Efficient Incrementalized Runtime Checking of Linear Measures on Lists

Alex Gyori*, Pranav Garg†, Edgar Pek‡, P. Madhusudan*
*University of Illinois at Urbana-Champaign
†Amazon[1]
‡Adobe Systems Inc.[1]
Email: gyori@illinois.edu, prangarg@amazon.com, pek@adobe.com, madhu@illinois.edu

***Abstract*—We present mechanisms to specify and efficiently check, at runtime, assertions that express structural properties and aggregate measures of dynamically manipulated linked-list data structures. Checking assertions involving the structure, disjointness, and aggregation measures on lists and list segments typically requires linear or quadratic time in the size of the heap. Our main contribution is an *incrementalization instrumentation* that tracks properties of data structures dynamically as the program executes and leads to orders of magnitude speedup in assertion checking in many scenarios. Our incrementalization incurs a constant overhead on updates to list structures but enables checking assertions in *constant time, independent of the size of the heap*. We define a general class of functions on lists, called *linear measures*, which are amenable to our incrementalization technique. We demonstrate the effectiveness of our technique by showing orders of magnitude speedup in two scenarios: one scenario stemming from assertions at the level of APIs of list-manipulating libraries and the other scenario stemming from providing dynamic detection of security attacks caused by malicious rootkits.**

I. Introduction

Writing and checking assertions is one of the most used techniques for detecting errors and providing information about fault locations [10], [13], [21], [32]; assertions are used widely in production code [11], [12], [18], [21] (e.g., a study showed that there are more than a quarter million assertions in the Microsoft Office suite [21]). The Eiffel system pioneered the systematic usage of assertions for method contracts and led to wider adoption in mainstream programming languages. The JML notation [1], the Spec# language [7], and the Code Contracts system [6] are examples of specification languages that were influenced by Eiffel [28]; Code Contracts were used for writing specifications and testing code within Microsoft. Assertions are simple to write and popular as programmers use them for testing; they are the most available form of specification and are leveraged for more sophisticated analyses, such as for unit testing [32], [37], [39], test-input generation [4], [9], [15], [26], [36], and regression testing [29], [33].

Assertions express the *expectation* of properties believed to be an invariant of the program at the respective program point, and can be written by a programmer or automatically mined from test data [17]. Checking such assertions at runtime, either during testing or in deployed production code, benefits several applications. One application is tracking likely invariants to use them in downstream regression testing and verification techniques [17], [31]. Another application is to mine likely invariants from *normal* runs of the program, and checking them at runtime in production code to find security breaches [16]. An example of such an application is a technique that detects malicious kernel-level rootkits by generating invariants during a training period and monitoring for their violations [5]. The invariants in this domain typically involve properties over data-structures (such as the linked lists in the kernel that store sets of processes for process accounting and scheduling).

Assertions are useful in testing during the software development process [40], where assertions express preconditions or invariants about the developers' own code or an external library. The assertion violations help detect, debug, and fix errors early. Mainstream programming languages lack standard assertion logics for the structure of the heap, and programmers often write such assertions by writing *procedures* that check properties. For instance, in Java, programmers write so-called RepOk methods for checking representation class invariants during testing [25]. These dynamic checks require traversing the entire data-structure (lists, in our case), and therefore they can be expensive to evaluate. Our goal is to provide incrementalizable ways of evaluating disjointness of lists and certain functions on lists, which can lead to significant speed up in checking properties of lists at runtime.

In this paper, we study the problem of expressing and efficiently checking assertions on lists and list-segments while they get dynamically manipulated by an imperative program making destructive updates to the heap. The assertions we support express properties of lists and list-segments, including their disjointness, and *linear measures* on them. Linear measures are a class of functions that map lists and list-segments to a domain that satisfies certain linearity, compositionality, and incrementalizability properties, and include a variety of functions like the lengths of lists or list-segments, the multiset of keys they contain, the set of locations that define them, etc. Our assertion logic supports atomic predicates that check for structural properties involving whether two pointers form a list-segment ($lseg(y, z)$), disjointness properties checking whether the locations defining two list segments are disjoint (like `Disjointlseg`$((x, z), (y, z))$), and properties of linear measures of lists and list-segments.

[1]This work is independent of the author's work at his affiliated institution and was done while the author was a student at the University of Illinois.

The main contribution of this paper is a procedure that continually and incrementally tracks properties of the heap (with a small overhead for each update of the heap) such that assertions like the one above can be checked quickly, in constant time, without traversing the heap. The key idea is to maintain summaries of large regions of the heap and incrementally maintain the summaries with each update to the heap. We show that structural properties of lists and list-segments as well as any linear measure on them can be effectively and incrementally maintained with low overhead using summaries involving very few regions.

We evaluate our technique using two experiments. The first one consists of small programs having API methods from the GLib library [20], which model library calls that maintain lists, and client code, which uses these libraries and contains very intensive assertion checking. We perform experiments that stress-test these programs on lists of varying size, and show that the incrementalized checking of assertions is significantly more efficient, and incurs a *constant overhead*. The second experiment involves an application of runtime checking to detect malicious rootkits by dynamically checking the integrity of data structures in the kernel. In this application, we wish to track whether the contents of two data-structures are the same by tracking the sum of their hashes for each list, which is a linear measure. We show that incrementalized checking of assertions using region summaries causes negligible overhead while checks on the concrete lists incur significant overhead.

II. Motivating Example

In this section we present a motivating example in the form of an assertion and illustrate general concepts relevant to the type of assertions our technique supports. Consider the following assertion:

```
lseg(x,nil) ∧ lseg(y,z) ∧  Disjointlseg((x,nil), (y,z))
            ∧ length((x,nil)) > length((y,z))
```

The above assertion expresses that the program variable x points to a list (more precisely, a nil-terminated list segment) and y to z forms a list-segment; the list x and list-segment (y, z) are disjoint, and the length of the list x is greater than the length of the (y, z) segment. One such heap satisfying these properties is illustrated in Figure 1. The simplest runtime check on the heap for the above property would take at least linear time on the length of the lists involved. When the sizes of the data structures get large (several thousand nodes), linear-time algorithms cause nontrivial overhead, and hence the runtime checking of such data structures does not scale, whether checks are written by the programmer or automatically generated from specifications.

Linear Measures: Our assertion logic is parameterized by a class of functions that map list-segments into a domain D; we identify general conditions (linear measures) under which our technique can incrementally evaluate the functions.

A linear measure is a function f that maps list-segments to a domain D, where D has an operator $\oplus$ that forms a monoid, i.e., is associative and there is an identity element.

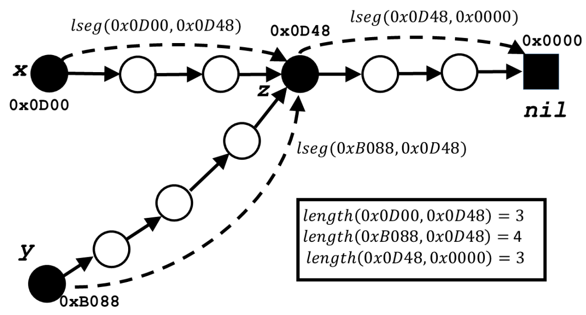

Fig. 1: Pertinent locations are depicted as solid nodes, the regions $\mathcal{R}$ are depicted using dashed edges. Linear measures of the regions in $\mathcal{R}$ are as shown in the bottom-right box.

Furthermore, we require the *linearity property*, i.e., for any two "consecutive" list-segments ls_1 and ls_2, $f(ls_1 \cdot ls_2) = f(ls_1) \oplus f(ls_2)$, so that the function on a list-segment can be computed from its value on any partition of the list-segment into two disjoint list-segments. In addition, we require the existence of two computable functions that will help incrementalize computation—one that computes the measure for single-element lists, and the second, *contracting* function, that computes, given a list-segment ls, the measure of the suffix of ls, obtained by removing the first element, solely from the measure of the list segment ls and the element removed. We show that a variety of interesting measures, including lengths of list-segments, the keys they contain, the heap locations that define them, the sum of hashes of the elements of the list, etc., can all be expressed as linear measures.

Region Summaries: The information that we continually track is a set of *regions* R, information on how these regions are connected, and *summaries* that capture for every region its linear measure. Figure 1 depicts a concrete heap forming several list segments and the corresponding regions overlaid. The regions of the heap of interest are list-segments that connect *pertinent* nodes; pertinent nodes are depicted using solid nodes and regions are depicted using dashed arrows. The pertinent nodes include (a) the locations pointed to directly by the program's pointer variables, and (b) the *first* locations at which list-segments *merge* (the node pointed to by z would be pertinent even if z would not point to it because it is the first location where two lists merge). Given these pertinent nodes, we track the precise reachability relations between them: for each pertinent node p, we track the *next* pertinent node q that is reachable when following the next-pointer from p (shown by the dashed arrows). These list-segments define regions in R, and for each of the regions, we track explicitly their linear measure. Note that the number of pertinent nodes is a constant if the number of program pointer variables pointing to list nodes is a constant, and hence so is the number of regions and the number of linear measures that are tracked.

When the program executes and modifies the heap, most of the summarized regions are unaffected, and all the program's actions are around the locations pointed to by the program variables. For any operation, we can incrementally update the regions as well as the linear measures on the regions.

A key property of the regions we track is that the *disjointness* of any two list-segments can be computed just from the information stored in the region summary. This allows us to check disjointness of list-segments efficiently. Furthermore, by the linearity property of linear measures, we can also compute the linear measure for any list segment (by aggregating the linear measures of all list-segment regions comprising it using $\oplus$). Consequently, the atomic predicates in any assertion can be efficiently computed from the regions we track.

Note that our technique gives fast assertion checking by evaluating the assertion using the region summaries we maintain, but introduces an overhead for each operation on list nodes to update the region summaries. Consequently, our technique would be most beneficial mainly when the sizes of the lists are large and the number of changes to the list is small between assertion checks.

One typical scenario consists of a framework where one has *library code* that implements low-level manipulation of lists, and *client code* calling such libraries to implement a higher-level functionality. Both the library code and the client code could contain programmer-written assertions about the lists in the heap. In such a setting, our scheme would maintain the region summaries of all list nodes globally, and assertions in the client code and library code are transformed to checks using these region summaries. All manipulations of the lists within the library methods are *instrumented* to incrementally maintain region summaries. Note that the client code (which does not manipulate lists directly) is left uninstrumented (hence causing no overhead).

III. Linear Measures

We define linear measures as the class of functions from *list segments* to an arbitrary countable domain D. We will view linear predicates as linear measures that map to the Boolean domain $\mathcal{B}$, and treat them uniformly.

We now define list-segments, independent of the heaps they occur in. We fix a set of locations Loc with a special location `nil` $\in Loc$. We also fix a *data-domain Data*. A list-segment is a *finite sequence of locations* $(l_0 l_1 \dots l_n, key)$, where all locations except the last must be non-nil, and a function $key : \{l_0 \dots l_{n-1}\} \to Data$ associating locations to the data they hold. For any list-segment $ls = (l_0 l_1 \dots l_n, key)$, we refer to l_0 as the head ($head(ls)$) and to $l_1 \dots l_n$ as the tail ($tail(ls)$) of the list-segment.

Two list-segments $(l_0 l_1 \dots l_m, key)$ and $(l'_0 l'_1 \dots l'_n, key')$ are disjoint if $\{l_0, \dots, l_{m-1}\} \cap \{l'_0, \dots, l'_{n-1}\} = \emptyset$. Given two disjoint list-segments $ls_1 = (l_0 l_1 \dots l_m, key_1)$ and $ls_2 = (l'_0 l'_1 \dots l'_n, key_2)$ where $l_m = l'_0$, we define their concatenation to be $ls_1 \cdot ls_2 = (l_0 \dots l_m l'_1 \dots l'_n, key)$ where key is the union of the two functions key_1 and key_2.

A linear measure is a function from list-segments into a domain D. Let LS denote the set of all list-segments with keys mapped to the data-domain $Data$.

Definition 1 (Linear Measures). *A linear measure on list-segments over a data-domain Data is a function $f : LS \longrightarrow D$, where D is an arbitrary (countable or finite) domain, and D is equipped with a computable function $\oplus : D \times D \to D$ and*

- Assoc: *$\oplus$ is associative: for every $d_1, d_2, d_3 \in D$, $(d_1 \oplus d_2) \oplus d_3 = d_1 \oplus (d_2 \oplus d_3)$.*
- Id: *There is an element d_0 of D such that it is the identity with respect to $\oplus$ (i.e., $d \oplus d_0 = d_0 \oplus d = d$, for every $d \in D$). Furthermore, for every empty list segment $ls = (l_0, key)$, $f(ls) = d_0$.*
- Lin: *If $ls_1, ls_2 \in LS$ and $ls_1 \cdot ls_2$ is defined, then $f(ls_1 \cdot ls_2) = f(ls_1) \oplus f(ls_2)$.*
- Sing: *Let singleton be the function that maps every singleton list-segment $ls = (l_0 l_1, key)$ to $f(ls)$. Then singleton is computable.*
- Con: *$contract : D \times Loc \times Data \to D$ is a* contraction function *such that for any list-segment $ls = (l_0 l_1 \dots l_n, key)$ with $n \geq 1$, if $ls' = (l_1 \dots l_n, key \downarrow \{l_1, \dots, l_{n-1}\})$ is the suffix of ls obtained by removing the first element, then $f(ls') = contract(f(ls), l_0, key(l_0))$* □

The first two conditions above require the co-domain $(D, d_0, \oplus)$ be a monoid. The third condition is the linearity condition that requires that f applied to a list-segment evaluates to the sum (by $\oplus$) of its applications to the list segments that constitute it.

The Singleton (Sing) condition requires that the measure be computable for singleton lists. Along with the third condition, it follows that the measure is computable for any list-segment (by using the value on each singleton list-segment composing it and using $\oplus$ to combine them). The contraction property (Con) says that the measure on the suffix of a list-segment starting from the second node can be computed from the measure of the whole list-segment and the first node and the key stored in it. Note that this computation is independent of the list-segment suffix— we require that we are able to compute the measure on the extended list only from the measure on the list-segment and the first location and the key stored in it. The computability of the measure on singleton list-segments, the computability of $\oplus$, and the computable contraction function will play a crucial role in the incremental computation we build.

We use $(D, d_0, \oplus, singleton, contract)$ to denote the domain of the linear measure. We will next present several examples of linear measures.

Example 1 (Multiset of keys). *Consider the function Keys that returns the* multiset *of keys stored in a list-segment ls. This function is a linear measure: we can define $\oplus$ as multiset union $\cup$, $d_0 = \emptyset$, satisfying the associativity and identity conditions. Furthermore, clearly the linearity property above is satisfied. Every singleton list $ls = (l_1 l_2, key)$, $Singleton(ls) = Keys(ls) = \{l1.key\}$ is computable. For any list-segment ls and $k \in \mathbb{Z}$, we can define $contract(S, u, k) = S \setminus \{k\}$ and this computes the measure on the suffix of the list-segment whose measure is S. Note that defining f to be the* set *of keys (as opposed*

$x_s \in SVar$ $\qquad c_s \in Scalar$ Constant

$x_p \in PVar$

$op \in Scalar$ Operation $\qquad pred \in Scalar$ Predicate

$$\begin{array}{rl} e_S ::= & x_S \mid c_S \mid e_S\ op_S\ e_S \mid x_P \rightarrow key \\ e_P ::= & x_P \mid \texttt{nil} \mid x_P \rightarrow next \mid \texttt{malloc}_{\texttt{P}}() \\ e_b ::= & e_S == e_S \mid e_P == e_P \mid e_S\ relop\ e_S \mid e_b \wedge e_b \mid \neg e_b \\ P ::= & x_S := e_S \mid x_P := e_P \mid x_P \rightarrow key := e_S \mid \texttt{free}(x_P) \\ & \mid x_P \rightarrow next := x_P \mid \texttt{skip} \\ & \mid \texttt{if}\ e_b\ \texttt{then}\ P\ \texttt{else}\ P\ \texttt{fi} \\ & \mid \texttt{while}\ e_b\ \texttt{do}\ P\ \texttt{done} \mid P;P \mid \epsilon \end{array}$$

Fig. 2: Programming Language Syntax

to multiset) stored in a list-segment will not form a linear measure, as an appropriate contract function does not exist.

Example 2 (Heaplet)**.** *Consider the function Heaplet that returns the set of addresses (locations) that form the list-segment. This function is also a linear measure: we can define* $\oplus$ *as union* $\cup$ *and* $d_0 = \emptyset$. *For any singleton list* $ls = (l_1 l_2, key)$, $Singleton(ls) = Heaplet(ls) = \{l1\}$ *is computable. Further, clearly the linearity property is satisfied, and for any location* u, *and* $k \in Data$, *we can define* $contract(H, u, k) = H \setminus \{u\}$.

Example 3 (Length)**.** *Consider the linear measure* $Length : LS \rightarrow \mathbb{N}$ *that maps list-segments to the length of the segment. This function is a linear measure and we can define* $D = \mathbb{N}$, $\oplus$ *as* $+$, *with* $d_0 = 0$, *and define* $contract(l, u, k) = l - 1$. *For any singleton list* $ls = (l_1 l_2, key)$, $Singleton(ls) = Length(ls) = 1$ *is computable.*

Example 4 (Counting)**.** *Let P be a predicate on keys. Consider the linear measure CountP that maps list-segments to the* number *of keys stored in the segment that satisfy P. This function satisfies the linearity property and we can define* $D = \mathbb{N}$, $\oplus$ *as* $+$, *with* $d_0 = 0$, *and define* $contract(l, u, k) = l - 1$ *if* $P(k)$ *holds and* l *otherwise. For any singleton list* $ls = (l_1 l_2, key)$, $Singleton(ls) = CountP(ls)$, *which is* 1 *if* $P(l1.key)$ *holds, and* 0 *otherwise, and is computable.*

For instance, in an explicit information flow setting, we may be tracking tainted data, and taintedness can be a property of the key. The above then counts the number of tainted nodes in list-segments, and using this count, we can determine whether the list-segment has any tainted nodes. Note that keeping track of just whether list-segments are tainted or not will not be linear, as there is then no well-defined contract function.

IV. Programs and Assertion Logic

In this section we briefly describe our *C-like* programming language syntax, semantics, and the assertion language.

A. Programming Language Syntax and Semantics

Figure 2 presents the syntax of our programming language. The programming language has scalar expressions, consisting of scalar variables, scalar constants, binary scalar operations, such as +, -, *, and pointer dereferencing for scalar fields; pointer expressions consist of pointer variables, a special constant `nil`, and pointer dereference. Boolean expressions consist of standard relational operators over scalars, equality check over scalar and pointer expressions, conjunction, and negation. Programs are sequential compositions of scalar and pointer assignments to variables and pointer fields, `if-then-else`, `while`, `skip`, `free`, and `malloc`; programs have list-typed pointers, having a next pointer and a scalar key. We assume the program is type-safe, in the sense that scalar variables do not point to locations nor pointer variables to scalars.

The semantics for our language is the usual semantics—configurations consist of stores (a scalar store and a pointer store), a heap, and a map that tracks the set of freed locations. The semantics is undefined if the program dereferences `nil` or accesses unallocated locations.

Figure 3 shows the semantic of our programming language. $\mathcal{S}_S$ is the scalar store, mapping scalar variables to values of type S. $\mathcal{S}_P$ is the pointer store, mapping pointer variables to memory locations. $\mathcal{H}$ is the heap mapping a location and label to another Location or a value of type S; a location has a key and a next pointer and they are mapped to S and Loc values respectively. $\mathcal{F}$ represents the set of locations that are freed.

The Lookup rules formalize look-up of variables and pointer dereferences in their corresponding stores and heap respectively. Note that `nil` pointer dereferences are undefined, as are dereferences of freed memory locations. Expressions evaluate to either locations, for pointer expressions, or scalar values for scalar expressions. Evaluating scalar operations just evaluates the left and right operands and applies the semantic of the operation to the results. Boolean expressions evaluate to a boolean value in the standard way and we skip the rules for them, for brevity. Assignment rules formalize assignments and are fairly standard; programs cannot dereference either `nil` or freed locations. A call to `malloc` produces a fresh location distinct from any previous call to malloc. Without loss of generality, for a freshly allocated node, the next field and the key field are by default initialized to `nil` and 0 respectively; this is just to simplify the presentation and not in any way a limitation of our technique. A call to `free` proceeds only if the location is not already freed and it updates the $\mathcal{F}$ map. The semantics for `skip`, `if then else`, `while` and sequential composition is standard and presented in Figure 3.

B. The Assertion Logic

The assertion logic, depicted in Figure 4, is parameterized by a linear measure *linm*, given by a tuple $(D, d_0, \oplus, singleton, contract)$, and our formalism allows tracking assertions written in this logic incrementally over a computation that reads and destructively updates the concrete heap. The domain for the scalars can include standard domains like integers, strings, etc., as well as the domain D for the linear measures. The functions *op* are arbitrary functions on the scalar domain, and includes operations such as $\oplus$ on D.

We require the user to provide the semantics of the linear measure by providing side-effect-free programs that compute

$$\mathcal{S}_S : SVar \longrightarrow S \qquad \mathcal{S}_P : PVar \longrightarrow Loc \qquad \mathcal{H} : Loc \times \{key, next\} \longrightarrow Loc \cup S$$

$$\mathcal{F} : Loc \longrightarrow bool \qquad c_S \in S$$

$$\text{PLookup}\ \dfrac{\mathcal{S}_P(\mathtt{x_P}) = loc}{\langle \mathtt{x_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc} \qquad \text{SLookup}\ \dfrac{\mathcal{S}_S(\mathtt{x_S}) = c_S}{\langle \mathtt{x_S}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow c_S}$$

$$\text{NextLookup}\ \dfrac{\langle \mathtt{x_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc' \quad loc' \neq \mathtt{nil} \quad \neg\mathcal{F}(loc') \quad \mathcal{H}(loc', next) = loc}{\langle \mathtt{x_P} \rightarrow \mathtt{next}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc}$$

$$\text{KeyLookup}\ \dfrac{\langle \mathtt{x_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc \quad loc \neq \mathtt{nil} \quad \neg\mathcal{F}(loc) \quad \mathcal{H}(loc, key) = c_S}{\langle \mathtt{x_P} \rightarrow \mathtt{key}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow c_S}$$

$$\text{AsgnS}\ \dfrac{\langle \mathtt{e_S}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow c_S}{\langle \mathtt{x_S} := \mathtt{e_S};, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}_S[x_S \mapsto c_S], \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle}$$

$$\text{ExprS}\ \dfrac{\langle \mathtt{e_S}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}\rangle \rightsquigarrow t \quad \langle \mathtt{e'_S}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}\rangle \rightsquigarrow t'}{\langle \mathtt{e_S}\ \mathtt{op_S}\ \mathtt{e'_S}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}\rangle \rightsquigarrow t\ [\![op_S]\!]\ t'} \qquad \text{ExprB}\ \dfrac{[\![\mathtt{e_b}]\!]_B(\mathcal{S}_S, \mathcal{S}_P, \mathcal{H}) \rightsquigarrow b}{\langle \mathtt{e_b}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}\rangle \rightsquigarrow b}$$

$$\text{AsgnP}\ \dfrac{\langle \mathtt{e_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc}{\langle \mathtt{x_P} := \mathtt{e_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}_S, \mathcal{S}_P[x_P \mapsto loc], \mathcal{H}, \mathcal{F}\rangle}$$

$$\text{AsgnNext}\ \dfrac{\langle \mathtt{x_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc \quad loc \neq \mathtt{nil} \quad \neg\mathcal{F}(loc) \quad \langle \mathtt{y_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}\rangle \rightsquigarrow loc'}{\langle \mathtt{x_P} \rightarrow \mathtt{next} := \mathtt{y_P};, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}[(loc, next) \mapsto loc'], \mathcal{F}\rangle}$$

$$\text{AsgnKey}\ \dfrac{\langle \mathtt{x_P}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc \quad loc \neq \mathtt{nil} \quad \neg\mathcal{F}(loc) \quad \langle \mathtt{e_S}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow c_S}{\langle \mathtt{x_P} \rightarrow \mathtt{key} := \mathtt{e_S};, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}[(loc, key) \mapsto c_S]\rangle}$$

$$\text{Malloc}\ \dfrac{fresh_P(Dom(\mathcal{F})) = loc}{\langle \mathtt{x_P} := \mathtt{malloc_P}();, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}_S, \mathcal{S}_P[x_P \mapsto loc], \mathcal{H}[(loc, next) \mapsto \mathtt{nil}, (loc, key) \mapsto 0], \mathcal{F}[loc \mapsto false]\rangle}$$

$$\text{Free}\ \dfrac{\langle \mathtt{x_p}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow loc \quad \neg\mathcal{F}(loc)}{\langle \mathtt{free_P}(\mathtt{x_p});, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}[loc \mapsto true]\rangle}$$

$$\text{Skip}\ \dfrac{}{\langle \mathtt{skip};, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle}$$

$$\text{ITEt}\ \dfrac{\langle \mathtt{e_b}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \mathtt{true}}{\langle \mathtt{if\ e_b\ then\ P1\ else\ P2\ fi}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \mathtt{P1}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle}$$

$$\text{ITEf}\ \dfrac{\langle \mathtt{e_b}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \mathtt{false}}{\langle \mathtt{if\ e_b\ then\ P1\ else\ P2\ fi}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \mathtt{P2}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle}$$

$$\text{Seq}\ \dfrac{\langle P1, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}\rangle \rightsquigarrow \langle \epsilon, \mathcal{S}'_S, \mathcal{S}'_P, \mathcal{H}'\rangle}{\langle P1; P2, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}\rangle \rightsquigarrow \langle P2, \mathcal{S}'_S, \mathcal{S}'_P, \mathcal{H}'\rangle}$$

$$\text{While}\ \dfrac{}{\langle \mathtt{while\ e_b\ do\ P_w\ done}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle \rightsquigarrow \langle \mathtt{if\ e_b\ then\ P_w;\ while\ e_b\ do\ P_w\ done\ else}\ \epsilon\ \mathtt{fi}, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}\rangle}$$

Fig. 3: Programming Language Semantics

the measure for singleton lists, that compute $\oplus$ of any two measures, and that compute the contraction function.

Apart from checking properties of scalar variables, the assertion logic allows expressing properties of the heap, in particular $\mathtt{lseg}(x, y)$ that asserts that x points to a list segment ending with y and $\mathtt{Disjointlseg}((x, y), (u, v))$ that asserts that the list-segment from x to y and the list-segment from u to v are disjoint (the sets of memory locations that form them are disjoint). The corresponding properties for lists can also be asserted as we treat a list as a list segment that ends with *nil*. The assertion logic allows terms of the form $linm(l_1, l_2)$ that denote the linear measure of the list-segment from l_1 to l_2 (provided that it is a list-segment) and allows combining these measures using functions and predicates on D.

The semantics of assertions on program configurations is the natural one, and when linear-measures are used on pairs of locations that do not form a list-segment, they evaluate to $\perp$, a value denoting undefinedness; predicates that use $\perp$ under an even number of negations will evaluate to false (and those under an odd number of negations will evaluate to true).

Loc Terms:	lt	$::=$	$x_p \mid$ `nil`
Scalar Terms:	st	$::=$	$c \mid x_s \mid op(\overline{st}) \mid linm(lt, lt)$
Assertions:	φ	$::=$	`true` $\mid pred(\overline{st}) \mid x_p \longmapsto lt \mid$ `lseg`(lt, lt)
			$\mid$ `Disjointlseg`$((lt, lt), (lt, lt))$
			$\mid \varphi \wedge \varphi \mid \neg\varphi$

Fig. 4: Syntax of the assertion logic parameterized by a linear measure *linm*

When evaluating assertions in our incrementalized framework we will use region summaries; the atomic assertions `lseg`(l_1, l_2), the disjointness assertion `Disjointlseg`$((l_1, l_2), (l_2, l_4))$, and the terms that compute linear measures $linm(l_1, l_2)$ will all be evaluated *using the region summaries*. We incrementally update regions as the heap is manipulated by the program. The rest of the assertion will be evaluated directly on the concrete program configuration as they can be quickly evaluated.

V. Incrementally Maintaining Region Summaries

In this section we define region summaries, explain how we leverage them to perform fast assertion checking, and define the operations we perform in order to maintain precise region summaries as the program executes.

A. Region Summaries

The regions of the heap we track are list-segments that lie between a particular subset of *pertinent* locations of the heap. These locations are defined as:

Definition 2 (Pertinent Locations)**.** *A memory location loc is pertinent if it is reachable from a pointer variable and at least one of the two conditions below holds:*
(i) loc is pointed to directly by a pointer variable
(ii) loc is the first location where two or more lists merge.

We fix a heap $\mathcal{H}$, where $\mathcal{H} : Loc \times \{key, next\} \rightarrow Loc \cup S$ giving the partial map for the next-pointer field and the key-field for locations (where S is the domain for scalar variables). In a configuration, let $\mathcal{N}_R$ denote the set of pertinent locations, including `nil`.

Definition 3 (Region Summary)**.** *A region summary in a configuration of the program is a tuple* $(\mathcal{N}_R, \mathcal{R}, \mathcal{LM})$ *where* $\mathcal{N}_R$ *is the set of pertinent locations in the current configuration,* $\mathcal{R} \subseteq \mathcal{N}_R \times \mathcal{N}_R$ *and contains a pair of locations* (l_1, l_2) *iff* l_2 *is the first pertinent node reachable from* l_1 *following the next-pointer, and* $\mathcal{LM} : \mathcal{R} \rightarrow D$ *is the linear measure of the list segments in* $\mathcal{R}$*.*

Note that a region summary maintains linear measures only over the atomic regions of the heap that are flanked by pertinent locations.

An example of a region overlaid on the heap is shown in Figure 1, with the length of list-segments serving as the linear measure. The pertinent nodes (depicted as solid ovals in Figure 1), are the nodes `0x0D00`, `0xB088`, `0x0D48`, and `nil`.

B. Maintaining Region Summaries

Figure 5 shows the rules for maintaining accurate region summaries as the program executes. The concrete store and heap (including freed locations) is maintained in the structure $\mathcal{E}$ and $[\![\texttt{st}]\!](\mathcal{E}) \rightsquigarrow \mathcal{E}'$ denotes the concrete semantics for the statement *st* transforming $\mathcal{E}$ to $\mathcal{E}'$. The region summary is maintained using $\mathcal{N}_R$, $\mathcal{R}$ and $\mathcal{LM}$, and $\mathcal{SF}$ is the subset of pertinent nodes $\mathcal{N}_R$ that are in $\mathcal{F}$.

Intuitively, when the program makes a change involving the store on a pointer variable or a destructive update of the heap, the only regions that change are those that are adjacent to the location where the change is being made. For example, when we execute the statement $x := y$ (rule *AsgnP1*), if x is pointing to a location *loc'*, then *loc'* may become not pertinent in the new heap; this is the case if there is no other pointer variable pointing to it and it is not a merge node. If the location is not pertinent, the function *smoothen*, shown in Figure 6, takes the region leading up to *loc* (there can be at most one since it's not a pertinent node) and concatenates it with the succeeding region (which can also be at most one), and then computes the aggregate linear measure of the two regions for the new larger region using the $\oplus$ operator of the linear measure.

When executing a statement of the form $x := y \rightarrow next$ (rule *AsgnP2*), location *y.next* may not be a pertinent location in the current heap, but is pertinent in the new heap, and we must add this as a new pertinent location (using the function *insertLoc*, shown in Figure 6). This splits the region succeeding y into two regions, and the linear measures on these regions must be computed (using singleton and contract functions) using the linear measure of the older region.

One important aspect is that we use the concrete heap to update the regions. For example, if we execute $x := x \rightarrow next$, we can determine whether the region we were tracking from x has become empty by checking if the new value of x was already pertinent; if so, then the region will collapse.

The following lemma captures formally the local aspect of our updates, in that it shows that a region does not need updating unless it is the immediate successor region to a region that was changed in the concrete heap.

Lemma 1. *A region is not updated unless it is a changed region, i.e., one of the two pertinent nodes is changed, or it is the immediate successor region to a changed region.*

Proof. Consider a pertinent node *loc* in the current configuration, and assume that there is a change to the configuration that does not affect any of nodes in the regions preceding *loc* nor one of the pertinent locations preceding *loc*. Now we argue that *loc* will continue to be pertinent. First, if a pointer variable points to *loc*, then it would continue to do so, and hence *loc* would be pertinent. Otherwise, *loc* must be a merged node that is reachable from two program pointer variables. In the new heap, *loc* will continue to be a merged node, and the only reason it may not be pertinent is that it is no longer reachable from two pointer variables. We will show that all pertinent nodes preceding *loc* will contribute at least one program variable that reaches *loc* through it. Consider

$$\mathcal{E} : \mathcal{S}_S \times \mathcal{S}_P \times \mathcal{H} \times \mathcal{F} \qquad \mathcal{N}_R : Loc_R \qquad \mathcal{R} \subseteq \mathcal{N}_R \times \mathcal{N}_R \qquad \mathcal{LM} : \mathcal{R} \to D \qquad \mathcal{SF} \subseteq \mathcal{N}_R$$

$$\text{AsgnP1}\ \frac{\begin{array}{c} [\![\mathtt{x_P} := \mathtt{e_P}]\!](\mathcal{E}) \rightsquigarrow \mathcal{E}' \quad [\![\mathtt{e_P}]\!](\mathcal{E}) = loc \quad loc \in \mathcal{N}_R \\ [\![\mathtt{x_P}]\!](\mathcal{E}) = loc' \quad smoothen(loc', \mathcal{E}', \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}) = \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}' \end{array}}{\{\mathcal{E}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}\}\ \ \mathtt{x_P} := \mathtt{e_P}\ \ \{\mathcal{E}', \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}'\}}$$

$$\text{AsgnP2}\ \frac{\begin{array}{c} [\![\mathtt{x_P} := \mathtt{y_P} \to \mathtt{next}]\!](\mathcal{E}) \rightsquigarrow \mathcal{E}' \quad [\![\mathtt{y_P} \to \mathtt{next}]\!](\mathcal{E}) = loc \quad loc \notin \mathcal{N}_R \quad [\![\mathtt{y_P}]\!](\mathcal{E}) = loc' \\ [\![\mathtt{x_P}]\!](\mathcal{E}) = loc'' \quad insertLoc(loc', loc, \mathcal{E}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}) = \mathcal{N}_R'', \mathcal{R}'', \mathcal{LM}'' \\ smoothen(loc'', \mathcal{E}', \mathcal{N}_R'', \mathcal{R}'', \mathcal{LM}'', \mathcal{SF}) = \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}' \end{array}}{\{\mathcal{E}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}\}\ \ \mathtt{x_P} := \mathtt{y_P} \to \mathtt{next}\ \ \{\mathcal{E}', \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}'\}}$$

$$\text{AsgnNext}\ \frac{\begin{array}{c} [\![\mathtt{x_P} \to \mathtt{next} := \mathtt{y_P}]\!](\mathcal{E}) = \mathcal{E}' \quad [\![\mathtt{x_P}]\!](\mathcal{E}) = loc \quad [\![\mathtt{y_P}]\!](\mathcal{E}) = loc' \\ update(loc, loc', \mathcal{N}_R, \mathcal{R}, \mathcal{LM}) = \mathcal{N}_R'', \mathcal{R}'', \mathcal{LM}'' \\ smoothen(\mathcal{R}(loc), \mathcal{E}', \mathcal{N}_R'', \mathcal{R}'', \mathcal{LM}'') = \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}' \end{array}}{\{\mathcal{E}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}\}\ \ \mathtt{x_P} \to \mathtt{next} := \mathtt{y_P}\ \ \{\mathcal{E}', \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}'\}}$$

$$\text{AsgnKey}\ \frac{\begin{array}{c} [\![\mathtt{x_P} \to \mathtt{key} := \mathtt{e_S}]\!](\mathcal{E}) = \mathcal{E}' \quad [\![\mathtt{x_P}]\!](\mathcal{E}) = loc \\ [\![\mathtt{e_S}]\!](\mathcal{E}) = k \quad Tupdate(loc, k, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}) = \mathcal{LM}' \end{array}}{\{\mathcal{E}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}\}\ \ \mathtt{x_P} \to \mathtt{key} := \mathtt{e_S}\ \ \{\mathcal{E}', \mathcal{N}_R, \mathcal{R}, \mathcal{LM}', \mathcal{SF}\}}$$

$$\text{Free}\ \frac{\begin{array}{c} [\![\mathtt{free_P}(\mathtt{x_P})]\!](\mathcal{E}) = \mathcal{E}' \quad [\![\mathtt{x_P}]\!](\mathcal{E}) = loc \\ performFree(loc, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}) = \mathcal{N}_R'', \mathcal{R}'', \mathcal{LM}'', \mathcal{SF}'' \\ smoothen(loc, \mathcal{E}', \mathcal{N}_R'', \mathcal{R}'', \mathcal{LM}'', \mathcal{SF}'') = \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}' \end{array}}{\{\mathcal{E}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}\}\ \ \mathtt{free_P}(\mathtt{x_p})\ \ \{\mathcal{E}', \mathcal{N}_R', \mathcal{R}', \mathcal{LM}', \mathcal{SF}\}}$$

$$\text{Mal}\ \frac{[\![\mathtt{x_P} := \mathtt{malloc_P}()]\!](\mathcal{E}) = \mathcal{E}' \quad [\![\mathtt{x_P}]\!](\mathcal{E}') = loc}{\{\mathcal{E}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}\}\ \ \mathtt{x_p} := \mathtt{malloc_P}()\ \ \{\mathcal{E}', \mathcal{N}_R \cup \{loc\}, \mathcal{R} \cup \{loc, \mathtt{nil}\}, \mathcal{LM}[(loc, \mathtt{nil}) \mapsto Singleton(loc)], \mathcal{SF}\}}$$

Fig. 5: Dynamic Update Rules for Region Summaries. Additional referenced functions are defined in Figure 6.

a pertinent node *loc′* preceding *loc* in the old heap. If *loc′* had a pointer variable pointing to it, then this pointer variable reaches *loc*. If not, then *loc′* must be a merged node and hence had at least two program variables that reached it in the old heap. In the new heap, at least one program variable will survive and reach *loc′*, which in turn will reach *loc*. Hence *loc* will continue to be a pertinent merged node in the new heap. For a region defined from l_1 to l_2, if no region preceding l_1 and l_2 are changed, it follows that both nodes will remain pertinent and the region defined by them is unaffected. □

Consequently, the *smoothen* function, when it repairs the summaries with respect to a location that has seen some change, may need to also repair the *next* pertinent location in the old region summary, *but not any other location*.

The above also means that the structural changes to regions we track can be achieved using a constant overhead to the operation performed— however, updating the linear measures, etc., involve calling functions such as singleton, contract, and $\oplus$ of the linear measure a constant number of times, and this may incur more overhead.

We can now state our main theorem:

Theorem 1 (Soundness and Completeness)**.** *After any execution of a program, an assertion φ evaluates to true in the domain of Region Summaries iff it evaluates to true in the concrete configuration.*

Our framework described above works for tracking any class of list-segments with a set of program variable pointers P pointing into it; the number of regions we track is linear in the size of P. In programs that have a constant number of pointers, the regions and their updates incur a constant overhead. However, we can also handle settings with an unbounded P, such as arrays of pointers or lists of pointers to lists, but in this case the number of regions will grow with P.

VI. Evaluation

In this section we demonstrate the benefits of using region summaries to check assertions on list-manipulating programs. Our evaluation aims to provide an insight into how applicable our technique is and assess how much more efficient it is to check assertions using region summaries in comparison with checking assertions on the concrete heap. We design our evaluation around two types of programs: first, a set of benchmarks, small programs, for stress-testing that perform intensive assertion checking, and second, a larger application that uses assertions to automatically detect malware rootkits at runtime.

A. Assertion Checking in the Small

Our benchmarks consists of (a) library programs that manipulate singly-linked lists, obtained from the C GLib library [20], such as programs for the concatenation of two lists, insertion into and deletion from a list, reversal of a list, etc., and (b)

```
$Tupdate(loc, k, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF})$
  $\mathcal{LM}' := \mathcal{LM}[(loc, loc') \mapsto t]\ where(loc, loc') \in \mathcal{R};$
  $t := Singleton(loc) \oplus contract(\mathcal{LM}(loc, loc'), loc, k);$
  return $\mathcal{LM}'$

$insertLoc(loc, suc, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM})$
  $\mathcal{N}'_R := \mathcal{N}_R \cup \{suc\};$
  $oldSuc := loc'where(loc, loc') \in \mathcal{R};$
  $\mathcal{R}' := \mathcal{R} \setminus \{(loc, loc') | (loc, loc') \in \mathcal{R}\}$
        $\cup \{(loc, suc), (suc, oldSuc)\};$
  $\mathcal{LM}' := \mathcal{LM}_{|\mathcal{R}'} \cup \{(loc, suc) \mapsto Singleton(loc, suc),$
          $(suc, oldSuc) \mapsto$
                $contract(\mathcal{LM}(loc, oldSuc), loc, \mathcal{H}(loc, key))\};$
  return $\mathcal{N}'_R, \mathcal{R}', \mathcal{LM}'$

$performFree(loc, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF})$
  $\mathcal{SF}' := \mathcal{SF} \setminus \{loc\};$
  $\mathcal{R}' := \mathcal{R} \setminus \{(loc, loc') | (loc, loc') \in \mathcal{R}\}$
  $\mathcal{LM}' := \mathcal{LM}_{|\mathcal{R}'};$
  return $\mathcal{N}_R, \mathcal{R}', \mathcal{LM}', \mathcal{SF}'$


$smoothen(loc, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF})$
  $\mathcal{N}'_R, \mathcal{R}', \mathcal{LM}', \mathcal{SF}' :=$
        $\overline{smoothen}(loc, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF});$
  $suc := loc'where(loc, loc') \in \mathcal{R};$
  return $\overline{smoothen}(suc, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}, \mathcal{N}'_R, \mathcal{R}', \mathcal{LM}', \mathcal{SF}');$

$\overline{smoothen}(loc, \mathcal{S}_S, \mathcal{S}_P, \mathcal{H}, \mathcal{F}, \mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF})$
  $pt := \{x_p | \mathcal{S}_P(x_p) = loc\};$
  $predecessors := \{pred | \mathcal{H}(pred, next) = loc\};$
  $if\ loc = \texttt{nil} \lor pt \neq \emptyset \lor |predecessors| > 1\ then$
     $//\ the\ location\ is\ pertinent$
     return $\mathcal{N}_R, \mathcal{R}, \mathcal{LM}, \mathcal{SF}$
    $else$
     $\mathcal{N}'_R := \mathcal{N}_R \setminus \{loc\};$    $\mathcal{SF}' = \mathcal{SF} \setminus \{loc\};$
     $\mathcal{R}' := \mathcal{R} \cup \{(loc', loc'') | (loc', loc) \in \mathcal{R} \land (loc, loc'') \in \mathcal{R}\}$
          $\setminus \{(loc', loc) | (loc', loc) \in \mathcal{R}\} \setminus \{(loc, loc'') | (loc, loc'') \in \mathcal{R}\};$
     $\mathcal{LM}' := \mathcal{LM}_{|\mathcal{R}'} \cup \{(loc', loc'') \mapsto \mathcal{LM}(loc', loc)$
         $\oplus \mathcal{LM}(loc, loc'') | (loc', loc) \in \mathcal{R} \land (loc, loc'') \in \mathcal{R}\};$
     return $\mathcal{N}'_R, \mathcal{R}', \mathcal{LM}', \mathcal{SF}'$

$update(loc, loc', \mathcal{N}_R, \mathcal{R}, \mathcal{LM})$
  $\mathcal{R}' := \mathcal{R} \setminus \{(loc, suc) | (loc, suc) \in \mathcal{R}\} \cup \{(loc, loc')\};$
  $\mathcal{LM}' := \mathcal{LM}_{|\mathcal{R}'} \cup \{(loc, loc') \mapsto Singleton(loc)\}\ in$
  return $\mathcal{N}_R, \mathcal{R}', \mathcal{LM}'$
```

Fig. 6: Additional Update Functions

small programs that use linked lists as a library, such as stack and queue implementations, LRU-cache implementations, etc. Our LRU implementation is based on the Least Recently Used page replacement algorithm, used for memory page management, described in Bovet et al. [8]. The first column of Table I lists the names of the benchmark programs we use in our evaluation; the names are self-descriptive. The list programs are fairly concise, ranging in size from 15 lines to 50 lines of code. Our programs that use lists as a library are moderately concise, with up to 200 lines of code. The programs vary in complexity, some executing in constant time, e.g., prepend, while others in linear time.

```
Node* insert_before(Node* slist, Node* sibling, int data)
requires   lseg(slist, nil) ∧ lseg(sibling, nil)
                ∧ Disjointlseg((slist, sibling), (sibling, nil))
invariant  lseg(slist, nil) ∧ lseg(slist, iter)
                ∧ lseg(iter, sibling) ∧ lseg(sibling, nil)
                ∧ Disjointlseg((slist, iter) , (iter, sibling))
ensures   lseg(return, nil)
```

Fig. 7: The contract and assertions for insert_before

We annotated GLib subjects with assertions for preconditions, postconditions, and loop invariants as we would in a typical verification task. Figure 7 shows an example of annotations for a program that inserts a new node with key `data` after the node `sibling`. `iter` is the pointer used to iterate over the list and it is the last node before sibling when the loop stops. Our assertions check structural correctness using the *lseg* (for list segments) recursive predicate and *Disjointlseg* for disjointness between list segments.

Note that the assertions in the GLib programs occur very often, within the loops/recursive calls. Consequently, the assertions are checked a large number of times, linear in the length of the lists manipulated. Some of the clients, such as the queue implementations do not have loop invariants, having only preconditions or postconditions, and hence the assertions occur less frequently in those subjects.

We performed all our experiments on an Intel Core i7 with 32 GB of RAM. We vary the workload in our programs by increasing the input list sizes (list sizes range from 10 to 16384 nodes). We report the total running time for each experiment, obtained over 100 repeating runs to account for noise in measurement, for the increasing list sizes.

We present the results of our evaluation in Table I. The various columns denote the length of the list input to the program. The first column lists the names of the programs we used, and the second column indicates the number of assertion checks that the program performs for a list of length n. Each 3-column group shows, for lists of varying sizes ranging from 10 to 16384, the total time for running the programs. We report the total time to run the programs with *no assertion checks* (**None**), with assertions in the program by *runtime checking on the concrete state* (**Conc**) and *runtime checking using region summaries* (**Reg**).

The runtime checking of the assertions on the concrete heap (Conc) grows with the size of the list, and takes typically linear to quadratic time to check the property. This is reasonable and acceptable for smaller lists (say a few hundred), but gets prohibitively expensive for larger lists, timing out on larger lists. We emphasize that even with a *manual encoding by the programmer*, the check will typically take this time— checking properties of lists should, after all, take longer when the lists are longer.

However, the overhead incurred by checking assertions is much smaller when using region summaries. The total time taken per runtime assertion aided by the region summaries is almost *constant*, varying very little with the length of the input! In lists ranging to 16K, this shows 20x to 3000x speedup

⇒ List Length		10			2048			4096			16384		
⇓ Program	**#Asrt**	**None**	**Conc**	**Reg**	**None**	**Conc**	**Reg**	**None**	**Conc**	**Reg**	**None**	**Conc**	**Reg**
Library													
append	$O(n)$	0	6	7	4	2109s	113	7	t.o.	2215	25	t.o.	8852
concat	$O(n)$	0	6	6	3	2120s	1108	7	t.o.	2230	26	t.o.	8862
copy	$O(n)$	0	6	3	31	2427s	626	62	t.o.	1255	250	t.o.	4980
find	$O(n)$	0	1	1	0	2s	1	0	10s	1	0	164s	1
free	$O(n)$	0	1	1	33	859	209	68	3s	421	274	68s	1691
insert	$O(n)$	0	4	6	4	312s	1068	7	t.o.	2133	28	t.o.	8716
last	$O(n)$	0	6	6	3	2108s	1098	7	t.o.	2208	26	t.o.	8778
reverse	$O(n)$	0	2	3	3	313s	523	7	t.o.	1056	29	t.o.	4256
remove-link	$O(n)$	0	4	4	3	915s	978	7	t.o.	1959	26	t.o.	7833
remove-all	$O(n)$	0	4	5	4	915s	1011	8	t.o.	1966	28	t.o.	7856
position	$O(n)$	0	5	5	3	2114s	840	7	t.o.	1699	25	t.o.	6893
nth-data	$O(n)$	0	3	2	3	2100s	559	6	t.o.	1165	25	t.o.	4566
nth	$O(n)$	0	3	2	3	2154s	562	6	t.o.	1122	25	t.o.	4627
length	$O(n)$	0	4	3	0	2115s	553	0	t.o.	1108	0	t.o.	4507
insert-at-pos	$O(n)$	0	0	4	2	1072s	701	4	t.o.	1394	13	t.o.	5593
index	$O(n)$	0	4	1	0	3s	1	0	13s	2	0	222	1
prepend	$O(1)$	0	1	0	0	4	0	0	9	0	0	31	0
create	$O(n)$	0	0	1	25	860	259	54	3s	520	219	68s	2085
swap	$O(1)$	0	0	1	0	4	0	0	7	0	0	31	1
Client													
split	$O(n)$	0	2	2	4	233s	276	7	t.o.	554	26	t.o.	2312
merge	$O(n)$	0	7	6	3	2716s	1062	6	t.o.	2102	25	t.o.	8278
reverse-sublist	$O(n)$	0	1	1	0	495s	217	0	t.o.	443	0	t.o.	1789
lasso-check	$O(1)$	0	0	0	3	8	11	7	7	22	29	60	84
insert-sorted	$O(n)$	0	0	1	0	893	0	0	3s	0	0	55s	1
queue	$O(1)$	0	0	0	2	138s	52	3	t.o.	110	11	t.o.	629
stack-queue	$O(1)$	0	0	0	1	13s	17	2	533	42	5	843s	356
LRU	$O(n)$	0	1	2	2	54s	18	4	225s	34	19	t.o.	133

TABLE I: Running time using no assertion **None**, concrete assertion checks **Conc**, and region summaries **Reg**. Times are shown in *ms* except where *s* is noted denoting seconds. Timeout (t.o.): 60 min for checking all assertions

in checking the assertions, in comparison with checking them on the concrete heap. Consequently, runtime checking using region summaries scales with acceptable overheads even for our largest input sizes.

Intuitively, the number of regions stays small (constant), and hence checking an assertion on the regions can be done very efficiently. However, the region summaries do require updating as the program executes, but this is a constant overhead for each heap manipulation. When the number of assertions is high and the length of the lists is large, the maintenance cost of the heaplet is not significant, and we incur essentially a constant cost for checking an assertion, independent of the length of the list.

This set of experiments shows that the use of region summaries can make runtime checking of data-structure properties much faster, when the assertions are checked often and the sizes of the data structure grow. We believe that using these in settings where assertions abound, such as in *class invariants*, where invariants are checked every time the data structure is accessed, can benefit greatly from our approach. However, when assertions are sparse, the cost of maintaining the regions may get expensive, and it may be more prudent to check the assertion on the concrete heap. An automatic hybrid approach that exercises these choices to instrument large programs is an interesting future direction.

B. An Application to Detecting Malicious Rootkits

We now evaluate the assertion checking based on region-summary for detecting malicious rootkits at runtime.

	fork			***exit***		
	None	**Conc**	**Reg**	**None**	**Conc**	**Reg**
1024	0.09	7.15	0.12	2.68	10.85	2.75
4096	0.27	134.65	0.38	61.86	242.12	64.75
16385	0.99	2191.98	1.44	1075.54	4451.27	1096.54

TABLE II: Average running times (in ms) for cross-checking linear measures of *allproc* and *pighashtbl* for various list sizes (column 1), and three types of runs: (1) no assertions, (2) concrete checks, and (3) region-summary checks.

Rootkits are programs used to control some aspects of an operating system's behavior, and usually occur in a malicious context aiming to hide a process' existence. Typically they are used to maintain *root* access after successful exploitation of an OS kernel or other programs that provide root privileges [24].

We consider an example of a rootkit that can hide the presence of a particular process from the end user and demonstrate how our technique can be used to detect a class of rootkits with very little overhead. In particular, this type of rootkit relies on a direct manipulation of kernel's underlying data structures: Direct Kernel Object Manipulation (DKOM) [22], and hence can be detected by checking these data-structures at runtime. We propose a technique that relies on protecting kernel objects by slightly modifying them so that all accesses to these data structures are performed through APIs (similar to [35]). Given this, we can instrument the APIs with runtime monitoring to detect kernel manipulation that violates kernel security invariants.

We illustrate the FreeBSD handling of process data structures [1]. FreeBSD [19] stores all running processes in two data structures. The first, *allproc*, stores all processes in a single linked-list used by functions requiring fast traversal of all the processes (such as *ps*). The second, *pighashtbl* is a contiguous array of list entries; it enables access based on the process id (PID) without linear search through the whole list.

We consider a rootkit that could hide its presence by removing the malicious process from the list of all processes while still being accessible through the PID. We implement a runtime check for this property by computing the sum of hashes of the PIDs stored in these lists, which is a linear measure. Given this linear measure, we can check at runtime whether the sum of the hashes of the processes in the two data-structures *allproc* and *pighashtbl* are the same, at appropriate times, e.g., right after a *fork* or *exit* call is executed. Table II shows the results for checking these assertions on the concrete and by using region summaries. This shows that while checks on the concrete cause significant overhead, the incrementalized check causes negligible overhead.

VII. Related Work

Runtime assertion checking has been successfully used in software engineering and programming language design (e.g., [13]). Debugging using assertions expressed as Boolean formulas is a routine software development practice [21]. However, there are relatively few approaches that can be used for checking properties of programs manipulating structurally complex data. A common technique of specification for that class of programs are *representation invariants* (i.e., RepOk [25]). Implementing representation invariants can be hard to get right; also it imposes a significant burden on the developer [9], [27]. Jump et al. [23] introduce dynamic shape analysis and check structural properties of the heap. Crane and Dingel [14] present a declarative language for specifying object models using the Alloy language, and perform runtime checks to ensure that certain user specified locations conform to an object model. However, runtime checking of these properties incurs large overheads.

Some recent approaches (e.g., [3], [30]) propose using runtime checking for assertions written in separation logic. Separation logic has been successfully used as a specification language in deductive verification of programs manipulating complex data structures, making it an attractive choice for runtime assertion checking. However, as noted by Nguyen et al. [30], runtime checks of separation logic assertions can be challenging due to implicit footprint and existential quantification. The main focus of the work described by Nguyen et al. [30] is alleviating potentially exponential blow-up of sets of locations that need to be considered when splitting the heap in two parts when checking separation. They use a marking technique to limit the set of footprints that needs to be explored when evaluating the formula. Their approach works well when checking only preconditions and postconditions of data structures at the boundary between statically verified and unverified code; however, performing multiple assertion checks often incurs prohibitively large overheads. In contrast, we focus on checking a logic for lists with linear measures, which are suitable for runtime assertion checking, and develop a technique that allows checking properties in near constant time using region summaries. Our technique works well both in cases where assertions are at the boundary and also when they are intensively checked throughout the program.

Agten et al. [3] devised another technique for run-time checks of separation logic assertions. This approach combines deductive verification with run-time checks of the unverified parts of the code to provide stronger run-time guarantees for the verified parts. A key difference to our approach is that the assertions are meant to be checked sparsely (only when crossing the verified-unverified boundary), while our approach excels even when assertions are checked frequently.

A technique proposed by Shankar and Bodik [34] reduces the run-time overhead through incremental assertion checking. The technique devises automatic memoization; it reuses the results of checks on unchanged parts of data structures, while recomputing on the concrete heap on parts that reach the change. In contrast, we show how to only locally update the regions without recomputing concretely on unchanged regions.

Vechev et al. [38] present Phalanx, a tool that uses parallelism to speed up assertion checking. PHALANX evaluates the assertions in a different thread, on a snapshot of the entire state of the program at the assertion. Similarly, Aftandilian et al. [2] introduce asynchronous assertions, which can be checked during debugging. Our work shares the goal of speeding up assertion checking, but we use summaries to perform the assertion checks in constant time, and the above techniques are orthogonal to our work.

VIII. Conclusions

Our main contribution is a technique for fast runtime checking of assertions over linked lists; we have shown that efficiently maintaining region summaries at runtime is feasible and helps in efficient runtime assertion checking, often close to constant time per assertion. We have demonstrated how to apply region summaries to check properties of interest for security. We envision that the idea of region summaries can help in other contexts where assertions are complex and expensive to check. Assertion checking of heap-properties of programs is under-utilized, and we foresee making steps towards good and intuitive assertion languages and designing procedures to quickly check complex properties at runtime could foster the use of more complex assertions.

Acknowledgments: We thank Owolabi Legunsen, Muhammad Suleman Mahmood, Darko Marinov, August Shi, and the anonymous reviewers for feedback on earlier drafts of this manuscript. This research was partially supported by the NSF Grant Nos. CCF-1138994, CCF-1421503, CCF-1527395, and CNS-1646305.

[1]Other operating systems use similar structures for process manipulation.

References

[1] JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. 1999.

[2] E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav. Asynchronous assertions. In *OOPSLA '11*, pages 275–288, 2011.

[3] P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. In *POPL'15*, pages 581–594, 2015.

[4] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, pages 209–234, 2005.

[5] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, pages 670–684, 2011.

[6] M. Barnett, M. Fahndrich, and F. Logozzo. Embedded contract languages. In *ACM SAC - OOPS*, pages 2103–2110, 2010.

[7] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *CASSIS'04*, pages 49–69, 2005.

[8] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[9] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *ISSTA'02*, pages 123–133, 2002.

[10] M. Carrillo-Castellon, J. Garcia-Molina, E. Pimentel, and I. Repiso. Design by contract in smalltalk. 1996.

[11] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. ICSE '15, pages 755–766, 2015.

[12] P. Chalin. Logical foundations of program assertions: What do practitioners want? SEFM '05, pages 383–393, 2005.

[13] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, pages 25–37, 2006.

[14] M. L. Crane and J. Dingel. Runtime conformance checking of objects using Alloy. In *Electronic Notes in Theoretical Computer Science*, pages 2–21, 2003.

[15] B. Elkarablieh, Y. Zayour, and S. Khurshid. Efficiently generating structurally complex inputs with thousands of objects. In *ECOOP'07*, pages 248–272, 2007.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. SOSP '01, pages 57–72, 2001.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, pages 99–123, 2001.

[18] H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *FM'14*, pages 230–246. 2014.

[19] https://github.com/freebsd/freebsd/blob/master/sys/sys/proc.h, 2015.

[20] https://developer.gnome.org/glib/, 2015.

[21] C. A. R. Hoare. Assertions: A personal perspective. *IEEE Ann. Hist. Comput.*, pages 14–25, 2003.

[22] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[23] M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *ISMM '09*, pages 119–128, 2009.

[24] J. Kong. *Designing BSD Rootkits*. No Starch Press, 2007.

[25] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. 1st edition, 2000.

[26] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE'07*, pages 416–426, 2007.

[27] M. Z. Malik, A. Pervaiz, E. Uzuncaova, and S. Khurshid. Deryaft: A tool for generating representation invariants of structurally complex data. In *ICSE '08*, pages 859–862, 2008.

[28] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.

[29] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[30] H. H. Nguyen, V. Kuncak, and W.-N. Chin. Runtime checking for separation logic. In *VMCAI'08*, pages 203–217, 2008.

[31] F. Pastore, L. Mariani, A. E. Hyvärinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad. Verification-aided regression testing. In *ISSTA'14*, pages 37–48. ACM, 2014.

[32] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, pages 19–31, 1995.

[33] D. Schuler and A. Zeller. Checked coverage: an indicator for oracle quality. *Software Testing, Verification and Reliability*, pages 531–551, 2013.

[34] A. Shankar and R. Bodík. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI '07*, pages 310–319, 2007.

[35] A. Srivastava and J. Giffin. Efficient protection of kernel data structures via object partitioning. ACSAC '12, pages 429–438, 2012.

[36] N. Tillmann and J. de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs 2008*, pages 134–153, 2008.

[37] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE'13*, pages 253–262, 2005.

[38] M. Vechev, E. Yahav, and G. Yorsh. Phalanx: Parallel checking of expressive heap assertions. In *ISMM '10*, pages 41–50, 2010.

[39] J. Voas and L. Kassab. Using assertions to make untestable software more testable. *Software Quality Professional*, pages 1–16, 1999.

[40] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *FSE'15*, pages 214–224, 2015.