Conditional Correlation Analysis for Safe Region-based Memory Management

Xi Wang[†] Zhilei Xu[†] Xuezheng Liu[‡] Zhenyu Guo[‡] Xiaoge Wang[†] Zheng Zhang[‡]

[†]Tsinghua University [‡]Microsoft Research Asia

{v-xwa, v-zhixu, xueliu, zhenyug, zzhang}@microsoft.com wangxg@tsinghua.edu.cn

Abstract

Region-based memory management is a popular scheme in systems software for better organization and performance. In the scheme, a developer constructs a hierarchy of regions of different lifetimes and allocates objects in regions. When the developer deletes a region, the runtime will recursively delete all its subregions and simultaneously reclaim objects in the regions. The developer must construct a *consistent* placement of objects in regions; otherwise, if a region that contains pointers to other regions is not always deleted *before* pointees, an inconsistency will surface and cause dangling pointers, which may lead to either crashes or leaks.

This paper presents a static analysis tool RegionWiz that can find such lifetime inconsistencies in large C programs using regions. The tool is based on an analysis framework that generalizes the relations and constraints over regions and objects as conditional correlations. This framework allows a succinct formalization of consistency rules for region lifetimes, preserving memory safety and avoiding dangling pointers. RegionWiz uses these consistency rules to implement an efficient static analysis to compute the conditional correlation and reason about region lifetime consistency; the analysis is based on a context-sensitive, field-sensitive pointer analysis with heap cloning.

Experiments with applying RegionWiz to six real-world software packages (including the RC compiler, Apache web server, and Subversion version control system) with two different region-based memory management interfaces show that RegionWiz can reason about region lifetime consistency in large C programs. The experiments also show that RegionWiz can find several previously unknown inconsistency bugs in these packages.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability; D.3.4 [*Programming Languages*]: Processors—Memory management; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Experimentation, Reliability, Verification

Keywords region, conditional correlation, program analysis, error detection, memory management

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

1. Introduction

Region-based memory management [21, 38, 6] is an alternative scheme to explicit allocation and deallocation (e.g., `malloc` and `free`) and automatic garbage collection [7, 41]. In this scheme, a developer constructs a hierarchy of regions (a.k.a. pools) of different lifetimes and allocates objects in regions. Region lifetimes are nested; when the developer deletes a region, the memory management runtime will delete the region and its subregions recursively, deallocating all the contained objects.

Regions expose a simpler interface than explicit allocation and deallocation to organize complex data structures, since programmers can delete sets of objects instead of only individual objects. It also preserves the safety of intra-region pointers and reduces the risk of leaks and double frees. On the other hand, compared to garbage collection, it still enables fine-grained control of the lifetimes of objects via regions, which is usually required in performance-critical systems software such as operating systems and servers. Furthermore, because the memory allocators of different regions are usually independent of each other, developers can separate related objects into the same region to express data locality, avoid lock contention, and batch allocation and deallocation. Hence, programs using regions can often achieve better performance [16, 17].

In practice, regions are popular in software that operates in stages, such as compilers and network applications. A staged application generally has an inherent hierarchical structure; region-based memory management can match up with the structure via a region hierarchy for better organization and performance. For example, a web server maintains a group of TCP connections, and a TCP connection contains a series of HTTP requests. Thus, an HTTP request is a "child" of a TCP connection, i.e., the request has a shorter lifetime. A developer can assign a region to the connection and a subregion to the request, then allocate resources used throughout the connection from the parent region and those used during processing the request only from the subregion. The developer can delete either the subregion (if the request has been processed), or the parent region (if the connection is closed) so that the runtime can further delete the subregions of the requests and reclaim all memory easily.

Figure 1 lists the pseudo code snippet of the web server example above, with two region primitives: 1) **rnew** creates a subregion of the given parent region, and 2) **ralloc** allocates a object in the given region. First, line 1 allocates a connection object `conn` in region `r` using **ralloc**. Later, line 3 creates a subregion `subr` using **rnew**, taking `r` as its parent region. Then line 5 allocates a request object `req` in `subr`, and line 6 assigns field `req.connection` with a pointer to `conn`. We omit further details of the hierarchy (e.g., the parent region and other subregions of `r`) for simplicity.

The connection-request example involves three relations:

```
1 conn = ralloc(r);
2 ...
3 subr = rnew(r);
4 ...
5 req = ralloc(subr);
6 req.connection = conn;
```

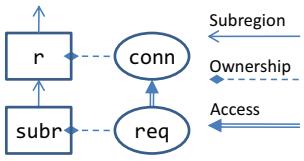

Figure 1. The connection-request example.

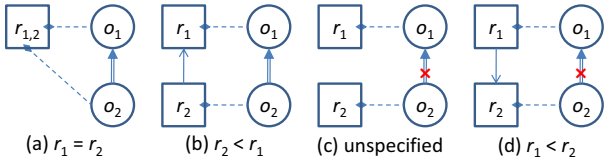

Figure 2. Four different subregion relations between r_1 and r_2.

- the *subregion* relation over regions, specified by calls to **rnew**, e.g., `subr` and `r` at line 3;
- the *ownership* relation over regions and objects, specified by calls to **ralloc**, e.g., `r` and `conn` at line 1, `subr` and `req` at line 5;
- the *access* relation over objects, implied by field assignments of inter-object pointers, e.g., `req` and `conn` at line 6.

The access relation requires that `req` should be reclaimed *before* `conn` to avoid the pointer `req.connection` being dangling. Thus, combined with the ownership relation, a safety requirement is that the region `subr` that owns `req` should have a shorter lifetime than the region `r` that owns `conn`. Meanwhile, the developer specifies the subregion relation that `subr` is a subregion of `r`, which is consistent with the safety requirement. Otherwise, if `subr` is not *always* deleted before `r`, e.g., `subr` is not a subregion of `r` or even it is the parent of `r`, the pointer `req.connection` may be dangling.

Therefore, the three relations are *correlated* and they must be *consistent*; the specified subregion relation and the access relation implied by inter-object pointers constrain each other via the ownership map over regions and objects. We refer such correlation as *conditional correlation*, and develop a formalization of the consistency problem in Section 3.

Dangling pointers caused by the inconsistencies can harm the correctness and robustness of software using regions. Intuitively, further use of the dangling pointers would lead to crashes. Furthermore, even a dangling pointer is never used and the program does not crash, it may cause leaks as well. Consider the example in Figure 1 again. The developer might not code `subr` as a subregion of `r`, e.g., `subr` might be a subregion of the root region that lives forever, or even `r` inappropriately takes `subr` as its parent region. In such cases, the object `req` that resides in `subr` *unnecessarily* consumes memory even after `conn` is reclaimed, and the developer cannot use this memory any more. More seriously, such objects of longer-than-necessary lifetime may lead to unpredictable memory consumption [5] especially if a function that contains such buggy code resides in recursions or loops. Of course, there are no real "leaks" in region-based memory management as defined in the `malloc-free` scheme, because the runtime will delete all regions eventually. In this paper, we use the term "leaks" to refer to such longer-than-necessary lifetime cases.

Unfortunately, it is difficult for developers to track such consistency interprocedurally in large code base. For example, as in Figure 1, the code at line 5 and 6 may be in a function `foo` given a object `conn` and a region `sub` as parameters. Consequently, `foo` assumes that the caller must be careful about both allocating `conn` from an appropriate region and creating `subr` with a consistent parent region. However, the parameters for `foo` may be passed deep along call paths, and the program points of allocating `conn` and creating `subr` may be far way from `foo`. Thus, caller code may be unaware of the implicit constraint, or it may assume that `foo` will keep a duplicated copy of the object referred by a parameter, which could be true especially when the object is a string. In either case, the code is prone to inconsistencies that may result in dangling pointers.

Developers may discover inconsistencies that lead to crashes after they surface at runtime. It is harder to find such bugs in multi-threaded programs with regions, because the deletion order of regions may vary due to scheduling and the bugs may not appear in each run. Several dynamic approaches have been proposed to preserve memory safety with regions at runtime [16, 17], but they cannot find inconsistencies that are on less-executed code paths and that are sensitive to runtime environments such as scheduling, nor can they solve the leaks caused by inconsistencies.

In this paper, we focus on static analysis techniques and present a prototype tool RegionWiz that searches exhaustively for region lifetime inconsistencies in source code. To the best of our knowledge, it is the first tool to address the problem in large C programs using regions. Specifically, the main contributions of this paper are: 1) a unified framework of conditional correlation that may be of independent interest; 2) a formalization of the region lifetime consistency problem as an instantiation; 3) an implementation that employs a context-sensitive, field-sensitive pointer analysis with heap cloning and that performs a conditional correlation analysis for the region lifetime consistency problem in large C programs; and, 4) the evaluation with six real-world software packages of two region-based memory management interfaces.

The rest of the paper is organized as follows. Section 2 gives an overview of the analysis methodology. Section 3 defines the concept of conditional correlation. Section 4 formalizes region lifetime consistency based on conditional correlation and describes a static analysis algorithm. Section 5 presents our implementation details. Section 6 reports our experimental results. We survey related work in Section 7 and conclude in Section 8.

2. Overview

In this section we present an overview of how RegionWiz reasons about region lifetime consistency.

For two regions r_1, r_2, we write $r_1 < r_2$ if r_1 is a subregion of r_2, and $r_1 = r_2$ if r_1, r_2 refers to the same region. Further, we write $r_1 \leq r_2$ if r_1 is a direct or indirect subregion of r_2, while $r_1 \not\leq r_2$ otherwise. The partial order $\leq$ is the reflexive transitive closure of the subregion relation $<$ over regions.

Consider a simple case that o_2 in region r_2 holds a pointer to o_1 in region r_1. There are four possible subregion relations between r_1 and r_2 in caller code, as illustrated in Figure 2.

(a) r_1, r_2 refer to the same region, i.e., $r_1 = r_2$, so o_1, o_2 share the same lifetime, and the intra-region pointer is always safe.

(b) r_2 is a subregion of r_1, i.e., $r_2 < r_1$, so o_2 will be deleted before o_1, and the inter-region pointer is always safe.

(c) There is no subregion relation between r_1 and r_2, i.e., $r_1 \not\leq r_2$ and $r_2 \not\leq r_1$, so the inter-region pointer may be dangling if r_1 is deleted first.

(d) r_1 is a subregion of r_2, i.e., $r_1 < r_2$, so o_1 will be deleted first, and the inter-region pointer will become dangling.

As in Figure 2, $r_2 \leq r_1$ holds in (a) and (b), where the pointers from o_2 to o_1 are always safe. Meanwhile, the pointers from o_2 to o_1 may be dangling in (c) and (d), where $r_2 \not\leq r_1$ and r_1 may be deleted before r_2. To sum up, we have the following rules for two objects o_1, o_2 in regions r_1, r_2, respectively.

```
1 o1 = ralloc(r1);
2 if P r = r0;
3 if Q r = r1;
4 r2 = rnew(r);
5 o2 = ralloc(r2);
6 o2.f = o1;
```

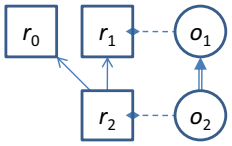

Figure 3. An aliasing example.

Proposition 2.1. If o_2 may access o_1, $r_2 \leq r_1$ *must* hold.

Proposition 2.2. If $r_2 \leq r_1$ holds, any pointer from o_2 that may access o_1 is *always* safe.

With Proposition 2.1 one can reason about consistency starting from objects. The relations of ownership and access combine to imply that $r_2 \leq r_1$ must hold. However, the must-subregion requirement may be hard to prove in the presence of aliasing. Consider a degradation case that o_1, too, holds a pointer to o_2, i.e., o_1, o_2 can access each other. To preserve memory safety, o_1 and o_2 under this circumstance must reside in the same region. In other words, it suffices to show that the two variables r_1 and r_2 must always refer to the same region. This approach requires a must-alias analysis [2], which is generally difficult to perform [33].

Nevertheless, we cannot just show that $r_2 \leq r_1$ may hold and conclude that region lifetime is consistent; the compromise is unsound and may lead to missing of inconsistencies. Consider the code in Figure 3, where r_2 may be a subregion of r_1 if condition Q evaluates true. However, the region lifetime in Figure 3 is not consistent because $r_2 \leq r_1$ does not always hold; r_2 can be a subregion of r_0 when P evaluates true and Q evaluates false. In this case there is no subregion partial order between r_2 and r_1, so that the pointer $o_2.f$ may be dangling.

RegionWiz takes the other approach to reason about consistency by using Proposition 2.2. As stated, if $r_2 \leq r_1$ holds, we can conclude that any pointer from o_2 to o_1 is always safe. Thus, it suffices to show that for any two regions x, y that have no subregion partial order $x \not\leq y$, each object in x must not access any object in y. By verifying the non-access property against region pairs that have no subregion partial order, we can prove that region lifetime is consistent.

Take Figure 3 as example again. Note that we have just discussed that $r_2 \leq r_1$ does not always hold. So the most conservative estimation of region pairs that have no subregion partial order is the set $\{r_i \not\leq r_j | i \neq j; i, j = 0, 1, 2\}$. To verify them, we can find that for $r_2 \not\leq r_1$, o_2 in r_2 may access o_1 in r_1, which violates Proposition 2.2 and is a potential inconsistency.

To sum up, our tool RegionWiz reasons about region lifetime consistency in the following steps:

1. start with region pairs that have no subregion partial order,
2. map regions to objects they own via the ownership relation, and
3. verify the non-access property against sets of objects.

In the example above, the chosen of region pairs that have no subregion partial order may be too conservative: if r_2 may be a subregion of more than one region due to aliasing, we have to verify its objects against those in all its ancestors for a sound approximation, which may yield a large mount of false warnings. We will discuss improvements in Section 4.3.

3. Conditional Correlation

The region lifetime consistency problem correlates the two sets, i.e., regions and objects, and involves three relations of subregion, ownership, and access. We generalize the correlation between two sets of objects as follows.

Definition 3.1 (Conditional Correlation)**.** Let $\mathbb{A}, \mathbb{B}$ be two sets with binary relations $f : \mathbb{A} \times \mathbb{A}$, $g : \mathbb{B} \times \mathbb{B}$, and map $\varphi : \mathbb{A} \to \mathbb{B}$. We define a *conditional correlation* $\langle f, \varphi, g \rangle$ over $\mathbb{A}, \mathbb{B}$ as

$$(x, y) \in f \implies (\varphi(x), \varphi(y)) \in g \tag{3.1}$$

If $\langle f, \varphi, g \rangle$ holds for (x, y), it means that $(\varphi(x), \varphi(y)) \in g$ can be proven under the assumption $(x, y) \in f$. In other words, φ is a *relation-preserving* map from $\mathbb{A}$ to $\mathbb{B}$ with respect to f and g.

Definition 3.2 (Consistent Conditional Correlation)**.** A conditional correlation $\langle f, \varphi, g \rangle$ is *consistent* over $\mathbb{A}, \mathbb{B}$ if it holds for all $(x, y) \in \mathbb{A} \times \mathbb{A}$.

Note that $\langle f, \varphi, g \rangle$ always holds for $(x, y) \in (\mathbb{A} \times \mathbb{A}) \backslash f$, so it suffices to show that (3.1) holds for $(x, y) \in f$ to prove the consistency of the conditional correlation.

We briefly describe the region lifetime consistency problem as an example of conditional correlation. Consider a set of regions $\mathbb{R}$ with the subregion relation, and a set of objects $\mathbb{O}$ where each object may access some other objects.

- $\mathbb{A} = \mathbb{R}$ is the set of regions;
- $\mathbb{B} = 2^{\mathbb{O}}$ is the powerset of objects;
- f is all region pairs (r_1, r_2) over $\mathbb{A} \times \mathbb{A}$ that $r_1 \not\leq r_2$ holds;
- φ maps a region to a set of objects that it owns;
- g is all object set pairs (s_1, s_2) over $\mathbb{B} \times \mathbb{B}$ that each object in s_1 cannot access any object in s_2;

By definition, we can see that the ownership relation φ is a relation-preserving map with respect to f and g, and $\langle f, \varphi, g \rangle$ is a conditional correlation for reasoning about region lifetime consistency.

Now we consider how to perform a conditional correlation analysis statically. Intuitively, as for a sound approximation, if we prove a conditional correlation $\langle F, \Phi, G \rangle$ is consistent, where F is a superset of f, Φ is a superset of φ, and G is a subset of g, we can conclude that $\langle f, \varphi, g \rangle$ is also consistent. Hence, to perform a conditional correlation analysis, we can estimate an over-approximation of f, φ, and an under-approximation of g. More generally, we define an abstraction relation $\preceq$ over conditional correlations.

Definition 3.3 (Conditional Correlation Analysis)**.** Given two conditional correlations $\langle f, \varphi, g \rangle$ over $\mathbb{A}, \mathbb{B}$ and $\langle F, \Phi, G \rangle$ over $\mathbb{A}', \mathbb{B}'$, let $\alpha : \mathbb{A} \to \mathbb{A}'$, $\beta : \mathbb{B} \to \mathbb{B}'$ be two maps. We define $\langle f, \varphi, g \rangle \preceq \langle F, \Phi, G \rangle$ if the following conditions are satisfied.

$$\forall (x, y) \in \mathbb{A} \times \mathbb{A} : (x, y) \in f \implies (\alpha(x), \alpha(y)) \in F \tag{3.2}$$

$$\forall x \in \mathbb{A}, s \in \mathbb{B} : \varphi(x) = s \implies \Phi(\alpha(x)) = \beta(s) \tag{3.3}$$

$$\forall (s, t) \in \mathbb{B} \times \mathbb{B} : (s, t) \notin g \implies (\beta(s), \beta(t)) \notin G \tag{3.4}$$

By definition, given two conditional correlations $\langle f, \varphi, g \rangle$ and $\langle F, \Phi, G \rangle$ such that $\langle f, \varphi, g \rangle \preceq \langle F, \Phi, G \rangle$, it is easy to show that $\langle f, \varphi, g \rangle$ holds if $\langle F, \Phi, G \rangle$ holds.

In general, to find an appropriate $\langle F, \Phi, G \rangle$ for a specific conditional correlation $\langle f, \varphi, g \rangle$ over $\mathbb{A}, \mathbb{B}$ usually depends on the topologies or the shapes [18] of the two sets, e.g., are they trees, directed acyclic graphs, or cyclic graphs? In the region lifetime consistency problem, we notice that the regions shape a tree while objects form a graph. We will further discuss analysis techniques exploiting the property in Section 4.

4. Region Lifetime Consistency

In this section, we formalize the region lifetime consistency problem based on conditional correlation and discuss static analysis algorithms to reason about the consistency.

4.1 Language

To develop the formalization, we describe a weakly-typed toy language. Let $\mathbb{R}$ be the set of regions and $\mathbb{H}$ be the set of normal objects allocated in regions. We use the term "object" to refer to either null, a region $r \in \mathbb{R}$, or a normal object $h \in \mathbb{H}$. Each normal object may have several fields $f \in \mathbb{F}$ to access other objects. The language has the following statements.

$$s ::= x = \textbf{null} \mid x = \textbf{rnew}\ y \mid x = \textbf{ralloc}\ y \mid x = y \mid x = y.f \mid x.f = y \mid s_1\ ;\ s_2 \mid \textbf{if} \sim s_1\ \textbf{else}\ s_2 \mid \textbf{while} \sim s$$

A variable may be initialized to **null**. A new subregion can be created using **rnew**, given a parameter as the parent region. A new normal object may be allocated in a given region using **ralloc**. If the parameter given in **rnew** or **ralloc** is **null**, it means the root region, denoted as Λ. In addition, the language has standard assignment, load, store, composition, branching, and looping statements. We consider the subregion relation enforces the region deletion order, so we do not model region deletion in the language.

Let $\mathbb{O} = \mathbb{R} \cup \mathbb{H}$ be the union of regions and normal objects, so that $o \in \mathbb{O}$ is a non-null object. $\mathbb{O}_\perp = \mathbb{O} \cup \{\perp\}$ represents all objects including **null**. We define the concrete contexts and effects as follows.

$$\begin{aligned}
\rho &::= \mathbb{V} \to \mathbb{O}_\perp && \text{(environment)}\\
\delta &::= \mathbb{H} \times \mathbb{F} \to \mathbb{O}_\perp && \text{(heap)}\\
\pi &::= \mathbb{R} \times \mathbb{R} && \text{(subregion)}\\
\phi &::= \mathbb{R} \times \mathbb{O} && \text{(ownership)}\\
\sigma &::= \mathbb{O} \times \mathbb{O} && \text{(access)}
\end{aligned}$$

The environment ρ maps a variable to an object. The heap δ tracks the set of objects that each normal object can access via its fields. In addition, three effects record necessary information for reasoning about consistency as follows.

- $r < r'$ in π records that r is a subregion of r'.
- $r \rhd o$ in ϕ records that region r owns object o.
- $o \twoheadrightarrow o'$ in σ records that object o can access object o'.

Since **null** can be used to refer to the root region Λ in **rnew** and **ralloc**, we define $\rho_\gamma : \mathbb{V} \to \mathbb{R}$ in addition to ρ for convenience. Note that ρ_γ is undefined for a normal object.

$$\rho_\gamma(x) = \begin{cases} r & \text{if } \rho(x) = r \\ \Lambda & \text{if } \rho(x) = \perp \end{cases}$$

Figure 4 shows the big-step operational semantics of the language. A judgment $s, \rho, \delta \Downarrow \rho', \delta', \pi_0, \phi_0, \sigma_0$ means that after each statement s the environment ρ and the heap δ change to ρ' and δ', respectively, and the statement s generates new tuples π_0, ϕ_0, σ_0 in relations π, ϕ, σ, respectively. We explain each judgment in detail.

- (4.1) Variable x is initialized as **null**.
- (4.2) A new region r is created as a subregion of the parent r' referred by y, and is assigned to x. A new subregion tuple $r < r'$ is generated in π.
- (4.3) A new normal object h is allocated in region r referred by y, and is assigned to x. A new ownership tuple $r \rhd h$ is generated in ϕ.
- (4.4) Variable x is assigned with the object that y refers to.
- (4.5) Variable x is assigned with the object that normal object h can access via field f, where h is referred by y.
- (4.6) Field f of normal object h is assigned with the object that y refers to, where h is referred by x. A new access tuple $h \twoheadrightarrow \rho(y)$ is generated in σ if y refers to a non-null object.

$$x = \textbf{null}, \rho, \delta \Downarrow \rho[x \mapsto \perp], \delta, \emptyset, \emptyset, \emptyset \tag{4.1}$$

$$\frac{\rho_\gamma(y) = r' \quad r \text{ fresh} \quad \pi_0 = \{r < r'\}}{x = \textbf{rnew}\ y, \rho, \delta \Downarrow \rho[x \mapsto r], \delta, \pi_0, \emptyset, \emptyset} \tag{4.2}$$

$$\frac{\rho_\gamma(y) = r \quad h \text{ fresh} \quad \phi_0 = \{r \rhd h\}}{x = \textbf{ralloc}\ y, \rho, \delta \Downarrow \rho[x \mapsto h], \delta[h.f_i \mapsto \perp], \emptyset, \phi_0, \emptyset} \tag{4.3}$$

$$x = y, \rho, \delta \Downarrow \rho[x \mapsto \rho(y)], \delta, \emptyset, \emptyset, \emptyset \tag{4.4}$$

$$\frac{\rho(y) = h}{x = y.f, \rho, \delta \Downarrow \rho[x \mapsto \delta(h.f)], \delta, \emptyset, \emptyset, \emptyset} \tag{4.5}$$

$$\frac{\rho(x) = h \quad \sigma_0 = \text{if } \rho(y) = \perp \text{ then } \emptyset \text{ else } \{h \twoheadrightarrow \rho(y)\}}{x.f = y, \rho, \delta \Downarrow \rho, \delta[h.f \mapsto \rho(y)], \emptyset, \emptyset, \sigma_0} \tag{4.6}$$

$$\frac{s_1, \rho, \delta \Downarrow \rho', \delta', \pi_1, \phi_1, \sigma_1 \quad s_2, \rho', \delta' \Downarrow \rho'', \delta'', \pi_2, \phi_2, \sigma_2}{s_1\ ;\ s_2, \rho, \delta \Downarrow \rho'', \delta'', \pi_1 \cup \pi_2, \phi_1 \cup \phi_2, \sigma_1 \cup \sigma_2} \tag{4.7}$$

$$\frac{s_1, \rho, \delta \Downarrow \rho', \delta', \pi, \phi, \sigma}{\textbf{if} \sim s_1\ \textbf{else}\ s_2, \rho, \delta \Downarrow \rho', \delta', \pi, \phi, \sigma} \tag{4.8}$$

$$\frac{s_2, \rho, \delta \Downarrow \rho', \delta', \pi, \phi, \sigma}{\textbf{if} \sim s_1\ \textbf{else}\ s_2, \rho, \delta \Downarrow \rho', \delta', \pi, \phi, \sigma} \tag{4.9}$$

$$\textbf{while} \sim s, \rho, \delta \Downarrow \rho, \delta, \emptyset, \emptyset, \emptyset \tag{4.10}$$

$$\frac{s, \rho, \delta \Downarrow \rho', \delta', \pi_1, \phi_1, \sigma_1 \quad \textbf{while} \sim s, \rho', \delta' \Downarrow \rho'', \delta'', \pi_2, \phi_2, \sigma_2}{\textbf{while} \sim s, \rho, \delta \Downarrow \rho'', \delta'', \pi_1 \cup \pi_2, \phi_1 \cup \phi_2, \sigma_1 \cup \sigma_2} \tag{4.11}$$

Figure 4. Operational semantics.

- (4.7) π, ϕ, and σ are union of the tuples generated by two composition statements.
- (4.8) (4.9) π, ϕ, and σ is the tuples generated by a branching statement.
- (4.10) The loop is not executed.
- (4.11) π, ϕ, and σ are union of the tuples generated during the execution of the loop.

Example 4.1. Consider the code in Figure 3. Assuming that $\rho(r_0) = \gamma_0$, $\rho(r_1) = \gamma_1$, P, Q both evaluate true: line 1, according to (4.3), allocates a normal object h_1, assigns $\rho(o_1) = h_1$, and generates $\gamma_1 \rhd h_1$ in ϕ; line 2, according to (4.4), (4.8), and our assumption, assigns $\rho(r) = \gamma_0$; line 3, similar to line 2, assigns $\rho(r) = \gamma_1$; line 4, according to (4.2), creates a new subregion γ_2, assigns $\rho(r_2) = \gamma_2$, and generates $\gamma_2 < \gamma_1$ in π; line 5, similar to line 1, allocates h_2, assigns $\rho(o_2) = h_2$, and generates $\gamma_2 \rhd h_2$ in ϕ; line 6, according to (4.6), assigns $\delta(h_2.f) = h_1$ and generates $h_2 \twoheadrightarrow h_1$ in σ.

4.2 Problem Formulation

Now we formulate the region lifetime consistency problem for the language. Let the partial order π^+ be the reflexive transitive closure of the subregion relation π. In addition, we extend the ownership relation ϕ to the reflexive closure $\phi^=$ to capture the inconsistency that a normal object in r_1 can access region r_2 if $r_1 \not\leq r_2$. To prove consistency, it suffices to show that for any regions x, y that have no partial order $x \not\leq y$, i.e., $(x, y) \notin \pi^+$, each object in $\phi^=(x)$ must not access any object in $\phi^=(y)$. In the words, the following set must be empty.

$$\left\{(o_1, o_2) | (o_1, o_2) \in \sigma : o_1 \in \phi^=(x) \wedge o_2 \in \phi^=(y) \wedge (x, y) \notin \pi^+\right\} \tag{4.12}$$

To simplify (4.12), extend the access relation σ to σ^* over the powerset of objects as follows.

$$\sigma^* = \{(s_1, s_2) | s_1, s_2 \in 2^{\mathbb{O}} \wedge \exists o_1 \in s_1, o_2 \in s_2 : (o_1, o_2) \in \sigma\}$$

Applying σ^* to (4.12), it suffices to prove the following proposition holds for consistency.

$$\forall (x, y) \notin \pi^+ : (\phi^=(x), \phi^=(y)) \notin \sigma^*$$

Let $\overline{\pi^+}, \overline{\sigma^*}$ denote the complement of π^+, σ^*, respectively. We rewrite the above proposition as follows.

$$\forall (x, y) \in \overline{\pi^+} : (\phi^=(x), \phi^=(y)) \in \overline{\sigma^*} \quad (4.13)$$

(3.1) and (4.13) combine to show that the region lifetime consistency problem is an instantiation of conditional correlation.

Definition 4.1 (Region Lifetime Consistency)**.** $\langle \overline{\pi^+}, \phi^=, \overline{\sigma^*} \rangle$ is a conditional correlation over $\mathbb{R}$ and $2^{\mathbb{O}}$.

Example 4.2. Continue with Figure 3. Example 4.1 already shows that it generates the tuples in the three relations $\pi : \{\gamma_2 < \gamma_1\}$, $\phi : \{\gamma_1 \rhd h_1, \gamma_2 \rhd h_2\}$, and $\sigma : \{h_2 \twoheadrightarrow h_1\}$. So there are five region pairs in $\overline{\pi^+}$: $\gamma_0 \not< \gamma_1$, $\gamma_0 \not< \gamma_2$, $\gamma_1 \not< \gamma_0$, $\gamma_1 \not< \gamma_2$, and $\gamma_2 \not< \gamma_0$. Further, $\phi^=$ maps them to corresponding object powerset pairs: $(\{\gamma_0\}, \{\gamma_1, h_1\})$, $(\{\gamma_0\}, \{\gamma_2, h_2\})$, $(\{\gamma_1, h_1\}, \{\gamma_0\})$, $(\{\gamma_1, h_1\}, \{\gamma_2, h_2\})$, and $(\{\gamma_2, h_2\}, \{\gamma_0\})$, respectively. It is easy to show that all the pairs are in $\overline{\sigma^*}$ in the case that both P, Q evaluate true, so the region lifetime here is consistent.

4.3 Static Analysis

Consider a static analysis algorithm that computes abstract effects Π, Φ, and Σ for π, ϕ, and σ, respectively, such that $\langle \overline{\pi^+}, \phi^=, \overline{\sigma^*} \rangle \preceq \langle \overline{\Pi^+}, \Phi^=, \overline{\Sigma^*} \rangle$ holds. As we have discussed in Section 3, a sound algorithm should estimate an over-approximation of $\overline{\pi^+}, \phi^=$, and an under-approximation of $\overline{\sigma^*}$, that is, an under-approximation of π, and an over-approximation of ϕ, σ.

We briefly describe a standard Anderson-style analysis [3]. In addition to the three abstract effects Π, Φ, and Σ, the algorithm estimates two abstract contexts: abstract environment Γ and abstract heap Δ. Using abstract locations instead of fresh objects, it is straightforward to compute abstract contexts and effects in initialization, region creation, normal object allocation, assignment, load, store, and composition statements for the toy language in parallel with the semantics. For branching statements, the algorithm joins the abstract contexts and effects on both paths by union of abstract contexts and effects. For looping statements, it iterates until reaching a fixed point. We omit the details for brevity. Intuitively, the algorithm estimates Π, Φ, and Σ as an over-approximation of π, ϕ, and σ, respectively, either flow-sensitive or flow-insensitive.

Example 4.3. Consider Figure 3 again. Similar to Example 4.1, assuming that $\Gamma(r_0) = \{\gamma_0\}$, $\Gamma(r_1) = \{\gamma_1\}$: line 1 generates (o_1, h_1) in Γ and $\gamma_1 \rhd h_1$ in Φ; line 2 generates (r, γ_0) in Γ; line 3 generates (r, γ_1) in Γ, so now $\Gamma(r) = \{\gamma_0, \gamma_1\}$; line 4 generates (r_2, γ_2) in Γ and $\gamma_2 < \gamma_0, \gamma_2 < \gamma_1$ in Π; line 5 generates (o_2, h_2) in Γ and $\gamma_2 \rhd h_2$ in Φ; line 6 generates $(h_2.f, h_1)$ in Δ and $h_2 \twoheadrightarrow h_1$ in Σ. Here $\gamma_0, \gamma_1, h_1, h_2$ are abstract locations; the abstract effects are $\Pi : \{\gamma_2 < \gamma_0, \gamma_2 < \gamma_1\}$, $\Phi : \{\gamma_1 \rhd h_1, \gamma_2 \rhd h_2\}$, and $\Sigma : \{h_2 \twoheadrightarrow h_1\}$.

Note that Π is an over-approximation of π. However, we need an under-approximation. As we have discussed in Section 2, using such imprecise subregion edges can be unsound and will miss region pairs that should be verified in $\overline{\Pi^+}$, e.g., $\gamma_2 < \gamma_1$.

Generally, the subregion relation π should form a tree, where each region (except for the root region Λ) has one and only one parent. A static analysis algorithm may conservatively estimate region r to be a subregion of several regions $r_1, r_2, \ldots, r_n$, rather than a unique parent in semantics. In such cases, regions form a join-semilattice, where the root region Λ is the top. We consider the parent region of r as the join of all its possible parent regions $\bigvee_{i=1}^{n} r_i$, and replace them with the tuple $r < \bigvee_{i=1}^{n} r_i$ in Π.

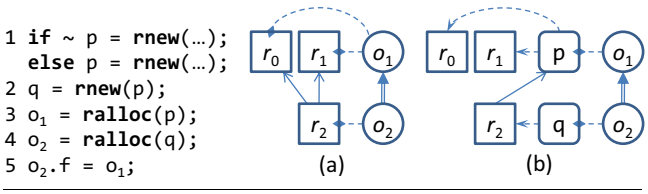

Figure 5. An intra-region pointer example.

Example 4.4. Continue with Example 4.3. Assume that the parent of either γ_0 or γ_1 is the root region Λ. The algorithm finally generates $\Pi : \{\gamma_i < \Lambda | i = 0, 1, 2\}$, so that there are six region pairs in $\overline{\Pi^+}$ to be verified, as discussed in Section 2.

The algorithm can be flow-insensitive, thus we adopt more precise pointer analysis techniques [22] while preserving scalability. We should distinguish effects (e.g., the subregion, ownership, and access relations) in individual contexts (e.g., call paths), so we need context-sensitivity [14]. Furthermore, we should precisely track what each field of an object refers to, so we also use field-sensitivity. In addition, we should be able to distinguish different instances of objects that are created at the same call site but on different call paths, thus heap cloning or specialization is critical for precision [36]. RegionWiz implements a flow-insensitive, context-sensitive, field-sensitive analysis with heap cloning.

However, the algorithm comes at a price. Consider the example in Figure 5. Line 1 creates either r_0 or r_1; line 2 creates a subregion r_2, taking the region that p refers to as its parent. When p refers to either $r_i, i = 0, 1$, o_1 is in r_i and r_2 is also a subregion of r_i. Since o_2 is in r_2, the region lifetime is always consistent. Our flow-insensitive algorithm will conservatively yield an imprecise result as shown in Figure 5(a) and report a false warning, because it fails to capture the fact that the parent of r_2 and the owner of o_1 are always the same region.

A possible approach is to introduce an indirect level. Refine the relations of subregion π and ownership ϕ as $\pi' : \mathbb{R} \times \mathbb{V}$ and $\phi' : \mathbb{V} \times \mathbb{O}$, respectively, where $\mathbb{V}$ is the set of variables. A new static analysis algorithm that computes def-use information for each variable can yield a more precise result, as shown in Figure 5(b). The new subregion tuple is $r_2 \leq' p$, while the new ownership tuples are $p \rhd' o_1, q \rhd' o_2$. In this case it suffices to verify against (region) variable pairs v_1, v_2 if there exists region r that variable v_2 may refer to such that $r \not\leq' v_1$. A practical implementation can adopt techniques such as IPSSA [29], an unsound but effective approach. We defer it to future work.

5. Implementation

Our prototype tool RegionWiz currently supports two region-based memory management interfaces used in real-world C programs: RC regions [17] and Apache Portable Runtime (APR) pools [23]. We use the interface of APR pools as an example, which is widely adopted by various software packages such as Apache web server and Subversion version control system. Figure 6 lists part of the interface.

Similar to **rnew**, `apr_pool_create` creates a new subregion of a given parent region (or the root if given null); the subregion can be retrieved by dereferencing the first parameter `newp`, a pointer to pointer. A function `apr_palloc`, similar to **ralloc**, allocates a

```
/* region creation (rnew) */
apr_status_t
apr_pool_create(apr_pool_t **newp, apr_pool_t *parent);
/* object allocation (ralloc) */
void * apr_palloc(apr_pool_t *p, apr_size_t size);
void * apr_pcalloc(apr_pool_t *p, apr_size_t size);
/* region deletion */
void apr_pool_clear(apr_pool_t *p);
void apr_pool_destroy(apr_pool_t *p);
/* cleanup registration */
typedef apr_status_t (*cleanup_t)(void *data);
void apr_pool_cleanup_register(apr_pool_t *p,
    const void *data, cleanup_t plain_cleanup, ...);
```

Figure 6. Part of APR pools interface.

specified size of memory in a given region; `apr_pcalloc` further fills the newly allocated memory with zero.

A region can be cleared using `apr_pool_clear` or deleted using `apr_pool_destroy`; the runtime will operate on all its descendants. Moreover, APR enables to register cleanup functions on regions via `apr_pool_cleanup_register`. For example, the developer opens a file descriptor using `open` and registers a cleanup function that calls `close` to delete the file descriptor on a region. When the region is cleared or deleted, the runtime will trigger the cleanup function to close the file descriptor, so that it can avoid resource leaks. In this way, APR manages systems resources such as file descriptors using `open-close` similarly to memory in regions. Interested readers may refer to APR documentation for detailed information.

Now we describe the implementation details of RegionWiz. It consists of four phases: 1) call graph construction, 2) context cloning, 3) conditional correlation computation, and 4) post processing. It mostly follows the standard steps in cloning-based context-sensitive pointer analysis [40], with additional supports for heap cloning [36].

5.1 Call Graph Construction

The first phase constructs an initial context-insensitive call graph in a standard way as a basis for further computation. We built a back-end plug-in for the Phoenix compiler framework [31], and transparently inserted the plug-in into the compiler phase list to extract instructions of the intermediate representation (IR) for programs. Each instruction consists of destination operands, opcode, and source operands.

```
// time_t t = time(0);
    1 t142    = CALL &_time, 0
    2 _t      = ASSIGN t142
// struct tm * (*mytime)(const time_t *timer);
// mytime = localtime;
    3 _mytime = ASSIGN &_localtime
// int week = mytime(&t)->tm_wday;
    4 t143    = CALL _mytime, &_t
    5 t144    = ADD t143, 24
    6 _week   = ASSIGN [t144]*
```

In the example above, line 1 is a direct CALL instruction to function `time`; line 3 assigns a function pointer to variable `mytime`; line 4 indirectly invokes the function pointer and assigns `t143` with the resulting structure `tm`; to access its field `tm.tm_wday`, line 5 adds the pointer with offset 24, which is machine-dependent; line 6 deferences the value.

The algorithm for call graph construction is expressed as Datalog rules and solved using the bddbddb deductive database [24] over such IR instructions. Let I be the set of IR instructions and F be the set of functions. The resulting call graph is in the form $call : I \times F$, the set of call edges; $call(i, f)$ means that the target of instruction i (that should be a CALL instruction) is function f. RegionWiz computes $call$ from direct, indirect, and implicit calls.

The target function of a direct call can be simply extracted from the first source operand of the CALL instruction, such as line 1 that calls `time` in the example above.

An indirect call requires to resolve the first source operand of a CALL instruction to determine what functions the variable may refer to. To do so, RegionWiz estimates the set $vF : V \times F$ for each variable, where V is the set of variables. For an initialization instruction such as line 3 in the example above, it is straightforward to add $(\texttt{mytime}, \texttt{localtime})$ into set vF. RegionWiz further propagates function pointer values along variable assignments intraprocedually and call-return instructions (e.g., parameters and return values) interprocedurally, and iterates to add (variable, function) pairs into set vF until it converges.

An implicit call such as system callback requires expert knowledge. For example, if there is a call instruction i to function `apr_thread_create` with the entry function `foo` as its parameter, the system will create a thread that invokes `foo` at runtime. Thus, in addition to the direct call $(i, \texttt{apr\_thread\_create})$, RegionWiz also adds the implicit call $(i, \texttt{foo})$ into set $call$ for a more complete call graph. Current implementation supports such thread creation functions provided by Windows API, libc, and APR.

Moreover, RegionWiz identifies the main entry for a program (usually the `main` function in C) and performs a reachability analysis to prune the instructions in those functions that are never called directly or indirectly from the main entry.

5.2 Context Cloning

RegionWiz transforms the context-insensitive call graph $call$ to a context-sensitive call graph $cc : C \times I \times C \times F$ via cloning; $cc(c_0, i, c_1, f)$ is a call edge that instruction i in context c_0 calls function f in context c_1. The transformation first reduces strongly connected components in $call$ into single nodes, finds a topological order, and then numbers individual call paths as calling contexts, following the standard algorithm [40]. Each context number for a function represents a unique call path that reaches the function from the main entry. Since the number of contexts is exponential, RegionWiz stores the relation cc using a finite domain implementation in BuDDy [28], a binary decision diagram (BDD) package.

Now we have a context-sensitive call graph cc, where each call to function f in context c is identified as a unique pair (c, f). Thus, for each call to region creation functions (e.g., `apr_pool_create`) or object allocation functions (e.g., `apr_palloc`), (c, f) can represent a region or object instance. Our subsequent computation uses such pairs to identify regions and objects.

In addition, variable v is identified as (c, v), so that we can compute points-to set for each variable in individual contexts, such as $vR(c, v, rc, rf)$ for variable v in context c that may refer to region instance (rc, rf). The propagation for computing the set works as follows. For intraprocedural statements such as assignment $v_2 = v_1$, add (c, v_2, rc, rf) into vR if (c, v_1, rc, rf) is in vR. Interprocedurally, for CALL instruction i, (c_1, i, c_2, f) in cc, assuming v_1 is the k-th variable of instruction i in caller code and v_2 is the k-th parameter of the target function f in callee code, add (c_2, v_2, rc, rf) into vR if (c_1, v_1, rc, rf) is in vR; in addition, assuming v_3 is assigned with the return value of instruction i and v_4 is the source operand of a RETURN instruction in f, add (c_1, v_3, rc, rf) into vR if (c_2, v_4, rc, rf) is in vR. The computation iterates until it converges.

5.3 Conditional Correlation Computation

This phase is the core part of our analysis to compute the conditional correlation over regions and objects.

5.3.1 Effect Computation

The computation for the effects iterates as described in Section 5.2 for a whole program. It is usually the most time-consuming part.

For calls to region creation functions (e.g., `apr_pool_create`) we estimate vR, the set of regions that each variable may point to, and the subregion relation over regions.

- $vR : C \times V \times C \times F$ is the points-to relation for regions; $vR(c, v, rc, rf)$ means that variable v in context c may point to region (rc, rf).
- $subregion : C \times F \times C \times F$ is the subregion relation over regions; $subregion(rc_0, rf_0, rc_1, rf_1)$ means that region (rc_0, rf_0) may be a subregion of (rc_1, rf_1).

For calls to object allocation functions (e.g., `apr_palloc`) we similarly estimate vH, the set of objects that each variable may point to, and the ownership relation between regions and objects.

- $vH : C \times V \times C \times F$ is the points-to relation for objects; $vH(c, v, hc, hf)$ means that variable v in context c may point to object (hc, hf).
- $ownership : C \times F \times C \times F$ is the ownership relation between regions and objects; $ownership(rc, rf, hc, hf)$ means that region (rc, rf) may own object (hc, hf).

We further compute the set $heap$ on each store statement `x->f = y` that x can access y via field f. Since C is a weakly-typed language, we use offset values instead of symbolic names for fields.

- $heap : C \times F \times N \times C \times F$ is the set for heap over objects; $heap(c_0, f_0, n, c_1, f_1)$ means that object (c_0, f_0) contains a pointer at the offset n that can access object (c_1, f_1).

We do not compute a separate access relation; $heap$ is sufficient.

5.3.2 Inconsistency Computation

We filter candidate region pairs that have no subregion partial order, and verify the non-access property against each pair.

- $regionPair : C \times F \times C \times F$ is the set of region pairs that may have no subregion partial order; $regionPair(rc_0, rf_0, rc_1, rf_1)$ means that $(rc_0, rf_0) \not\leq (rc_1, rf_1)$.
- $objectPair : C \times F \times N \times C \times F$ is the resulting inconsistent object pairs; $objectPair(c_0, f_0, n, c_1, f_1)$ means that object (c_0, f_0) contains a possible dangling pointer at field offset n to object (c_1, f_1).

The computation of $regionPair$ is based on the conservative way to estimate parent regions described in Section 4.3. It is straightforward to compute the result $objectPair$ based on $regionPair$, $ownership$, and $heap$ according to (4.12) and (4.13).

5.4 Post Processing

As a static analysis tool may generate a large amount of warnings, it is necessary to process the reported warnings and aid developers to locate and inspect the suspicious code.

First, since the result $objectPair$ is over context-sensitive object pairs, the size is usually quite large because the object pairs can be inconsistent in many similar contexts. We condense context-sensitive object pairs to context-insensitive instruction pairs (I-pairs) for further inspection.

Besides, we rank reported warnings. RegionWiz does not compute def-use information, as we have discussed in Section 4.3, so it may report warnings on intra-region pointers that should be always safe. To filter them our current implementation applies one ranking heuristic: for an inconsistent object pair, if their owner regions never have the subregion relation, we rank them high in the result.

	KLOC	exe	brief description
rcc	37	1	RC compiler
apache 2.2.6	42	9	web server and utilities
freeswitch 1.0b1	109	1	telephony platform shell
jxta-c 2.5.2	114	1	P2P framework shell
lklftpd	5	1	FTP server
subversion 1.4.5	240	9	version control system

Figure 7. Benchmarks. The "exe" column lists the number of executables in each packages. The rcc package uses RC regions. Other software packages are based on APR, where the code size of APR ($\sim$ 200 KLOC) does not count.

	high-ranked (cause)	inconsistency (cause)
rcc	1 (1)	1 (1)
apache	1 (1)	0 (0)
lklftpd	2 (2)	2 (2)
subversion	21 (6)	9 (4)
total	25 (10)	12 (7)

Figure 8. Numbers of high-ranked warnings and inconsistencies, as well as their unique causes.

5.5 Limitations

RegionWiz supports pointers to pointers, since they are typical in C functions for retrieving newly created objects, e.g., `apr_pool_create`. It also handles unsafe typecasts including casts between integers and pointers, and uses low-level offset integers rather than symbolic field names for structures and unions. It is unsound for more complex pointer operations such as arithmetic.

RegionWiz tracks thread creations as implicit calls, but it may still miss some call edges due to other implicit callbacks from the operating systems or underlying libraries. It tracks function pointers but may still fail to resolve some call sites due to complex pointer operations or dynamic loading of shared libraries. The current Phoenix version does not emit all information generated by the front-end, so RegionWiz may miss some indirect call edges. These limitations will result in an incomplete call graph.

Besides, our post processing is unsound. Developers may focus on high-ranked warnings and miss lower-ranked inconsistencies.

6. Experiments

We have applied our prototype tool RegionWiz to six software packages, as listed in Figure 7. Among them rcc uses RC regions, while others use APR pools. All packages in the experiments were the latest stable releases (if possible), so we did not expect there would be many inconsistencies. Development versions are more likely to have serious inconsistencies; interested readers can search their repository logs for fixes.

RegionWiz reported 230 warnings of instruction pairs for inconsistent objects and ranked 25 of them high (10 unique causes). We examined them and found 12 inconsistencies (7 unique causes), as shown in Figure 8.

6.1 Case Study

Figure 9 illustrates an inconsistency case between a hash table and an iterator in Subversion. The hash table should have a longer lifetime than the iterator. On the contrary, in Figure 9(a) the hash table resides in subregion `subpool`; in Figure 9(b) the iterator `hi` is allocated in the parent `pool`, which is inconsistent. Though the inconsistency does not lead to crash, the longer-than-necessary lifetime is a potential memory leak.

```
/* libsvn_subr/xml.c:svn_xml_make_open_tag_v */
apr_pool_t *subpool = svn_pool_create(pool);
apr_hash_t *ht = svn_xml_ap_to_hash(ap, subpool);
svn_xml_make_open_tag_hash(str, pool, ..., ht);
svn_pool_destroy(subpool);
```

(a) A hash table `ht` is allocated in `subpool`.

```
/* libsvn_subr/xml.c:svn_xml_make_open_tag_hash */
for (hi = apr_hash_first(pool, ht); hi; ...)
```

(b) Retrieve an iterator `hi` for the hash table `ht`.

```
/* apr/tables/apr_hash.c: apr_hash_first */
if (pool)
    hi = apr_palloc(pool, sizeof(*hi));
else
    hi = &ht->iterator;
hi->ht = ht;
```

(c) If the given region `pool` is not null, a new iterator `hi` is allocated in `pool`; otherwise, `hi` uses a field of `ht` intrusively. The iterator can access the hash table via `hi->ht`.

Figure 9. An inconsistency between a hash table and its iterator. The iterator `hi` allocated in parent `pool` holds a possible dangling pointer `hi->ht` to the hash table `ht` allocated in `subpool`.

Note that the code in Figure 9(a) allocates the hash table in a subregion and deletes the subregion before exit; the developers should have intended to free all temporary memory, but the intention fails due to `hi` that inconsistently resides in `pool`. The developers might argue that using a separate subregion could be more likely to be thread safe (see Section 6.4 for further discussion), but `hi` is allocated in the parent `pool`, which contradicts the argument. To fix the bug, the call to `svn_xml_make_open_tag_hash` in Figure 9(a) can pass `subpool` instead of `pool`. Alternatively, the call to `apr_hash_first` in Figure 9(b) can pass null instead of `pool` as the first parameter; in Figure 9(c) iterator `hi` will share the same region as the hash table.

Another type of inconsistency relates to strings. The warning generated for rcc is such a case that an object holds a pointer to a string while their owner regions have no subregion partial order. We omit the code for brevity since it involves about 10 functions. The inconsistency does not lead to crash because the two owner regions are never deleted. However, the object should not expect client code never deletes the region that owns the string; a better way could be to duplicate the string in the object's owner region.

Temporary inconsistencies that violate the consistency semantics within a scope of code are "benign". Figure 10 illustrates an example. Object `lock` is allocated in `pool`, and `lock->set` may be assigned with a temporary hash table allocated in `subpool` via a call to `apr_hash_make`. Our tool reported a warning for the case since it violates the semantics defined in Section 3, though later `lock->set` is reassigned with `associated->set` before `subpool` is deleted.

A more precise analysis with path sensitivity may help to eliminate the temporary inconsistency. Particularly, the analysis may have to prove that object `lock` is *always* allocated in a parent region `pool` in either branch of `write_lock`, that `lock->set` is set to a hash table allocated in a subregion `subpool` under the condition P that both `levels_to_lock!=0` and `associated` hold, that `lock->set` is reassigned with `associated->set` under the condition Q that `associated` holds (independent of `levels_to_lock`), and that P implies Q.

Generally, temporary inconsistencies make code that involves complicated branch conditions more error-prone. Developers have to carefully handle assignments of objects of different lifetimes in various code branches. Nevertheless, a better way to organize

```
/* libsvn_wc/lock.c:do_open */
svn_wc_adm_access_t *lock;
apr_pool_t *subpool = svn_pool_create(pool);
if (write_lock)
    lock = adm_access_alloc(..., pool);
else
    lock = adm_access_alloc(..., pool);
if (levels_to_lock != 0) {
    if (associated)
        lock->set = apr_hash_make(subpool);
    if (associated) { ...
        lock->set = associated->set;
    }
}
if (associated)
    lock->set = associated->set;
svn_pool_destroy(subpool);
```

Figure 10. A (slightly simplified) temporary inconsistency example. Object `lock` is allocated in `pool` (in either branch of `write_lock`). Its field `lock->set` is temporarily assigned with a hash table, which is allocated in `subpool`; the field is later reassigned with `associated->set`.

code could be updating object fields only when necessary to avoid temporary inconsistencies and reduce the risk of crashes and leaks.

In our experience, region lifetime inconsistency usually involves several functions and deep call paths, thus it requires a precise interprocedural analysis as we have employed.

6.2 False Warnings

Since our analysis is mostly flow-insensitive, we expected there would be corresponding false warnings. We looked 205 lower-ranked warnings. Most of them are false; we found 1 temporary inconsistency there. So our simple heuristic described in Section 5.4 effectively pruned most false warnings. However, to eliminate the false warnings (3 unique causes) in high-ranked ones we found that all of them require extra effort in addition to flow sensitivity.

Here is an example.

```
1 /* libsvn_subr/error.c:make_error_internal */
2 if (child)
3     pool = child->pool;
4 else
5     if (apr_pool_create(&pool, NULL))
6         abort();
7 new_error = apr_pcalloc(pool, ...);
8 new_error->child = child;
```

At first glance, line 5 creates a separate region `pool` and line 7 allocates object `new_error` in the region; at line 8 the object seems to hold a possible dangling pointer to `child`. In fact, `pool` is a separate new region only if `child` is null; in this case pointer `new_error->child` is assigned with null. Otherwise `pool` refers to `child->pool` at line 3; because `new_error` shares the same region with `child`, pointer `new_error->child` is intra-region and always safe. In either case, the region lifetime is consistent.

To eliminate such false warnings, we may apply heuristics similar to lock-unlock pairs computation in RacerX [15] for race detection, or employ a path-sensitive analysis [12, 43] to track branch conditions. We leave it as future work.

6.3 Quantitative Results

Figure 11 shows our experimental results in detail. For each executable, we measure the analysis time, the total numbers of regions $\mathbb{R}$ and normal objects $\mathbb{H}$, and the sizes of the three relations subregion, ownership, and heap (access). We also count the numbers of region pairs that have been verified ($\mathbb{R}$-pairs) and suspicious ob-

	time	$\mathbb{R}$	$\mathbb{H}$	sub.	own.	heap	$\mathbb{R}$-pair	$\mathbb{O}$-pair	I-pair	high
rcc	19m21s	10	2536	9	1577	746940	70	1	1	1
ab	49s	11	111	10	53	24	92	0	0	0
htdbm	51s	3	15	2	12	10	4	0	0	0
rotatelogs	51s	3	21	2	17	21	4	0	0	0
httxt2dbm	56s	4	80	3	27	45	8	0	0	0
htcacheclean	1m21s	13	242	12	162	230	120	0	0	0
htdigest	1m27s	3	293	2	264	315	4	0	0	0
htpasswd	1m50s	3	406	2	338	343	4	0	0	0
flood	2m06s	6	324	5	62	97	24	0	0	0
httpd	34m04s	19	4546	18	2341	2273	319	410	9	1
freeswitch	14m55s	20	3174	46	3065	2499	360	456	4	0
jxta-c	58m24s	17	5007	16	27	10	256	0	0	0
lklftpd	2m34s	7	622	6	622	565	34	6	2	2
diff	16m29s	427	1941	680	64833	4274	181477	260	13	1
diff3	19m30s	424	1865	535	25135	2766	178930	189	13	1
diff4	21m24s	425	1877	538	25147	2781	179767	190	13	1
svndumpfilter	44m46s	6517	28378	6870	2069908	29153	42458253	4072	15	2
svnadmin	53m20s	7274	31620	8326	3881275	39133	52896514	7741	23	2
svnlook	1h00m57s	8194	35638	8760	4846261	37928	67125232	5289	23	2
svnsync	1h21m43s	8123	36589	10863	5003865	62491	65965730	7896	24	3
svnserve	15h09m41s	47480	195255	93771	158244795	314511	2254148642	43874	57	3
svn	25h59m53s	53754	238521	542402	897921834	280671987	2889375908	134798	31	6

Figure 11. Experimental results. All experiments were conducted on a server with Intel Xeon 2.0 GHz and 32 GB of RAM. The "time" column lists the analysis time for each executable. "$\mathbb{R}$", "$\mathbb{H}$" are the numbers of regions and normal objects, respectively (Section 5.2). "sub.", "own.", and "heap" are the sizes of the $subregion$, $ownership$, and $heap$ relations, respectively (Section 5.3.1). "$\mathbb{R}$-pair" is the number of region pairs to be verified and "$\mathbb{O}$-pair" is the number of reported inconsistent object pairs (Section 5.3.2). "I-pair" is the number of context-insensitive instruction pairs of $\mathbb{O}$-pairs and "high" is the number of high-ranked I-pairs (Section 5.4).

ject pairs that have been detected ($\mathbb{O}$-pairs). The context-insensitive instruction pairs (I-pairs) and high-ranked pairs are useful for further inspection. The time for call graph construction does not count since they are relatively small compared to the analysis time.

As reported [24], BDD variable order can greatly affect efficiency of bddbddb. We randomly tried a few orders and picked a not-so-bad one. 18 out of 22 experiments can finish within an hour. However, as calling contexts grow, the numbers of objects ($\mathbb{R}$ and $\mathbb{H}$) increase fast and lead to a large amount of relations and region pairs to be verified. The most time-consuming experiment, svn, takes more than one day to finish, which at first we thought could not produce a result. Notably, it yields more than 2 billions $\mathbb{R}$-pairs and thus complicates the computation. The result suggests that reducing calling contexts is an important factor to improve scalability. While RegionWiz uses call paths as contexts, we are investigating other precise context-sensitivity for C programs that yields a smaller number of contexts.

6.4 Discussion

It is arguable that few inconsistencies imply good code practice. Our tool reported few to none inconsistencies for Apache that manages regions in an elaborate way [23], while there were dozens of warnings for Subversion. As an example, let's compare XML parser creation API implementations provided by the two packages. Both implementations return a newly created parser object, based on the popular Expat library (in Expat `XML_ParserCreate` creates a new instance and `XML_ParserFree` frees it, similar to `open-close` for file descriptors).

As shown in Figure 12(a), `apr_xml_parser_create` in Apache allocates the parser object from the given `pool`, creates an Expat instance, and registers a cleanup function `cleanup_parser` (code not list here). If client code deletes `pool`, the runtime will trigger the cleanup function that invokes `XML_ParserFree` to free the Expat instance.

```
APU_DECLARE(apr_xml_parser *)
apr_xml_parser_create(apr_pool_t *pool) {
    apr_xml_parser *parser = apr_pcalloc(pool, ...);
    parser->xp = XML_ParserCreate(NULL); ...
    apr_pool_cleanup_register(pool, parser,
                              cleanup_parser, ...); ...
    return parser;
}
```

(a) `apr_xml_parser_create` in Apache.

```
svn_xml_parser_t *
svn_xml_make_parser(..., apr_pool_t *pool) {
    svn_xml_parser_t *svn_parser;
    apr_pool_t *subpool;
    XML_Parser parser = XML_ParserCreate(NULL); ...
    /* ### we probably don't want this pool;
           or at least we should pass it
       ### to the callbacks and clear it periodically. */
    subpool = svn_pool_create(pool);
    svn_parser = apr_pcalloc(subpool, ...); ...
    return svn_parser;
}
```

(b) `svn_xml_make_parser` in Subversion.

Figure 12. Two XML parser creation API implementations.

On the other hand, Figure 12(b) shows the Subversion counterpart `svn_xml_make_parser`. It does not register any callback for `XML_ParserFree`, so to delete `pool` in client code will lead to leaks (though they provide a separate parser deletion API). More importantly, it creates `subpool` of the given `pool`, and allocates the parser object in `subpool` rather than in `pool` as Apache code does. The region usage is debatable. Interestingly, as shown in Figure 12(b), the developers themselves put a comment that concerns the issue in the code.

First, as we have discussed in Section 1, one advantage of regions is that developers can gather relevant objects in the same region and batch allocation and deallocation for better performance. However, because `svn_xml_make_parser` allocates the object in a separate subregion, the memory allocator of which is independent, client code loses fine-grained control and cannot take the performance advantage.

Moreover, since the memory allocator of the subregion is independent, one may argue that the usage provides better thread safety and concurrency, as some developers responded; the parser can use `subpool` independent of `pool` that might be shared with another thread without locks. However, it is *not* the XML parser API's responsibility to take care of the multi-threading issue; the parser does not create any thread. It is the client code that should manage threads and regions because only the client code has the knowledge to determine whether `pool` is shared among threads and which region the parser object should reside.

Because the parser in `svn_xml_make_parser` is created in `subpool`, any object that is created in `pool` and that can access the parser involves an inconsistency, such as object `loggy` in the following code.

```
/* libsvn_wc/log.c:run_log */
struct log_runner *loggy = apr_pcalloc(pool, ...);
parser = svn_xml_make_parser(..., pool);
...
loggy->parser = parser;
```

RegionWiz reports a warning for every such use.

7. Related Work

Regions have long been an underlying abstraction for memory management [5], such as in ML [38, 1], real-time Java [8, 10, 11], and C [25]. The abstraction is also useful for detecting memory errors [20, 13]. Some C dialects support regions via language extensions. Cyclone [19] enables developers to add region annotations in a restricted discipline to source code. It infers regions and performs type checking to statically detect dangling pointers. C@ [16] and RC [17] support more flexible region usages. They maintain reference counts for regions at runtime; a region will not be deleted if it is still referenced by inter-region pointers from outside. While useful, dynamic techniques for safe regions do not fix bugs generally; objects still reside inconsistently in regions, and resources in the regions cannot be reclaimed. RegionWiz can find these bugs before deployment and usually provide a higher coverage.

Correlations that capture consistency constraints between two sets of objects pervade in software systems. MUVI [30] infers multi-variables that should be updated consistently, e.g., a buffer and its length, and detects violations in source code. Another notable example is the correlation between locks and memory locations for race detection, which is an inspiration of our work. RacerX [15] employs heuristics to detect races and dead locks in millions lines of systems code. LOCKSMITH [37] infers the correlation for C programs and formalizes the problem as a constraint resolution. Chord [33, 34] further introduces a disjoint reachability analysis for Java; proving race-free is formalized as a conditional must not aliasing problem based on analyzing the heap. Ownership types for real-time Java [8, 11], based on the concept of encapsulation, can also be used to prove conditional correlation for region lifetime consistency, though not necessary [34]. Besides, they infer ownership between objects; our ownership relation is between regions and objects, which is more natural for region interfaces such as RC regions and APR pools.

RegionWiz adopts a cloning-based context-sensitive pointer analysis [40] with heap cloning [36] to distinguish different call paths to region creation and object allocation sites. Most previous cloning-based work [40, 4, 39] did not clone the heap; they consider these sites in a context-insensitive way. We feel that heap cloning is necessary for our analysis, but the numbers of contexts using call paths are too large in some cases. The Chord race detector [33] uses object sensitivity [32] for a smaller number of contexts in Java programs. We are investigating more appropriate context sensitivity for C programs. In addition to the cloning-based approach, we may also adopt other context-sensitive pointer analysis with heap cloning [35, 26].

Besides, our analysis algorithm is mostly solved by bddbddb [24], the BDD-based deductive database. Since BDD is often a key component to achieve scalability in context-sensitive analysis [27, 40, 44], we are interested in a parallel BDD implementation that can help to build more efficient analysis tools.

To achieve a lower false-positive rate, we may employ more precise path-sensitive analysis techniques, such as IPSSA [29] and SAT-based approaches [42, 9].

8. Conclusion

Region-based memory management is widely used in systems software, but it is prone to lifetime consistency that can cause dangling pointers. We formalize the problem based on the concept of conditional correlation, which may be of independent interest. We have built a prototype tool RegionWiz to detect such inconsistencies in source code and applied it to real-world applications. It is useful to find inconsistency bugs in practice.

We would like to further investigate more precise and scalable analysis techniques for computing conditional correlation. Besides, we are working on extensions to support analysis of open programs such as libraries. Our future work also includes to study other conditional correlations, such as locks and memory locations.

Acknowledgments

We thank David Gay for insightful comments on the RC compiler and John Whaley for useful discussions about bddbddb. We would like to thank Frans Kaashoek, Zhengwei Qi, Linchun Sun, Ming Wu, Jianian Yan, and the anonymous reviewers for their valuable feedback on earlier drafts of this paper. Xi Wang and Xiaoge Wang are supported in part by National High-Tech Research and Development Plan of China under Grant No. 2006AA01Z198 and by Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995.

[2] R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1995.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *International Conference on Software Engineering (ICSE)*, 2005.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[6] L. Birkedal, M. Toft, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1996.

[7] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, 1988.

[8] C. Boyapati, A. Sălcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[9] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[10] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *International Symposium on Memory Management (ISMM)*, 2004.

[11] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[12] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[13] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[14] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[15] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[16] D. Gay and A. Aiken. Memory management with explicit regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[17] D. Gay and A. Aiken. Language support for regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[18] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1996.

[19] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[20] B. Hackett and R. Rugina. Region based shape analysis with tracked locations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[21] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software — Practice and Experience*, 20(1):5–12, 1990.

[22] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[23] N. Kew. *The Apache Modules Book: Application Development with Apache.* Prentice Hall PTR, 2007.

[24] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2005.

[25] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[26] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[27] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[28] J. Lind-Nielsen. BuDDy: A binary decision diagram package. `http://buddy.sourceforge.net/`.

[29] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

[30] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[31] Microsoft. Phoenix compiler framework. `http://research.microsoft.com/phoenix/`.

[32] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[33] M. Naik and A. Aiken. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[34] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[35] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis Symposium (SAS)*, 2004.

[36] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Importance of heap specialization in pointer analysis. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2004.

[37] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[38] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1994.

[39] X. Wang, Z. Guo, X. Liu, Z. Xu, H. Lin, X. Wang, and Z. Zhang. Hang analysis: Fighting responsiveness bugs. In *ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2008.

[40] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[41] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management (IWMM)*, 1992.

[42] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2005.

[43] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[44] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.