

We use a statistical measure that represents how critical resources are with respect to a lock. For each feature, we estimate how characteristic (or discriminating) its value is for the critical section. We consider a term possibly critical if it appears in a critical section, and non-critical if it appears outside the critical section in non-buggy code. For example, in Table 1, if variable name *a* appears exclusively within a critical section for the buggy lock, regardless of how often it appears, then the probability that it is a critical resource is high. As a result, lines including this variable name such as Lines 3, 6, and 8 in the code on the left side of Figure 1 should be protected with higher probability, since they contain this critical feature. However, if *a* has not appeared in any critical section in the reference corpus, then this lowers the probability that these lines need to be protected. The criticality of a line is estimated from a linear combination of the criticality values calculated for the features (e.g., variable names and variable types) in the line, with weights for each features fit against the ground truth in the non-buggy code.

Once the regression model is trained on the non-buggy code, we use it to infer criticality between 0 and 1 for all lines of code in the synthesized repair candidates. Then, we compare this number with the color associated by the repair synthesis algorithm. Intuitively, a candidate patch scores higher, if its criticality score is consistent with the coloring, and lower if they disagree. Consider the two patches in Figure 1. Assume that for Lines 3, 6, and 8 a very high criticality score is calculated while the rest of the lines have lower criticality score, and that 1 and 0 are associated with black and orange vertices, respectively. Hence, after comparison, the patch on the left is scored higher than the one on the right (e.g., 0.9 versus 0.5). As a result, the patch on the left-hand side will be ranked above the one on the right-hand side.

3 REPAIR SYNTHESIS

We now delve into the technical details of Crayon's repair synthesis mechanism.

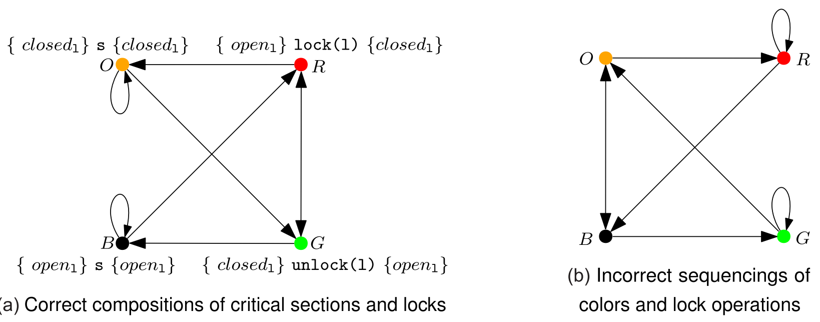

{ closed1} s {closed1}
{ open1} lock(l) {closed1}
O
R
B
G
{ open1} s {open1}
{ closed1} unlock(l) {open1}
(a) Correct compositions of critical sections and locks
O
R
B
G
(b) Incorrect sequencings of colors and lock operations

Fig. 4. Locking API correct and incorrect uses. Red (R) represents a lock call, green (G) an unlock call, orange (O) represents a third statement in a critical section, and black (B) a statement outside.

3.1 Characterizing Sequential Locking Bugs

Throughout this article, we assume a lock `l` has two states, $open_1$ and $closed_1$. The statement `lock(l)` will set the state of `l` to $closed_1$ iff `l` is in the $open_1$ state. Analogously `unlock(l)` sets the state of `l` to $open_1$ iff `l` is in the $closed_1$ state. Any other statement `s` does not modify the lock state. We can formalize this specification using Hoare triples as follows:

$$\begin{gathered}\{\, open_1 \,\}\, \texttt{lock(l)} \,\{\, closed_1 \,\}\\ \{\, closed_1 \,\}\, \texttt{unlock(l)} \,\{\, open_1 \,\}\\ \{\, closed_1 \,\}\, \texttt{s} \,\{\, closed_1 \,\}\\ \{\, open_1 \,\}\, \texttt{s} \,\{\, open_1 \,\} \quad .\end{gathered}$$

To construct all possible API misuses, we assign a vertex to each Hoare triple $R(ed), G(reen), O(range), B(lack)$ and draw an arrow between two vertices iff the precondition of the target matches the post-condition of the source (see Figure 4(a)). By construction, any arrow not in Figure 4(a) violates the composition of our Hoare triples. The complement of Figure 4(a) characterizes all possible locking API misuses (see Figure 4(b)). We say two colors are *consistent* if there is an arrow between them in Figure 4(a), we say they are *inconsistent* otherwise. Consequently, two colors are inconsistent if they are adjacent in Figure 4(b).

Let G_e be the single exit node CFG of a function f such that e is the entry vertex. We define a partial *initial coloring* C to be such that, for a vertex u:

$$C(u) = \begin{cases} R & \text{iff the vertex } u \text{ is a lock operation on the lock } \texttt{l}, \\ G & \text{iff the vertex } u \text{ is an unlock operation on the lock } \texttt{l}. \end{cases} \tag{2}$$

If the above equation does not define C for the entry vertex e, then we split the coloring in two, C^B and C^O, equal to C everywhere, but $C^B(e) = B$ and $C^O(e) = O$. Figures 3(a) and 3(b) show the two initial colorings for the example of Figure 1.

We subsequently construct the set of all total *induced colorings* as a fix point (Figure 5). Each induced coloring is constructed iteratively selecting an already colored node u and its uncolored neighbors. If u's color is B or O, then its neighbors will have the same color. If u's color is R (G), then its neighbors will be colored O (B). This color assignment is how our Hoare triplets composition is translated into a coloring problem on the CFG. In Figures 3(c)–3(e), we show the three colorings induced by Figures 3(a) and 3(b).

An induced coloring is *buggy* if an *inconsistent* edge uv exists such that the colors of its vertices are inconsistent in the sense of Figure 4(b). In Figures 3(c)–3(e) all inconsistent edges are dotted. Table 2 summarizes inconsistent colorings and the locking bugs they represent.

```
Data: $G_e, C_{\text{init}}$
// A single exit node CFG with entry vertex $e$, $G_e$ and initial coloring $C_{\text{init}}$
$C \leftarrow C_{\text{init}}$ // Start with the initial coloring
while $C$ is not total do
    let $(v, \alpha) \in C$ // Pick an already colored vertex
    // Get all outgoing neighbors of v without color
    let $N^+ \subseteq \Delta^+(v)$, $u \in N^+$ iff $C(u)$ is not defined
    // Get the next color according to Fig. 4a
    let $\beta \leftarrow \textsc{next}(\alpha)$ where $\textsc{next}(x) = x$ if $x \in \{O, B\}$ and
      $\textsc{next}(R) = O$, $\textsc{next}(G) = B$
    foreach $u \in N^+$ do $C \leftarrow C \cup \{(u, \beta)\}$ // update
return $c$
```

Fig. 5. A non-deterministic completion of an initial coloring, marking protected (orange) and unprotected (black) code.

Table 2. Classification of Inconsistent Colors (Edges in Figure 4(b)) into Lock API Misuses

Bug Type	**Witness [color trace]**
lock without unlock	$O \to R, R \to R$
unlock without lock	$B \to G, G \to G$
missing lock	$B \to O, G \to O$
missing unlock	$O \to B, R \to B$

The notation $x \to y$ means that a CFG node is colored y and its predecessor has color x.

Table 3. Summary of Repairs Considered by Our Tool

Bug Type	**Witness**	**Repair**	**Interpretation**
lock without unlock	$O \to R, R \to R$	Remove (last) R color	Remove (last) lock taking call
		Replace (last) R by G	Replace (last) lock taking call by lock release call
unlock without lock	$B \to G, G \to G$	Remove (last) G color	Remove (last) lock release call
		Replace (last) G by R	Replace (last) lock release call by lock taking call
missing lock	$B \to O$	Insert R vertex	Insert lock taking call
	$G \to O$	Replace G by R	Replace lock release call by lock taking call
missing unlock	$O \to B$	Insert G vertex	Insert lock release call
	$R \to B$	Replace R by G	Replace lock taking call by lock release call

3.2 Generating Fixes

We combine Figures 4(a) and 4(b) to derive the repairs we consider. Table 3 summarizes all considered repairs. If the error is a *lock-without-unlock*, then two repairs are possible: the last R color (lock) can be removed from the induced coloring or replaced by G (unlock) in the induced coloring. If the error is a *missing-lock* with the witness $B \to O$, then since the only possible path from B to O in Figure 4(a) is through an R vertex, we insert a new vertex colored R between the vertices colored B and O. If the witness is $G \to O$, then we replace the color G by R, or change an unlock for a lock call. A *lock-without-unlock* is symmetric to an *unlock-without-lock*, and a *missing-lock* is symmetric to a *missing-unlock*.

We define a *coloring problem* as a directed graph G, a coloring of G, a list of repairs, and a list of inconsistent edges. To start the repair process, we create a coloring problem for each induced

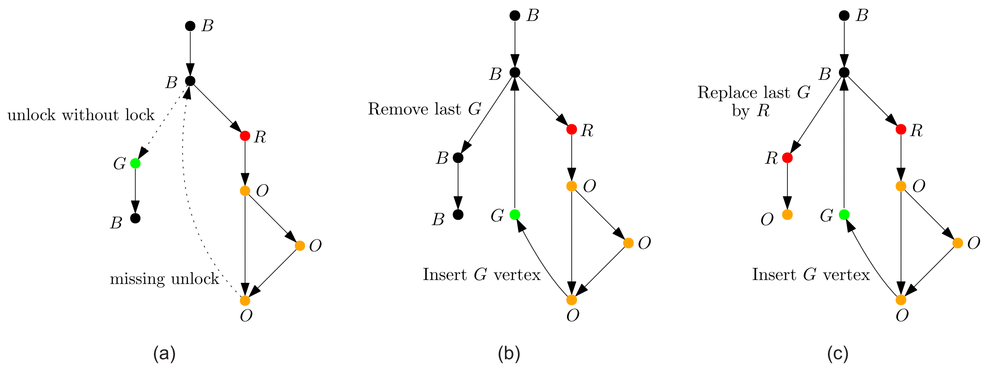


Fig. 6. Example of the repairs performed on the induced coloring in Figure 3(c). The ill-colored edges in panel (a) (dotted edges) are annotated with the corresponding bug type. Panels (b) and (c) are the two calculated repairs according to Table 3; the previously ill-colored edges are annotated with the corresponding repair.

coloring. We repair every inconsistent color pair in all coloring problems. Per each repair, we extract the initial coloring and the graph to create new coloring problems of all the new induced colorings. The list of new problems is sorted greedily with respect to the number of inconsistent edges in ascending order and only a predefined number, called *candidate threshold*, of top candidates are considered for further repair. This pruning does not compromise coverage as the initial coloring is preserved, but it helps with termination and convergence on a repaired CFG. If the problem under repair has no inconsistent pairs, then we report the repairs as a candidate patch. If the number of repairs exceeds a predefined threshold, then we give up the exploration of that particular problem.

As far as we know, the number of correct initial colorings for an arbitrary CFG, that is, initial colorings that induce a unique coloring without incorrect edges, is unknown (an open problem). However, it is possible to construct directed graphs isomorphic to CFG graphs that accept an exponential number of initial colorings all of which induce correct colorings. To construct them, we consider a base graph consisting of six nodes such that it resembles an `if-else` with entry and exit nodes. There exist four initial different colorings such that the entry and exit nodes are both black. We can paste k copies of this base graph to create a bigger (CFG) graph, which will have 4^k different correct initial colorings with the root colored black. Thus, we do not aim at exploring all colorings in Crayons.

Figure 6 shows the induced coloring in Figure 3(c) annotated with the type of bug as well as the repairs obtained according to Table 3. Interestingly, Figure 6(c) introduces an opaque error, cf. Figure 1, making the critical section too small. We minimize the possibility of showing this kind of repairs to the developer by ranking the generated repairs using a model of code criticality, discussed next.

3.3 Ranking Patch Proposals

The patch generator may produce more than one patch candidate. This motivates a mechanism to rank them, such that the approach ultimately presents a small number of most likely patches to a developer. If patch generation terminates successfully, then the patched candidate CFG contains no ill-colored edges, and its coloring shows which nodes are assumed to be critical by the patch candidate. To avoid introducing opaque errors, we aim to assess the correctness of this assumption, and use this assessment to rank the patches.

Assume that we could obtain an oracle that for each node in the CFG can estimate criticality as a real number between 0 and 1. We will explain how to construct such an oracle in detail in

Section 4. Given such an oracle, we reduce the ranking problem for patches to computing the agreement (via distance) between the patch's conception of the critical section and the oracle's. We map the color c_i of a node s_i to $x_i = 1$ if it is critical, and to $x_i = 0$ if it is non-critical. Let $w_i \in [0, 1]$ be the criticality weight assigned by the oracle to s_i. We ignore the lock and unlock nodes and compute the absolute value of the difference between the patch coloring and the oracle:

$$r = \sum_i |x_i - w_i|. \tag{3}$$

We rank higher the patches with a smaller error r.

4 LEARNING AND PREDICTING CRITICAL SECTIONS

We now detail the design of the **Crayons**' criticality model for lines of code, that is, whether they should be protected by a lock. We use this information to rank the produced patches (Section 3.3).

4.1 Learning Metrics

The criticality of a line (and consequently of each CFG vertex) is not explicit in C. However, we intuit that, in a large project, competent programmers are likely to have written correct code using the same lock elsewhere. We therefore learn an approximate model of criticality from functions that use the lock involved in our bug, but that do not contain tracked bugs themselves. This requires a metric that can numerically describe which program terms are typically used under the protection of a lock, and not used without it. This is reminiscent of a similar idea used in document summarization to identify natural language terms specific to a summarized document. We thus adapt **Term Frequency–Inverse Document Frequency (TF-IDF)**, a classic metric for identifying the most characteristic terms from a document, to our domain. In natural language processing, TF-IDF measures the importance and relevancy of a word to a document with respect to a collection of other documents, a reference. The TF-IDF of a word is directly proportional to the number of its occurrences in the document of interest, and inversely proportional to the number of documents in the reference collection that contain the word. In our application, a resource is critical if it is specific to the critical sections in the code base, and it is not critical if it appears in the collection of other code, outside known critical sections. The feature values (respectively, effects and regions, names, and types) are the terms, and we use TF-IDF to estimate how discriminating each feature is for the critical section; this information in turn is used to train a model to estimate line criticality.

Generating a training corpus. Given a buggy function and a lock l participating in the bug, we first obtain a corpus of presumably correct intra-procedural CFGs from the system under repair. By *presumably correct*, we mean that the static analyzer does not flag lock usage on this code. We use other functions in the same file or in files nearby in the project's source tree (for the Kernel, we choose files in the same subsystem). The CFGs are decorated with the static abstraction of resources and colored as described in Section 2. Every vertex in a labeled CFG contains a reference to the original line, the computational effects and memory regions involved (aliasing information), as well as types and names of the variables occurring in the line. The entire training corpus is constructed automatically during repair, given the program code base and the involved lock l. It has to be done in the same run of the static analyzer that found the bug, so that the used abstract memory regions are the same in the CFG, and the region assigned to l during bug finding can be used to detect when the incriminated lock is manipulated in a CFG.

Terms (features). We use the following features as terms for TF-IDF. Each feature describes an aspect of statements/resources in a line s_i in the code.

Regions $R(s_i)$. All memory regions read and modified by a line. The set of participating regions abstractly characterizes the resources affected by a line (possibly protected by the lock). Table 1 shows regions embedded into effects, but we treat them as a separate feature during learning. For example, *reg23*, *reg153* in Lines 1 and 2 are regions.

Effects & Regions $ER(s_i)$. All effects produced in the line of code, parameterized by the affected regions. Effects provide an abstract characterization of the semantics of the line. Shown in the third column of Table 1, right after the source code.

Variable Names $VN(s_i)$. Names that appear in a line. The feature provides a different characterization of resources than regions. Names are oblivious to aliasing but carry meaning in the text of the name, which is absent from abstract regions identifiers. Shown in the rightmost column of Table 1, before each slash.

Variable Types $VT(s_i)$. Types of variables that appear in a line. Type names carry information about the resources, too. For instance, modifications of a linked list are likely to be protected to get thread safety. Shown after each slash in Table 1.

Estimating criticality. We first explain how to calculate TF-IDF for a single value of a single feature. The goal is to obtain a metric that increases proportionally with the number of times that the value appears in critical sections. Additionally, it is discounted by the number of times that it appears outside the critical section in the reference collection. Let critical(t), respectively, non-critical(t), represent the number of occurrences of the feature value t (a token) appearing in a line labeled as critical, respectively, non-critical, in the corpus. Let N be the total number of lines in the corpus and n is the number of lines in which t appears in. Then,

$$\text{TF-IDF}(t) = \frac{\text{critical}(t)}{\text{critical}(t) + a \cdot \text{non-critical}(t)} \cdot \frac{N}{n}, \tag{4}$$

where a is a parameter controlling the importance of appearance in non-critical vertices of CFGs. When a is large, even a small number of occurrences of the feature value outside critical vertices, makes the score low. In this article, we use $a = 1,\ 10,\ 100,\ 1,000$, but exploring other values is potentially interesting in the future.

After computing the TF-IDF values for each valuation of each feature separately in each line, we calculate the weight (criticality) of the lines. The criticality of a line depends on the criticality of the valuations of regions, effects, variable names and types that appear in the line. To obtain a single value representing the criticality of the entire line with respect to a single feature, say a variable name, we either (i) sum the criticality values of all variable names in the line (an additive superposition) or (ii) take the maximum value of criticality among the variable names appearing in the line (winner-takes-all). The intuition behind considering the sum of values is that if there are several variable names that have a positive criticality this means that the line has a higher probability to be critical. The intuition behind considering the maximum value is that the criticality of a line is directly proportional to the most critical variable name (and other features) in that line. This criticality value is calculated for all four features in the same way. After calculating these four values that represent the criticality for the four features, we combine these values to obtain a weight for each line. But the contribution and influence of each feature in the weight of a line is not obvious and should be estimated using a learning method such as linear regression.

The following equation shows the calculation of criticality of a line as a linear combination of the TF-IDF value of the features present in the line. The criticality of line s_i, denoted w_i, is estimated as

$$w_i = \alpha \bigoplus_{t \in \mathrm{R}(s_i)} \text{TF-IDF}(t) + \beta \bigoplus_{t \in \mathrm{ER}(s_i)} \text{TF-IDF}(t) + \gamma \bigoplus_{t \in \mathrm{VN}(s_i)} \text{TF-IDF}(t) + \omega \bigoplus_{t \in \mathrm{VT}(s_i)} \text{TF-IDF}(t), \tag{5}$$

where we use two operations: $\oplus = \{+, \max\}$. Intuitively, using + takes into account the combined importance (TF-IDF) of terms and using max means taking into account only the strongest terms in each line—so a line is critical as soon as a single resource in a line is critical. In the above formula, α, β, γ, and ω are coefficients that determine the contribution of each feature in the weight calculated for each line. These coefficients are tuned using logistic linear regression. Note that the TF-IDF of feature values that do not appear in any critical section is zero (cf. Equation (4)). Hence, these values do not contribute to the weight of the line. Logistic regression guarantees that $w_i \in [0; 1]$.

5 EXPERIMENTAL EVALUATION

We implemented the above repair synthesis, criticality learning, and proposal ranking in an APR tool **Crayons**. We used EBA [16] as the static analyzer. The repair synthesis is implemented in OCaml. The maximum repair threshold is set to 6, and the candidate threshold to 3, that is, the tool will suggest repairs with at most six edits and in each round of repair, it will keep the three coloring problems with the fewest inconsistently colored edges. Given the nature of the bugs we are targeting, six edits per repair is a good heuristic upper bound. A six-edit patch means that the developer has forgotten to release the lock on at most six distinct error paths. In practice most bug-fixing patches are short, fewer than five edits [17]. Keeping the top three coloring problems allow us to explore our search space without committing to a particular path. We used parallel map functions to calculate patch candidates when possible.

We use the regression and scaling algorithms from sci-kit (https://scikit-learn.org/) to implement the criticality estimation component, in Python. As regression tends to perform better when numerical input variables are normalized, we scale the input term values to zero mean and unit variance, before feeding them into the regression algorithm. Since the data for training is imbalanced (there are many more non-critical than the critical lines), we use a variant of regression with class weighting. The model is fined tuned for best AUC-ROC [18], which measures how well the model distinguishes between critical and non-critical lines. We pick the best performing hyper-parameter values (max iterations, tolerance for stopping criteria and choice of solver for the LogisticRegression model in the library) on the training corpus, before we use it to predict line criticality.

We used ten intra-procedural double-lock bugs from the work of Abal [16] to guide the design process. Once we were satisfied with the results, we selected 100 locking bugs from the Linux Kernel git history to evaluate its performance (Section 5.1). The key research question of interest are:

RQ1. To what extent does **Crayons** suggest and prioritize plausible patches similar to those proposed by developers (Section 5.1)?

We complemented the above main research question, with two additional ones, aimed at each of **Crayons**' main components:

RQ2. To what extent does **Crayons** synthesize high quality repair proposals (Section 5.2)?
RQ3. How effective is the learnt oracle at discriminating critical and non-critical code (Section 5.3)?

5.1 RQ1: Quality of the Patch Synthesis and Ranking

This first research question evaluates **Crayons** as a whole, seeking to establish how well **Crayons** construct and rank patch candidates for real-world bugs in the Linux Kernel.

Experiment design. To empirically validate **Crayons** as a whole, we selected historical bugs in the Linux Kernel git history, ran them through an implementation of our approach, and compared the generated patches and their ranking against the historical ground truth.

Table 4. Statistics of Commits Discarded from Consideration in the Evaluation Data Set Design Process

Discard class		**Count**
Bugs outside the targeted group	Duplicated commits	1
	Fix interfunctional bug	2
	Fix interaction between two locking APIs	5
	API misuse was about what was protected	4
	Locking API not supported by EBA	7
	EBA crashes	10
Limitations of implementation of **Crayons**	Locking API not supported by **Crayons**	4
	Crayons failed to construct a single root CFG	3
	Crayons failed to construct the initial coloring	1

Data selection. Locking is ubiquitous in operating systems, particularly in the Linux Kernel (e.g., the string `lock` appears in more than 124,000 git commits). Selecting a test set would be difficult if we approached it by manually reviewing 124,000 commits. Moreover, we are unaware of any particular name for the handled bugs within the Linux Kernel community that could be used as a more narrow search term. Instead, we decided to select our test set from the commits authored by Dan Carpenter, an independent (from us) expert contributor known for fixing bugs of this kind. He is also the original author of the Linux-specific bug finding tool Smatch.[2]

We started by identifying 935 commits authored by Carpenter and including the word `lock` within the header, summary, or the patch. We preprocessed all the files modified by these commits using a "mid-life" version of GCC compiler: GCC 6.2.1 on a Fedora Workstation 25. Files from 711 commits were preprocessed correctly. We manually selected random commits out of these until we collected 100 bugs. The commits were not selected by any particular ordering, not even git timestamps, and the filtering criterion was that the bugs involved lock manipulation. Table 4 summarizes the reasons used to discard commits during sampling. Bugs were chiefly discarded for reasons outside control of **Crayons** (29), but some were also removed because of limitations in our prototype, which could not complete execution (8). In three cases, the parsing component, on which **Crayons** and EBA rely, did not create a single rooted CFG. **Crayons** contains checks in its translation from CFG to directed graph such that when the constructed graph contains more than one root it discards the function. In one other case, this interaction with the parser component is responsible for **Crayons** not being able to construct an initial coloring, in this case several control structures were removed and an artificial double lock bug was introduced using statements originally present in the function; once this modifications were made, **Crayons** was able to construct an initial coloring.

Among the 100 commits selected for testing, we chose to include 19 that patch locking API-misuse errors using no-busy wait locking (e.g., `mutex_lock_interruptible`), that is, the calling thread gets a return code indicating whether it holds the lock or not instead of spinning until the lock is acquired. These kind of APIs do not conform precisely with our specification described in Section 3, but they are *close enough*; hence, we included them to study the precision of the suggested patches modulo the predicted imprecision. We call this set of bugs *non-blocking API bugs*.

In total our test set is composed of 76 missing lock/unlock bugs, 12 double lock bugs and 12 unlock without lock bugs. Table 5 shows further statistics on the selected commits.

Execution and results. **Crayons** were run for each of the 100 bugs. We divided the synthesized patches into the following categories by closeness to the developer patch:

[2]https://repo.or.cz/w/smatch.git.

Table 5. Descriptive Statistics for Commits Included in the Evaluation Data Set

Measurement	#LOC	|CFG|	#CNodes
Mean	67.8	61.2	14.8
Median	53	49	11
Mode	53	23	6
Standard Deviation	57.43	47.73	12.94
Min	12	10	1
Max	414	265	59
Total	6,509	5,876	1,425

The column labeled **#LOC** are the number of lines of code in the buggy files. The |**CFG**| column show the test suite commits CFGs size statistics. Finally, the **#CNodes** column shows the statistics of the number of control nodes in the selected commits.

- *Developer patch.* The lock API call changes suggested by **Crayons** are the same as the ones the developer applied.
- *Equivalent patch.* The edits suggested by **Crayons** in lock API calls are *not* the same as the developer applied, but the colorings of both solutions are the same.
- *Close patches for non-blocking APIs.* The edits suggested by **Crayons** include a predicted lock release instruction that arises from imperfect dealing with non-blocking APIs (e.g., `mutex_lock_interruptible`). If these predicted instruction is removed, then the remaining edits give the Developer or equivalent.
- *Over-protects.* The colorings of both solutions are not the same, but the critical section of the developer's solution is contained within the critical section suggested by **Crayons**.
- *Possibly patched.* The original C code does not manipulate locks directly; they are introduced as part of macro expansion or inlining. Hence, we cannot be sure of **Crayons** correctness.
- *Incorrect.* The colorings of both solutions are not the same and the critical section of the developer's solution overlaps with the critical section suggested by **Crayons**.

We consider the patches in the first three groups as successful results for **Crayons**. Figure 7 shows how the patch proposals in the first three categories were ranked by **Crayons**; the lower the rank the higher on the position of suggestions presented to the developer the patch is. Hence, the higher the left-hand side bars in Figure 7 the better.

The median of the ranks for patches that match the developer's version as well as patches that we classified as equivalent to the human-provided solution is 1, that is the majority of these kinds of patches were suggested first. For the non-blocking API bugs, the median rank of the corresponding Close patches for non-blocking APIs suggestions was 3. (Recall that this bugs are not strictly within the scope of the tool, but nevertheless the produced results are reasonable.) For 15 evaluation cases no successful patch was generated—these cases are not included in Figure 7.

It is possible that the specific characteristics of a set of repairs leads the Developer/Equivalent patch to get the same rank as other suggestions. Currently, we do not perform any tie-breaking.

We consider a **Crayons** generated patch to be unsuccessful when it is within the category Over-protects, Possibly Patched, or Incorrect. The commit `79d753209245` (double unlock) is an example of the former. **Crayons** proposes a critical section three statements bigger than the developer solution. In addition, our learning infrastructure correlated a variable representing a file system inode with taking the buggy lock. Thus the model wrongly learns that each time the variable is read or written, the lock must be held. Since the function opens with assigning the variable in question,

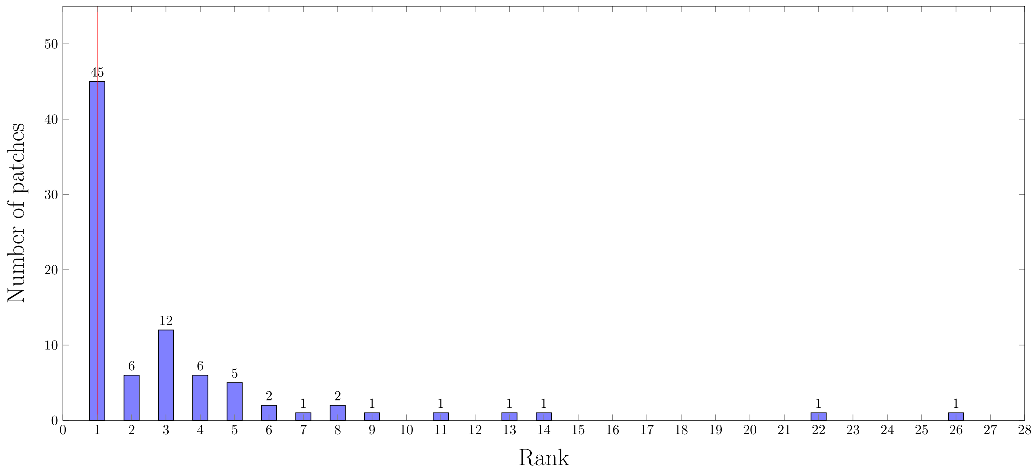


Fig. 7. Ranking results, the red line shows the median: for 51 of 100 evaluation bugs the highest or second-highest ranked patch is identical or equivalent to the developer patch.

patches that assumed the function to be called with the lock held (orange entry vertex) were ranked higher than the patches that did not, resulting in mis-ranking of most of the proposals.

In response to RQ1, we note that for 63% of evaluation cases, a successful repair proposal (identical or similar to the one already merged into the kernel) has been synthesized and ranked within the top three and 74% within the top five. It appears, thus, that **Crayons** is able to suggest and prioritize plausible patches well, which means that a tool like **Crayons** could reasonably be used as part of interactive patching interface, even if only three patch proposals were to be presented to the user.

5.2 RQ2: Feasibility of Static Template-based Patch Generation

We now turn our attention only to the synthesis part of **Crayons**. In particular, we seek to evaluate the quality of the repair proposals generated.

Experiment design. We want to assess whether **Crayons**' patch generation method presented in Section 3 scales for real files and whether it is able to produce patches that human developers would use. We focus purely on the feasibility of producing a good patch, ignoring ranking and ordering patches. This part of the experiments is performed on C files (in the git history of the Linux Kernel) with known locking API bugs that have been fixed by kernel developers (using the same dataset as for RQ1).

Each C file is first pre-processed, and the patch generator is given the function name and the lock details for the bug in question. After the patch candidates have been constructed, we check if they contain the human solution from the git history (our ground truth); in the case that it is not found, we look for an *equivalent* or *the closest* one. We say a repair is *equivalent* to the developer's patch when the lock behavior is the same in both solutions. In the case that the patches suggested by our tool are neither, we look for the one with the closest locking characteristics to the developer's solution; we elaborate on our definition of "close" in the results, below. Because our analysis is path insensitive, **Crayons** may introduce more modifications than the developer, e.g., when a function is meant to terminate with the lock acquired in some branches but released in others **Crayons** will commit to a solution with the same lock state on all branches.

Data selection. We used the same data set as for RQ1.

Table 6. Classification of the Patches Generated by **Crayons** and the Number of Patches in Each Category

Class name	Description	Patch count
Developer patch	The proposed fix matches the developer's solution	28
Equivalent patch	The induced colorings of the developer solution matches the ones from the suggested patch	38
Close patches for non-blocking APIs	The proposed repair includes manipulations arising from analysis imprecisions	19
Over-protecting	The critical section of the suggested patch is bigger than the human-made solution	8
Possibly patched	We cannot be sure of the correctness of this patch as the original (non-preprocessed) code does not manipulate locks directly	3
Incorrect	The proposed solution induces an inter-functional bug	4

For each bug the best result (the most similar to human patch) synthesized by **Crayons** is classified.

Execution and results. The experiments were performed on a ThinkPad x250 with a 2.3 GHz dual core i5-5300U CPU and 8 GB RAM running Fedora 32 (Workstation). The average time for generating patches per file is 23.14 s. For most bugs, we produce seven repair candidates but the distribution is heavily skewed to the right with the median number of candidates per bug being 12, and the mean 20.86 ± 25.81 (note the high variance). The furthest outlier received 143 repair candidates.

To understand the characteristics of the synthesized patches, we classify them into six broad categories, attempting to describe the *closeness* of the proposed repair to the one provided by a developer. The classes range from identical with the human-made patch to introducing an inter-functional locking bug. Table 6 list all the categories, their descriptions (for quick reference), and patch counts.

Equivalent patches arise when the developer introduced a `goto` label with the fix (e.g., unlock call) and then fixed various paths with a jump to the newly introduced label. In other cases, the label was already present, but not all execution paths would reach it, then the developer introduced the missing `goto` jumps; in all these cases **Crayons** proposes introducing the correct locking API call at each branch. *Close patches for non-blocking APIs* suggestions are the proposals that **Crayons** generated for the *non-blocking API* bugs set (Section 5.3). These bugs use a non-busy wait locking API, such API is not precisely modeled by our specification, because when a lock function is called (e.g., `mutex_lock_interruptible`) a status code is returned indicating whether the lock was successfully acquired or not instead of performing a busy wait. **Crayons** assume the lock acquiring function performs a busy wait, so the code that actually checks for lock acquisition is marked as protected, hence when repairing such bugs an unlock after this code is inserted by **Crayons**, however, if this extra unlock is removed then the patch is equivalent or identical to the human-made solution. We discuss how to address this limitation in Section 6.1. Patches that *over protect* are primarily due to **Crayons** introducing an API call as the last statement before a control flow join point while the developer introduced it before. In a few cases (*Possibly patched*), we cannot be sure if the patch is correct as the original code does not manipulate locks directly, but they are introduced as part of macro expansions. Finally, *incorrect patches* are those calculated by **Crayons** such that if applied, they would introduce an inter-functional bug; it could be an inter-functional API misuse or a critical section breaking bug, that is, the lock should not be released on some control paths and **Crayons** introduces unlocks on those.

The Equivalent patch class is further divided into five classes depending on what control flow mechanism the developer employed in the solution. The most common repair was the introduction

Table 7. Equivalent and Close Patches for Non-blocking APIs Sub-classes and their Count

Repair Class	**Repair sub-class**	**Count**
Equivalent Patch	Equivalent	1
	Equivalent (`goto jmp`)	18
	Equivalent (`goto label + jmp`)	16
	Equivalent (`break`)	3
Close patches for non-blocking APIs	Equivalent	9
	Equivalent (`break`)	1
	Equivalent (`goto label + jmp`)	7
	Equivalent (`goto jmp`)	1
	Developer	1

of `goto` jumps to existing labels, followed by repairs that introduce a `goto` label and a jump to it. The Close patches for non-blocking APIs suggestions are subdivided similarly, except that they include an an extra unlock due to the analysis imprecision of our tool when handling non-busy wait locking APIs. Table 7 lists the Equivalent and Close patches for non-blocking APIs sub-classes and their sizes.

Over-protecting repairs emerge when the developer, using criticality knowledge, rearranges the lock acquisition or release (as in commit 79d7532 in Section 5.3) to make the critical section smaller but overlapping with the original one. In contrast **Crayons** suggest a conservative repair inserting API calls at join points.

In four cases (4%) **Crayons** proposed patches that, if adopted, would introduce inter-functional bugs. These bugs are of two types, locking API misuse and inter-functional critical section breakage. Inter-functional API misuses are introduced in one case (1%) when the proposed repair suggests to insert an unlock after a function call, however the function being called reacquires the buggy lock. However, critical section breakage errors are introduced when a function should return with the lock held in some paths and with the lock released in others. **Crayons** uses a single virtual sink node to which all exit nodes are redirected and detects this lock state difference as a coloring problem at the sink. To fix it, **Crayons** transforms the graph and the coloring so that all paths leave the control with the lock held or released but not both. In Section 6.1, we discuss how we can address this limitation.

5.3 RQ3: The Quality of the Learnt Oracle

Experiment design. We want to evaluate how well the learned models predict criticality. To answer RQ3, we simulate training and criticality evaluation on a set of labeled CFGs, with types, names, regions, and effects, generated from 1,000 non-buggy files in the Linux Kernel source tree for release 5.6.19.

We split the generated set into the training evaluation sets, ensuring that there is no information flow between the two to avoid overfitting. We only consider files in which at least one lock is used. For each identified lock, we select for evaluation one function f that manipulates or uses it. The CFGs from the remaining functions in the same file then serve as the training set F. If there is no function in the C file that uses the same lock, then we explore files in the same subsystem to find a function that uses the same lock, i.e., a lock with the same name and type. We grow the training set with CFGs from other files that belong to the same subsystem in the Linux Kernel.

We calculate the TF-IDF for features in the body of the evaluation function f and in all code in the training set F. Tokens are extracted with respect to each of the four features listed in Section 4.1,

Table 8. The Average AUC-ROC Calculated Over 100 Iterations for Seven Subsystems, under Different Choices of the TF-IDF Parameter *a*, and Two Modes of Criticality Superposition (Additive Versus Winner-takes-all)

		mode = sum (additive)				**mode** = max (winner-takes-all)			
Subsystem	**#LOC**	**a** = 1	**a** = 10	**a** = 100	**a** = 1000	**a** = 1	**a** = 10	**a** = 100	**a** = 1000
drivers	245928	0.75	0.75	0.76	0.76	0.76	0.77	0.77	0.77
fs	14555	0.72	0.72	0.72	0.72	0.74	0.74	0.74	0.74
sound	27948	0.75	0.76	0.76	0.76	0.75	0.78	0.78	0.78
kernel	6112	0.71	0.71	0.71	0.71	0.72	0.72	0.72	0.72
net	10816	0.76	0.76	0.76	0.76	0.77	0.78	0.78	0.78
lib	1024	0.75	0.74	0.75	0.74	0.75	0.74	0.75	0.74
arch	6195	0.73	0.73	0.77	0.78	0.73	0.73	0.80	0.80

#LOC is the total number of lines in the bodies of functions in each subsystem after preprocessing; header files are not counted except if containing functions.

and we calculate TF-IDF per line, summing over or taking max of multiple feature values like in Equation (5). This is congruent with how the learning component is used for ranking in our static repair method.

Performance evaluation. **Area Under The Curve (AUC) Receiver Operating Characteristics (ROC)** (in short **AUC-ROC**) is an established evaluation metric for checking classification model performance. The metric indicates how well a model distinguishes between classes. The range of values for this metric is [0,1]; higher is better. The value of 0.5 indicates that the model has no class separation capacity and values of 0 and 1, respectively, represent perfect misclassification and perfect classification for a classifier. We use this metric to evaluate the performance of the models that are learned for each sub-system as explained above. It allows us to avoid selecting a specific classification threshold, which is consistent with the ranking method described in Section 3.3—our patch prioritization does not use a threshold either, but rather uses the real number from the classifier directly.

Data selection. The search in the kernel codebase (version 5.6.19) gives 3,352 candidate files that have a lock acquired in at least one function (restricted to the files that are processed by EBA within a 60-min time limit and within 24 h total time limit for seven subsystems). We used 1,000 random files, of the search results, from seven subsystems: Drivers, Net, Fs, Sound, Kernel, Lib, and Arch. The labeled CFGs for these files are generated using EBA (Section 4), separately per each lock in the file. The average size of the files included in the experiments is 1.5 KB. We hold out 20% of the set for evaluation. We run the experiments in 100 iterations for each subsystem and report the average AUC-ROC. The average time for each iteration is 2.59 s.

Execution and results. The calculation of TF-IDF for all files is performed on a cluster of Linux machines with eighty 2.20 GHz CPU nodes. We parallelized this calculation per file. Depending on the size of the input a single file is processed in few minutes, but it can take up to hours (due to the sheer number of features involved). There are plenty of opportunities to optimize this part that we have not explored. The actual regression and evaluation are relatively fast (they take ca. 9 minutes). We do not report the precise times, as we used a time shared machine with other users for this experiment.

Figure 8 shows the obtained AUC-ROC curve for the seven kernel subsystems and Table 8 details the values obtained. The performance of the best tuned classifiers varies between 0.71–0.80 (the fraction of correctly ordered positive and negative cases). This is consistent with the overall

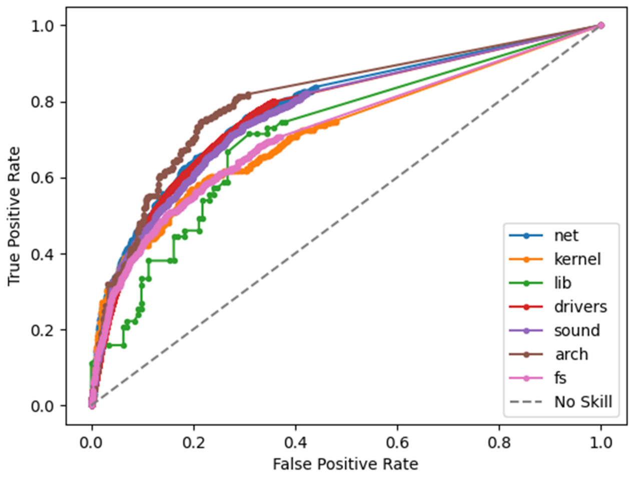


Fig. 8. ROC curve calculated for data from seven sub-systems in the Linux Kernel (with a = 1,000 and mode = max).

performance of **Crayons** discussed in Section 5.1, so it is likely responsibly for the quality of the obtained ranking.

We were also interested in how the different features contribute to the quality of the classifier, and, in particular, whether there is any benefit of using the static analyzer's memory and effect abstractions, over the syntactic information in the names of variables and types. The regression mechanism captures this information in the fitted coefficients of the linear Equation (5). The average weight value calculated for the four features (assuming a = 1,000 and mode = max, which are the values used in patch ranking) is as follows:

regions: −0.1, effects (with regions): 1.0, variable type: 0.2, variable name: 0.3,

showing that the effects with regions embedded are the feature that has the most influence on correct prediction, and it is considerably more informative than the syntactic features. Indeed, the influence of the other features is minimal.

To conclude, the results of the experiments show that we can obtain models constructed by the learning method, with performance 0.71–0.80, which can effectively predict criticality of lines, largely thanks to the semantic information about effects and regions produced by the static analyzer.

6 DISCUSSION

6.1 Limitations of Crayons APR

We want to briefly reflect on the limitations of our APR method. The Close patches for non-blocking APIs patch problem (Section 5.2), arising for no-busy-wait locks, is not an inherent issue with the method, but more of an implementation issue. To handle these patches first class, the semantic part of a tool implementing our approach must be able to distinguish when such an API is used and create an initial coloring that only assigns orange to the first statement inside the critical section. According to our test cases, it seems a no busy wait locking API, like `mutex_lock_interruptible`, is used within the condition of an `if` as follows: `if(mutex_lock_interruptible(l)) return v; pstmt1;` such that `v` is a default error code and `pstmt1` is the first protected statement. Hence, instead of assigning *red* to `mutex_lock_interruptible(l)`, the initial coloring should only assign *orange* to `pstmt1`.

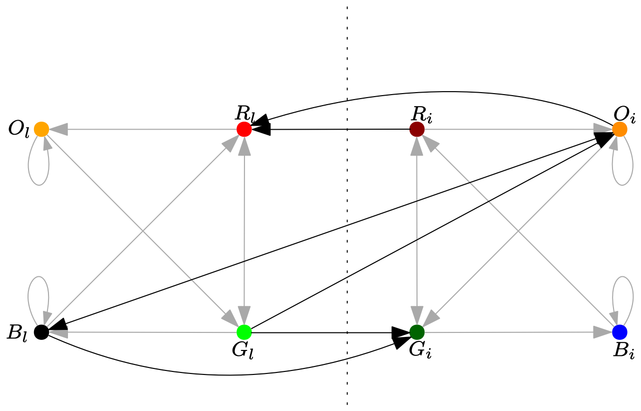


Fig. 9. Lock order model for two locks i and l when lock l must be taken after i has been acquired and released before i. The left-hand side of the dotted line models the correct API use for the lock l while the right-hand side models it for the lock i. The crossing black arrows define the acquisition order.

The Over-protecting class (Section 5.1) exposes a limitation of **Crayons**' understanding of critical sections. In the patch generator, we determine a critical section by the placement of the API calls in the initial coloring. Then repair transformations are performed at the *latest valid point*, e.g., at control flow join points. Patches in this class propose a larger than necessary critical section, and the tool is not attempting to synthesize smaller proposals. A compelling alternative would be to integrate the criticality model and the synthesizer more tightly, so that a more refined initial coloring is produced, e.g., assigning *black* to statements that *surely* are not critical, and *orange* to statements that *surely* should be protected. This would let our synthesis machinery construct candidates that respect those constraints. Exploring these possibilities is left to the future work.

Presently **Crayons** do not handle functions where the state of the lock is different at different exit points. To address this issue, at some extent, we could classify the exit points according to their lock state and the available semantic information (e.g., successful or failed execution) and make sure the classes do not intersect (i.e., all successful executions leave the function holding the lock and only the failed executions release the lock).

Our repairs may introduce deadlocks between threads, if they change the order of operation between distinct locks. Our approach could be modified to handle lock order as well. Figure 9 shows how the use of two locks i and l could be expressed when the acquisition of lock l must happen after lock's i. The construction of this model can be automatized once the lock order is defined, which we would like to add as an additional learning objective. However, as we generate human-made patches for many cases now, this is unlikely to improve results practically (while reasoning about concurrent deadlocks statically requires an entirely different machinery, e.g., Reference [19]).

Finally, **Crayons** does not serialize patches to the diff format applicable to the source files in C. This is because patches need to be applied to code before preprocessing, and reversing the preprocessing is a difficult problem, which requires dedicated research effort.

6.2 Threats to Validity of the Evaluation Experiments

Internal Validity. In the experiments for RQ3, we generate the corpus of labeled CFGs without knowing that the subject source files do not contain double-lock bugs. This is a reasonable assumption. Bugs are rare and an odd file containing a bug should not influence the results much. In fact, this setup reflects the way the oracle is later used for ranking: Since EBA is incomplete, we cannot be sure that when ranking bug candidates we will not be learning from buggy functions as well.

EBA generates CFGs including line numbers, the set of effects, regions, variable names and types for each line, for all lines except for the branching points, where only the information about the regions and effects is presented. This reduces the accuracy in learning the criticality model in comparison to having the full range of features in all lines. Similarly, in some places the name and type of a `struct` used in the line is included without the name and the type of the accessed member. These problems do not invalidate the results. Resolving them would likely lead to a better ranking of the patches.

External Validity. The development of **Crayons** has been guided by the 10 double-lock bugs in the Linux Kernel described by Abal et al. [16]. To avoid bias in evaluation, the experiments of Section 5 have been executed on 100 different cases of lock manipulation bugs fixed by Dan Carpenter in the Linux Kernel history. There is a risk of the tool performing particularly well on Carpenter's bugs by accident. However, constructing suitable sets of confirmed bugs for a system of complexity and scale of the Linux Kernel is extremely difficult and requires expert knowledge. Thus to avoid construct validity issues, we rely on known historical bugs that have been fixed by an expert. The design of the tool has followed broad semantic principles, not the work of this expert, though. Furthermore, we used a rather generic metric (sum of errors) for creating the ranking to avoid biasing the design towards the evaluation set. At the same time, being able to evaluate on real examples gives us confidence that the method is scalable and sophisticated enough for real problems.

6.3 Beyond Locking API Misuses

An interesting question is: *To what extent can **Crayons** be generalized/adapted to automatically suggest patches for misuses of other APIs?* To address this question, we focus on **Crayons**' main components.

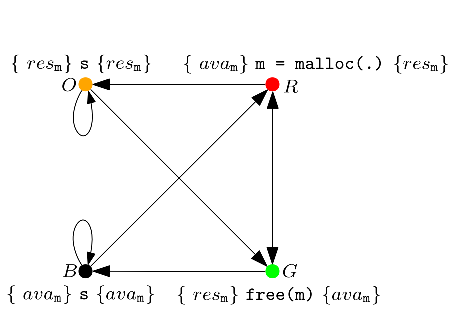
The labels on Figure 4(a) reflect how the pre-conditions and post-conditions of our Hoare triples relate. This means that we can replace them by any other Hoare triples as long as the graph depicts their relation, that is, the post-condition of any arrow source matches the pre-condition of its head. The graph in Figure 4(a) neither imposes any restriction to the type or form of the statement of the Hore triple. It can be a function call in the way we use it in the present article or a block of statements; it can even be expressed not as a C statement but as an intermediate representation statement. This allows more complex statements to be considered (e.g., as in bug `c202e2ebe1dc`). The misuse graph in Figure 4(b) is defined in terms of the graph in Figure 4(a) and the repairs in Table 3 are defined by considering the colors of the nodes in Figure 4. It is only when we interpret the colors of the nodes back to the Hoare triplets' functions (`lock`, `unlock`) that we give the repairs a meaning in terms of a locking API.

Crayons' learning component is also driven by labeling. To learn other APIs the labeling needs to be consistent with the one used on the synthesis part. In learning, we use Equation (4) as our criticality heuristic and Equation (5) as a measure for our line labeling. The general from of the equation for calculating the weight of lines can be considered as

$$w_i(\mathrm{h}) = \alpha \bigoplus_{t \in \mathrm{R}(s_i)} \mathrm{h}(t) + \beta \bigoplus_{t \in \mathrm{ER}(s_i)} \mathrm{h}(t) + \gamma \bigoplus_{t \in \mathrm{VN}(s_i)} \mathrm{h}(t) + \omega \bigoplus_{t \in \mathrm{VT}(s_i)} \mathrm{h}(t). \tag{6}$$

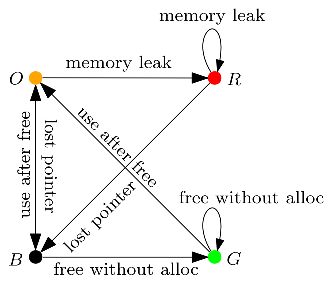
Equation (6) suggests that to generalize **Crayons**' learning component only designing a new learning heuristic (h) is required. For example, in case of a K-function APIs, this heuristic can be a weight obtained using a multilabel classifier. Furthermore, this observation also suggest a future work research question: *to what extent **Crayons** learning component can be API agnostic?*

We hypothesize that if the tracking characteristic (criticality, memory management) is expressible in the language's grammar (e.g., variable assignment and use in the case of memory



{ res_m } s { res_m }
{ ava_m } m = malloc(.) { res_m }
O
R
B
G
{ ava_m } s { ava_m }
{ res_m } free(m) { ava_m }

(a) Correct compositions of memory locations and API.



(b) Incorrect sequencings of colors and the corresponding misuses.

Fig. 10. Memory management API correct and incorrect uses.

management) then learning can be reduced to a form of liveness analysis. However, when the tracking characteristic is not expressible in the language's grammar (e.g., which statements must be protected by a given lock), then our learning proposal is applicable out-of-the-box considering the modifications we discussed earlier.

Figure 10(a) depicts how a memory management API can be modeled by relabeling the nodes in Figure 4(a). This model assumes an *infinite* amount of memory, that is, `malloc` never returns `NULL`; modeling `NULL` return is equivalent to modeling non-busy-wait locks. Figure 10(b) shows what memory management API misuses are derived from Figure 10(a). The main technical difference between a locking API and a memory management API is that critical sections are significantly smaller compared to *live* sections, e.g., a memory location reservation can live through a series of function calls before its release. This difference implies that to have a more precise view of the program, using a strong alias analysis is required as well as a constraint solver. The former is needed, because the likelihood of introducing aliases (e.g., function pointers) increases with the size of the code. The latter is needed to keep track of functions that return different states (return codes) in which the tracked property (memory management) is not consistent among them. However, we believe it can be possible for **Crayons** to tackle simpler intra-functional memory bugs, e.g., `1b9dadba5022`.

6.4 A Static APR Architecture

Our previous discussion suggests an architecture for static APR that we depict in Figure 11. It is composed of five main modules: Static Analysis, Learning Oracle, Fault Localization, Patch Generation, and Patch Ranking. The *Static Analysis* module is responsible for collecting and providing static analysis(es) results. The *Learning Oracle* module generalizes the information provided by the Static Analysis module; if the Static Analysis module is precise enough to properly rank suggestions for the repairs under consideration, then the Learning Oracle can be reduced to an identity module.

The *Fault Localization* module is responsible for bug finding while the *Patch Generation* module generates patch candidates; both obtain their program information from the Static Analysis module. Finally, the Patch Ranking module, aided by the Learning Oracle, ranks the patches generated by the Patch Generation module to be presented to the user.

Figure 12 shows how we realized our architecture proposal in **Crayons**. We use the EBA Bug Finder (Section 2.1) as the Static Analysis and Fault Localization modules. Our coloring algorithm (Section 3.2) conforms the Patch Generation module. Our technique for learning and predicting critical sections in Section 4 composes the Learning Oracle module and finally the difference

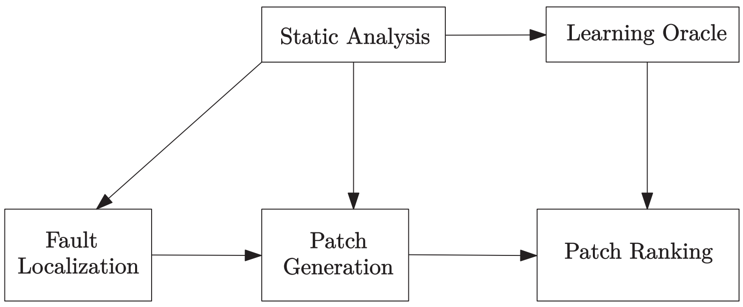


Fig. 11. Static Automatic Program Repair Architecture. The boxes represent the main components and the arrows how the information flow between them.

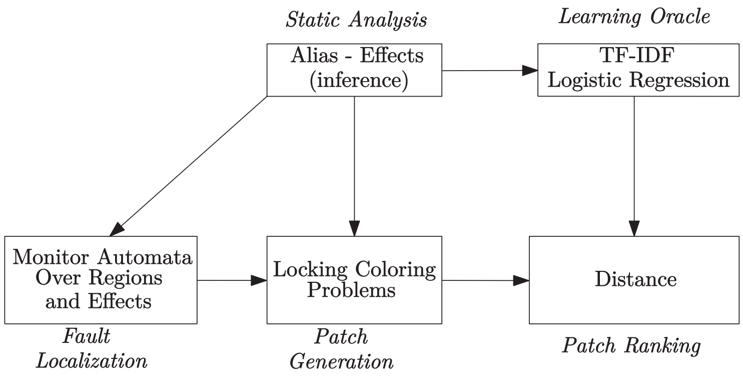

Static Analysis
Learning Oracle
Alias - Effects (inference)
TF-IDF Logistic Regression
Monitor Automata Over Regions and Effects
Locking Coloring Problems
Distance
Fault Localization
Patch Generation
Patch Ranking

Fig. 12. **Crayons** realization of the architecture in Figure 11.

between the oracle's prediction and the critical sections of the proposed repair in Equation (3) is our Patch Ranking module.

7 RELATED WORK

Program repair concerns the general problem of automatically producing source-level patches for bugs in programs [1]. Much of the research in APR focuses on *dynamic* repair, which uses test cases to identify bugs and validate patches. Techniques vary in patch construction methodology, from heuristic application of mutation operators [2, 7, 8], to use of inductive program synthesis over inferred specifications to construct replacement code [9, 10, 20], to combinations of both semantic and heuristic approaches [21]. Certain classes of dynamic repair have seen adoption in industry contexts [22, 23]. However, tests are only suitable for identifying certain types of bugs in certain classes of software. Some bugs are difficult to test for deterministically, like resource leaks or concurrency errors, while certain types of software (like many parts of the Linux Kernel) are not amenable to widespread automated testing. Most general-purpose, test-based, dynamic approaches do not readily apply to our domain.

Static analysis techniques are growing in popularity to find such bugs in practice [24]. The fact that fixing statically identified bugs remains a largely manual process is a barrier to widespread adoption of such tools [25, 26]. To that end, some static analyzers have begun suggesting lightweight "quick fixes" [27], but these tend to not be very semantically deep (e.g., inserting the `final` modifier). Researchers have recently sought to develop more complex templates for such warnings, either manually [28, 29] or using machine learning over source control histories [30, 31].

Machine learning over source control histories is a promising route for learning richer or more semantically meaningful static bug repairs, as demonstrated by Getafix [14] (which targets bugs flagged by Infer [32]) and Phoenix [33] (which integrates with Findbugs [34]). Deepfix applies a sequence-to-sequence model to learn fixes for compiler errors in student programs [35].

Although we also use static analysis to flag the bug under repair and to validate its removal, we do not rely on clustering or machine learning to suggest patches, and all training data required for our technique is taken from the program under repair. Our technique is perhaps more conceptually similar to FootPatch [13], which uses a separation logic-based static analysis to find heap-related bugs, and then identify (and syntactically contextualize) fixing code from elsewhere in the same program callgraph. We target a different class of bugs, and do not used logic-based inference to identify fix code, but do rely on analysis of resource and API usage in the rest of a given program to inform patch construction.

More formal methods for static repair include verification-based approaches, using LTL specifications [36, 37], SAT [38], deductive synthesis [39], model checking for Boolean programs [40], or abstract interpretation [41]. Although theoretically more rigorous in their guarantees, such verification-based program repair techniques have not been shown to scale or apply to bugs in real-world programs like the Linux Kernel.

Other closely related works include dynamic techniques that focus specifically on concurrency errors or atomicity violations, including Afix [42], Axis [43], Grail [19], PFix [44], HFix [45], and AFixer [46], and Cai and Cao's work [47], among others. Despite making use of some degree of static reasoning, this class of techniques typically either assume the provision as input of a bug report identifying a particular atomicity violation to be repaired, and are built on top of dynamic bug finders, or make use of runtime analysis explicitly (e.g., to analyze memory access patterns). Since we do not have access to runtime analysis of our test suite subjects, we are unable to compare against these tools. A broader class of concurrency bugs, including deadlocks, have been addressed using constraint solving and bounded model checking [48, 49]. These techniques provide strong guarantees, and can be guided to produce patches that minimally change the program, but have not been shown to scale beyond a couple hundred lines of code.

Deadlocks (and atomicity violations [50]) can also be avoided using runtime techniques [51–53], which instrument or otherwise monitor a program to preempt deadlocks during execution. This type of self-healing strategy does not seek to patch program source and can be viewed as complementary to our own approach. It typically imposes runtime overhead.

8 CONCLUSION

Crayons are a new static APR method and tool for bugs in using the lock API. **Crayons** scale to the code base of the Linux Kernel. The approach uses a combination of static, template-based generation of patch candidates with a learning-based model for prioritizing plausible patches. Our experiments show that the TF-IDF regression model has up to 80% performance in detecting critical sections, and that the entire repair technique generates a modest number of patches (it scales), it is able to recover human-made patches for 8 of 10 real Linux Kernel bugs, and it prioritizes the human-made patches among the top three for 63% of the cases and among the top five for 74%. To our best knowledge, this is the first static APR tool able to handle a so substantial and real code base. The prototype implementation of **Crayons** and the bugs used for testing are available at https://bitbucket.org/itu-square/crayonstosem.git.

REFERENCES

[1] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. DOI: http://dx.doi.org/10.1145/3318162

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 54–72.

[3] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 254–265. DOI: http://dx.doi.org/10.1145/2568225.2568254

[4] R. Kou, Y. Higo, and S. Kusumoto. 2016. A capable crossover technique on automatic program repair. In *Proceedings of the 7th International Workshop on Empirical Software Engineering in Practice (IWESEP'16)*. 45–50.

[5] F. Y. Assiri and J. M. Bieman. 2014. An assessment of the quality of automated program operator repair. In *Proceedings of the IEEE 7th International Conference on Software Testing, Verification, and Validation*. 273–282.

[6] Josh L. Wilkerson, Daniel R. Tauritz, and James M. Bridges. 2012. Multi-objective coevolutionary automated software correction. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO'12)*. ACM, New York, NY, 1229–1236. DOI: http://dx.doi.org/10.1145/2330163.2330333

[7] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, New York, NY, 1–11. DOI: http://dx.doi.org/10.1145/3180155.3180233

[8] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the Conference on Principles of Programming Languages (POPL'16)*. 298–31.

[9] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 691–701. DOI: http://dx.doi.org/10.1145/2884781.2884807

[10] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Trans. Softw. Eng.* 43, 1 (2017), 34–55.

[11] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. 532–543.

[12] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (Jan. 2011), 55 pages. DOI: http://dx.doi.org/10.1145/1889997.1890000

[13] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. ACM, New York, NY, 151–162. DOI: http://dx.doi.org/10.1145/3180155.3180250

[14] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. DOI: http://dx.doi.org/10.1145/3360585

[15] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 306–317. DOI: http://dx.doi.org/10.1145/2635868.2635898

[16] Iago Abal, Claus Brabrand, and Andrzej Wąsowski. 2017. Effective bug finding in C programs with shape and effect abstractions. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*, Ahmed Bouajjani and David Monniaux (Eds.).

[17] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 913–923.

[18] James A. Hanley and Barbara J. McNeil. 1982. *The Meaning and Use of the Area Under a Receiver Operating Characteristic Curve*. Vol. 143. 29–36.

[19] Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: Context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 318–329. DOI: http://dx.doi.org/10.1145/2635868.2635881

[20] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'17)*. 593–604. DOI: http://dx.doi.org/10.1145/3106237.3106309

[21] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2020. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Trans. Softw. Eng.* (2020). DOI: http://dx.doi.org/10.1109/TSE.2019.2944914

[22] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. 2017. *Fixing Bugs in Your Sleep: How Genetic Improvement Became an Overnight Success*. ACM, New York, NY, 1513–1520. https://doi.org/10.1145/3067695.3082517

[23] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated end-to-end repair at scale. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'19)*. 269–278. DOI: http://dx.doi.org/10.1109/ICSE-SEIP.2019.00039

[24] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective static analysis of concurrency {use-after-free} bugs in linux device drivers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. 255–268.

[25] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. ACM, 332–343. DOI: http://dx.doi.org/10.1145/2970276.2970347

[26] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE Computer Society, 672–681. DOI: http://dx.doi.org/10.1109/ICSE.2013.6606613

[27] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building useful program analysis tools using an extensible Java compiler. In *Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*. IEEE Computer Society, 14–23. DOI: http://dx.doi.org/10.1109/SCAM.2012.28

[28] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. *J. Syst. Softw.* 168 (2020), 110671. DOI: http://dx.doi.org/10.1016/j.jss.2020.110671

[29] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 314–325.

[30] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning quick fixes from code repositories. Retrieved from http://arxiv.org/abs/1803.03806.

[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. IEEE, 1–12.

[32] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *Proceedings of the Conference on NASA Formal Methods (NFM'15)*. 3–11.

[33] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. ACM, New York, NY, 613–624. DOI: http://dx.doi.org/10.1145/3338906.3338952

[34] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. DOI: http://dx.doi.org/10.1145/1052883.1052895

[35] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, 1345–1351.

[36] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program repair as a game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*. Springer-Verlag, Berlin, 226–238. DOI: http://dx.doi.org/10.1007/11513988_23

[37] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. 2014. Cost-aware automatic program repair. In *Static Analysis*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer International Publishing, Cham, 268–284.

[38] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based program repair using SAT. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*. 173–188.

[39] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive program repair. In *Proceedings of the Conference on Computer Aided Verification (CAV'15)*. 217–233.

[40] Andreas Griesmayer, Roderick Bloem, and Byron Cook. 2006. Repair of Boolean programs with an application to C. In *Proceedings of the Conference on Computer Aided Verification (CAV'06)*. 358–371.

[41] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. 133–146.

[42] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, 389–400.

[43] Peng Liu and Charles Zhang. 2012. Axis: Automatically fixing atomicity violations through solving control constraints. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, 299–309.

[44] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. 2018. PFix: Fixing concurrency bugs based on memory access patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. ACM, New York, NY, 589–600. DOI: http://dx.doi.org/10.1145/3238147.3238198

[45] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, New York, NY, 715–726. DOI: http://dx.doi.org/10.1145/2950290.2950309

[46] Yan Cai, Lingwei Cao, and Jing Zhao. 2017. Adaptively generating high quality fixes for atomicity violations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 303–314. DOI: http://dx.doi.org/10.1145/3106237.3106239

[47] Yan Cai and Lingwei Cao. 2016. Fixing deadlocks via lock pre-acquisitions. In *Proceedings of the 38th International Conference on Software Engineering*. 1109–1120.

[48] Sepideh Khoshnood, Markus Kusano, and Chao Wang. 2015. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. ACM, New York, NY, 165–176. DOI: http://dx.doi.org/10.1145/2771783.2771798

[49] Yiyan Lin and Sandeep S. Kulkarni. 2014. Automatic repair for multi-threaded programs with Deadlock/Livelock using maximum satisfiability. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, New York, NY, 237–247. DOI: http://dx.doi.org/10.1145/2610384.2610398

[50] Lee Chew and David Lie. 2010. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 307–320. DOI: http://dx.doi.org/10.1145/1755913.1755945

[51] Yarden Nir-Buchbinder, Rachel Tzoref, and Shmuel Ur. 2008. Deadlocks: From exhibiting to healing. In *Runtime Verification*, Martin Leucker (Ed.). Springer, Berlin, 104–118.

[52] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. 2017. UNDEAD: Detecting and preventing deadlocks in production software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE Press, 729–740.

[53] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. 2008. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 281–294.

Received 29 September 2021; revised 24 May 2022; accepted 24 June 2022