Points-to analysis (specifically, the may-points-to flavor of points-to analysis) computes, for each pointer \( q \) and program point \( i \), a points-to set, \( pt-set_i(q) \) containing abstract locations that \textit{may} be validly pointed to by \( q \) at program point \( i \) (just before the statement at \( i \) is executed). Figure 4.1 shows the points-to sets computed for an illustrative example:

At line 1, \( p \) is assigned the address of \( x \), so \( x \) is added to \( p \)'s points-to set for program point 2. The assignment at line 2 copies the points-to set from the right-hand-side (\( p \)) to the left-hand-side (\( q \)). As a result of the conditional at line 3, the points-to set for \( pp \) (for program point 4) states that \( pp \) may point to either \( p \) or \( q \) at runtime. For the assignment via a pointer dereference at line 4, we must use the points-to set for \( pp \) to determine which locations \textit{may} be assigned the new value: the points-to sets computed for line 5 state that both \( p \) and \( q \) may point to either \( x \) or \( y \).

We adopt the standard assumption that the points-to set for a pointer \( q \) includes only locations to which \( q \) may \textit{validly} point. That is, the points-to set for \( q \) does not include

\begin{verbatim}
int x, y, *p, *q, **pp;
1. p = &x;
2. q = p+1;
3. pp = (...)?&p:&q;
4. *pp = &y;
\end{verbatim}

\begin{tabular}{|c|c|c|}
\hline
\text{Line} & \text{Points-to Set} & \text{Points-to Set} \\
\hline
1. & \text{pt-set}_1(p) = \{\} & \text{pt-set}_1(q) = \{\} & \text{pt-set}_1(pp) = \{\} \\
2. & \text{pt-set}_2(p) = \{x\} & \text{pt-set}_2(q) = \{\} & \text{pt-set}_2(pp) = \{\} \\
3. & \text{pt-set}_3(p) = \{x\} & \text{pt-set}_3(q) = \{x\} & \text{pt-set}_3(pp) = \{\} \\
4. & \text{pt-set}_4(p) = \{x\} & \text{pt-set}_4(q) = \{x\} & \text{pt-set}_4(pp) = \{p, q\} \\
5. & \text{pt-set}_5(p) = \{x, y\} & \text{pt-set}_5(q) = \{x, y\} & \text{pt-set}_5(pp) = \{p, q\} \\
\hline
\end{tabular}

Figure 4.1 Points-to Analysis Example
locations to which \( q \) may point as a result of an invalid (e.g., out-of-bounds) access. For example, after line 2, \( q \) may in fact point to \( y \) in a way that violates memory safety, but \( pt-set_3(q) \) does not include \( y \).

Precise points-to analysis is \( NP \)-hard [Lan+91, Hor97], so algorithms must trade off precision for lower complexity. Flow-sensitive analyses [Lan+92, Ema+94, Wil+95], which compute a program-point-specific points-to set for each dereference, can give precise results, but do not scale well to large programs. Flow-insensitive analyses [And94, Ste96, Sha+97, Das00], which compute for each pointer \( q \) a single points-to set \( pt-set(q) \) that holds at all program points, are more efficient but less precise than flow-sensitive analysis. Our implementation uses the One-Level-Flow analysis [Das00], which is flow-insensitive and runs in near-linear time (and thus scales well to large programs).

To ensure the correctness of points-to analysis — that is, to ensure that all locations that may validly be pointed-to by \( q \) are included in \( pt-set(q) \) — the whole program must be analyzed. For libraries or other components for which source code may not be available, the behavior of each function must be safely accounted for with a model of the function.

Our points-to analysis does not distinguish between fields of structures or elements of arrays. This means that if \( q \) points to some field of structure \( s \) then \( q \)'s points-to set includes all of \( s \). It also means that if a field \( s.f \) of structure \( s \) may point to some location \( x \), then the analysis will report that \( any \) field of \( s \) may point to \( x \) — this simplifies the analysis but at the cost of some loss in precision. Since our analysis does not distinguish between structures, we can treat dereferences of the form \( p->\text{mem} \) as equivalent to \( *p \). Therefore, in this chapter we assume all dereferences are of the form \( *p \).

### 4.1 Classifying Tracked Locations

For a given \( \text{checked-derefs}(P) \), the corresponding \( \text{tracked-locs}(P) \) must include all locations that may be a valid target of a dereference in \( \text{checked-derefs}(P) \). Since the points-to set for a pointer \( p \) is guaranteed to contain any location that may be a valid target of \( *p \), \( \text{tracked-locs}(P) \) can be computed as the union of the points-to sets of the dereferences in
checked-derefs(P):

\[
\text{tracked-locs}(P) = \bigcup_{\langle i, q \rangle \in \text{checked-derefs}(P)} \text{pt-set}_i(q)
\]

This gives a concise set of locations that may be validly accessed by a checked dereference at runtime.

In the example from Figure 3.2, only buf is classified as tracked, since \( \text{pt-set}(\text{dst}) = \{\text{buf}\} \). The other locations, src, fp, and dst, are not in the points-to set of any dereferenced pointer, so they can be excluded from the tracked-locs set. This gives us the runtime configuration of column (d) that we wanted.

With the naive checked-derefs(P), we computed the corresponding tracked-locs(P) with this technique. Using this classification to instrument the programs, the resulting runtime performance is significantly better than the naive approach, as shown in Figure 4.2: the average slowdown is 208% for checking reads and writes (a 35% improvement over the naive classification), and is 60.3% for checking writes only (a 62% improvement over the naive classification).

### 4.2 Extended Points-To (EPT) Analysis

To reduce the size of checked-derefs(P), we use static analysis to determine whether a given pointer dereference is definitely safe, that is, if it can be guaranteed to never violate memory safety. A dereference that is definitely safe need not be instrumented, and thus can be removed from checked-derefs(P). A dereference that cannot be statically determined to be definitely safe is potentially unsafe, and must be included in checked-derefs(P).

As a first step in classifying definitely safe and potentially unsafe dereferences, we develop an Extended Points-To (EPT) analysis to conservatively identify pointers that may cause a spatial access error. In C, a pointer dereference \(*p\) can cause a spatial access error only if it has been assigned one of the following:

- a numeric value (including NULL);
The extended points-to analysis introduces a special “bottom” location ($\perp$) to represent the target of a non-zero numeric value or a computed value. The idea is that if $\perp$ is in the points-to set of some pointer $q$, then $q$ could potentially point outside its valid target. For example, $\perp$ is included in $pt-set(q)$ as a result of analyzing any of the following: $q = 3$, $q++$, $q = a+b$, $q = a\mid b$. 

Figure 4.2 Points-To Analysis Classification: Runtime Overhead

- a computed value (a value that is the result of applying an arithmetic or bitwise operator to one or more operands);

- the address of an object that is smaller than the size of $*p$’s static type.
Note that we do not add $\bot$ to $q$’s points-to set if $q$ is assigned the value zero (NULL). This is because in most programming settings, dereferencing a null pointer causes a segmentation violation, which terminates the program. In the context of a security tool, a segmentation violation is a safe termination of a program that prevents an attack from successfully causing harm; therefore, we treat null-pointer dereferences as definitely safe. Since the NULL pointer (or value 0) is the default value of global variables and occurs frequently in C programs, this adjustment significantly reduces the number of potentially unsafe pointers identified.

The points-to analysis algorithm accounts for propagating the $\bot$ location from one points-to set to another as a result of assignments. For example, assume that $\bot$ is in $p$’s points-to set, and that $p$ is in $q$’s points-to set. Given the assignments $a = p$ and $b = *q$, points-to analysis will determine that $\bot$ is also in the points-to sets of both $a$ and $b$.

After performing this extended points-to analysis, we classify the dereference $*p$ at program point $i$ as potentially unsafe if any of the following holds:

1. $\bot \in pt\text{-}set_i(p)$, or
2. $x \in pt\text{-}set_i(p)$ such that $\text{sizeof}(x) \leq \text{sizeof}(\star p)$, or
3. $pt\text{-}set_i(p)$ contains a stack variable, or
4. $pt\text{-}set_i(p)$ contains a heap location that may be freed.

A dereference that does not satisfy any of these criteria is definitely safe, and can be excluded from $\text{checked-derefs}(P)$.

The criteria above describe conditions under which $*p$ may cause an invalid access: 1 and 2 because $p$ may point outside its valid target (potentially causing a spatial access error), 3 because $p$ may point to a stack variable that is deallocated when a function returns, and 4 because $p$ may point to a heap object that has been freed. To check criterion 4, we search the program $P$ for calls to $\text{free}$; for each call $\text{free}(q)$, each location in $q$’s points-to set that is of the form $\text{malloc}_i$ is included in the set of heap locations that may be freed.
Note that since an array-index expression $A[i]$ is treated as the equivalent pointer dereference $*\text{tmp}$, with $\text{tmp} = A+i$, the $+$ computation adds $\bot$ to $\text{tmp}$’s points-to set, thus making $*\text{tmp}$ a checked dereference; therefore, all array-index expressions are considered to be potentially unsafe dereferences, even if the index value is a statically known constant (an analysis for identifying array index expressions that are guaranteed to be in-bounds is described in Chapter 6: Pointer-Range Analysis).

### 4.2.1 May-Be-Uninitialized Pointers

Using the results of extended points-to analysis, it is possible for a dereference of an uninitialized pointer to go undetected. This possibility arises because the underlying flow-insensitive analysis assumes that all variables are properly initialized. This means that even if a pointer $p$ may be dereferenced while containing uninitialized data, $*p$ may be classified as definitely safe. This is a problem, as dereferencing an uninitialized pointer may potentially be exploited by an attacker.

To prevent dereferences of uninitialized pointers from happening, we add instrumentation to zero-initialize any location that may be the source of an uninitialized value that is dereferenced. The idea is to effectively translate any potential uninitialized-pointer dereference to a null-pointer dereference, which would cause the program to crash, and thus would not be exploitable.

The set of locations that must be zero-initialized is identified by performing a simple flow-insensitive analysis. First, an assignment graph is built, with a vertex for each location in $\text{locs}(P)$, and an edge from $x$ to $y$ representing a possible flow of value from $x$ to $y$ (resulting from a direct assignment $y = x$ or an indirect assignment like $*q = *p$ where $x \in pt\text{-}set(p)$ and $y \in pt\text{-}set(q)$). Then, for each pointer in an unchecked dereference (that is, for each $p$ such that there is some $\langle i, *p \rangle \notin \text{checked\text{-}derefs}(P)$), any stack or any heap location that is backward-reachable from $p$ in the assignment graph, including $p$ itself, must be zero-initialized. (Note that global or static variables are already required to be zero-initialized by the C language specification.)
4.2.2 Performance

Using the \textit{checked-derefs} set consisting of potentially unsafe dereferences, and the corresponding \textit{tracked-locs} set, the runtime overhead of the instrumented programs improved slightly overall, and significantly for two small benchmarks when checking writes only, as shown in Figure 4.3. The slowdown introduced by zero-initializing may-be-uninitialized locations (not shown in the graphs) is negligible, averaging less than 1%, with a maximum slowdown of 9% (for \textit{mesa}).
Figure 4.4 shows the compilation time slowdown, comparing the time it takes to analyze, instrument, and compile each program with the time it takes to compile the uninstrumented executable. The black portions of the bars give the slowdowns introduced by instrumentation, which are similar to the overheads for instrumenting the programs with the naive MSE classification. The lighter portions show the slowdowns due to Extended Points-To (EPT) Analysis, which is negligible in most cases, and reasonable for the larger programs.\(^1\) This is due to EPT being based on a fast flow-insensitive points-to analysis that scales well to large programs. The analysis of \texttt{mesa} is slow because the program has some large structures (with over a thousand fields), which our implementation currently does not handle efficiently.

\(^1\)The slowdown factors are defined as follows:

Let \(t'_{\text{ana}}\) be the analysis time,
\(t'_{\text{inst}}\) be the MSE instrumentation time,
\(t'_{\text{comp}}\) be the compilation time of the instrumented source files, and
\(t_{\text{comp}}\) be the compilation time of the original (uninstrumented) source files.

The instrumentation slowdown is \(\frac{t'_{\text{inst}}+t'_{\text{comp}}}{t_{\text{comp}}}\), while the analysis slowdown is \(\frac{t'_{\text{ana}}}{t_{\text{comp}}}\).
4.3 Stack and Heap Locations

In the extended points-to analysis, criteria 3 and 4 (on page 40) appear to be overly conservative: \( *p \) is considered potentially unsafe if \( p \)'s points-to set contains any stack variable or any heap location that is ever freed. This is to account for the possibility that the stack or heap location may be deallocated prior to the dereference, thus resulting in a dangling-pointer dereference. By using lifetime or escape analysis [Rug+88, Par+92], we could identify cases where such a dangling-pointer dereference could never occur.

To gauge the potential improvement from such an analysis, Figure 4.5 shows the runtime overhead if we modify the Extended Points-to (EPT) classification scheme to ignore criteria 3 and 4. ‘Stack OK’ assumes that stack locations are never dereferenced via dangling pointers (i.e., criterion 3 is ignored), while ‘Heap+Stack OK’ assumes that no dangling pointer is ever dereferenced (i.e., both criteria 3 and 4 are ignored). These unsafe assumptions give an upper bound on the improvement that can be attained by performing escape analysis.

The difference in performance compared to EPT is usually quite small. It may be surprising that the difference is not greater, since stack and heap locations account for most of the memory usage in these programs. The reason for the small difference in performance is that large objects are likely to be classified as tracked because of criterion 1, and the handling of large objects is a dominant contributor to the runtime overhead (when allocating or deallocating a large object, the tags of the entire object must be set or cleared). In the few cases where criteria 3 and 4 make a notable difference, most of the potential improvements are due to stack variables that are passed by reference to a function.
Figure 4.5 Unsafe Stack/Heap Assumptions: Runtime Overhead