Kaleidoscope: Precise Invariant-Guided Pointer Analysis

Tapti Palit
tpalit@purdue.edu
Purdue University
West Lafayette, Indiana, U.S.A

Pedro Fonseca
pfonseca@purdue.edu
Purdue University
West Lafayette, Indiana, U.S.A

Abstract

Pointer analysis techniques are crucial for many software security mitigation approaches. However, these techniques suffer from imprecision; hence, the reported points-to sets are a superset of the actual points-to sets that can possibly form during program execution. To improve the precision of pointer analysis techniques, we propose Kaleidoscope. By using an invariant-guided optimistic (IGO) pointer analysis approach, Kaleidoscope makes optimistic assumptions during the pointer analysis that it later validates at runtime. If these optimistic assumptions do not hold true at runtime, Kaleidoscope falls back to an imprecise baseline analysis, thus preserving soundness. We show that, compared to the average points-to set size by 13.15x across a set of 9 applications over the current state-of-the-art pointer analysis framework. Furthermore, we demonstrate how Kaleidoscope can implement control flow integrity (CFI) to improve the security of traditional CFI policies.

ACM Reference Format:

1 Introduction

Applications developed in C/C++ make extensive use of code and data pointers. Therefore, accurately resolving the targets of pointers in these applications is important for various software security and software engineering techniques. For example, forward-edge control flow integrity (CFI) [11] requires the points-to sets of each function pointer. Software debloating [12, 27] requires the precise callgraph of the application, and therefore needs to resolve the targets of function pointers, too. Similarly, automatic privilege separation mechanisms [19, 35, 42] that isolate accesses to privileged data, must resolve which data pointers point to privileged data. These techniques, therefore, critically rely on pointer analysis—a class of static code analysis techniques that identifies the application object targets of each pointer.

Unfortunately, pointer analysis techniques struggle to achieve precision, especially when applied to complex systems that consist of hundreds of thousands of lines of code. Static pointer analysis imprecision arises primarily due to the inability to model all runtime information statically. For example, a function accepting pointer arguments that is invoked from multiple callsites with different arguments would result in different points-to relationships depending on the arguments passed from each calling context. An analysis technique that does not differentiate between these different calling contexts, therefore, results in imprecision as the pointer arguments at these different callsites cannot be modeled distinctly.

Imprecision significantly hinders the applicability of pointer analysis. For example, imprecise pointer analysis results in overly permissive control flow integrity (CFI) policies, thus weakening security guarantees. Figure 1 compares the number of CFI targets for each callsite derived by the modern SVF [10] pointer analysis framework against the number of targets observed experimentally through runtime execution for the popular lightweight SSL library, MbedTLS [9], when performing 1000 SSL requests for a 4KB file. While it is impossible to ensure full coverage using execution alone, it is indicative of the imprecision that the static analysis concludes that 92% of all indirect callsites can invoke 184 out of all 185 address-taken functions. This results in a highly permissive CFI policy where every indirect call-site is allowed to invoke all of the address-taken functions in the application, thereby reducing the CFI security effectiveness.

To improve pointer analysis precision, we propose Kaleidoscope, a system that combines the knowledge gained during the static analysis process with dynamic runtime information. Unlike traditional efforts [15, 43, 49, 52, 53] that aim to reduce the imprecision statically, we observe that by making optimistic assumptions about the points-to sets of certain key pointers, during the static analysis, we can significantly reduce the impact of static imprecision on the pointer analysis use cases. These optimistic assumptions are then monitored.
we present the notion of Invariant Guided Memory Views. The first insight stems from the observation that once imprecise invariants hold, our analysis can reduce imprecision, thus providing us with hints that indicate likely imprecise derivations that can cause further explosion in imprecision down the line. Using these hints, we can postulate likely invariants (e.g., a particular pointer can never point to a particular object) that are likely to hold true during the lifetime of the program. By assuming that these likely invariants hold, our analysis can reduce imprecision, thus resulting in stricter security policies (e.g., reduced authorized targets in CFI). Clearly, the choice of likely invariants is key for our system to be effective and practical. In particular, we want the likely invariants to mitigate a significant amount of imprecision and also be few to limit the runtime overhead of monitoring them.

In order to maintain soundness guarantees, if the likely invariants assumed to hold during analysis are violated during program execution, we must provide a way for the system to recover from the violation. Thus, to preserve soundness, Kaleidoscope adapts to the violation of the likely invariants at runtime, without compromising soundness. To this end, we present the notion of Invariant Guided Memory Views. First, when the static pointer analysis algorithm encounters a derivation that Kaleidoscope deems to be a candidate for likely invariant, the analysis explores both paths—one where the likely invariant holds, and one where it does not. Thus, at the end of the analysis, we generate two points-to collections, one where all of the likely invariants hold, and one where none of them hold.

Kaleidoscope uses these two points-to collections to generate different versions of the hardened instructions for the binary, one according to the precise analysis, where all likely invariants hold, and one according to the imprecise analysis, where the likely invariants do not hold. We call these hardened instructions, that depend on the likely invariant, Memory Views. In the case of control flow integrity, these hardened instructions consist of the function pointer validation checks that are inserted before each indirect function call. We further instrument the target application to insert monitors that observe the state of the likely invariant during runtime at critical program points. If the execution of a particular program statement causes the likely invariant to be violated, Kaleidoscope securely switches the memory view to the fallback MV.

While Kaleidoscope can be applied to a variety of use-cases, in order to illustrate the precision improvements provided by Kaleidoscope, we implement a fine-grained forward-edge control flow integrity framework using Kaleidoscope for 9 popular applications. Using the results of our system, we insert CFI validation checks before each indirect call-site that involves a function pointer. Evaluating the hardened binaries with popular benchmarking tools validates our choice of likely invariants as we observe that none of the likely invariants are violated during execution. Across all 9 applications, Kaleidoscope provides an average precision improvement of 13.15× with a performance overhead of 5.45%, thus showing that Kaleidoscope is a practical solution for pointer analysis precision improvement.

2 Background and Motivation
2.1 Static Pointer Analysis
To illustrate how pointer analysis works, we consider the fragment of code snippet shown in Figure 2. The three program statements, P1, P2, and P3 initialize and assign the three pointers p, q, and r. The goal of pointer analysis techniques is to soundly derive all potential points-to relationships between all such pointers and objects in the application. For example, in this case, the analysis would determine that \( \text{PTS}(r) = \emptyset \). Pointer analysis aims to derive these points-to relationships precisely across thousands of such constraints.

Pointer analysis consists of two phases. First, the modeling phase converts program statements to constraints. These constraints are represented as a constraint graph. The nodes of the graph are the program pointers and objects, while the edges represent the constraints that determine how the program statements impact the points-to sets of the pointers. Once the constraints directly corresponding to program

![Figure 1. Indirect callsite targets for the MbedTLS library.](image-url)
statements are captured, they are solved by iteratively applying constraint resolution rules, progressively updating the points-to sets of each application pointer. Because applications commonly use double and triple pointers, constraint resolution often adds new constraint edges to the constraint graph that represent the flow of pointer values through these multi-level pointers. Table 1 summarizes the different constraints and their resolution rules for Andersen’s [15] solver, which is the basis of the state-of-the-art pointer analysis frameworks [10].

**Primitive and Derived Constraints.** Pointer analysis constraints either (a) correspond directly to a program statement or (b) are generated by solving an already known constraint. For presentation purposes, we call the former *primitive* constraints and the latter *derived* constraints. Primitive constraints can be simple and correspond to a direct assignment \( p = q \), or can be conditional, such as \( p = *q \) or \( *q = p \), where the resolution is conditioned on the points-to set of \( q \). As depicted in Table 1, primitive constraints of type *Load* and *Store*, which correspond to indirect load and store operations, result in the addition of new derived *Copy* constraints based on the points-to sets of the pointers involved in the indirect access.

Figure 2 illustrates a simple sample code snippet and its corresponding constraint graph. Constraint C1 adds the object \( q \) to the points-to set of \( p \). Constraint C2 adds \( p \) to the points-to set of \( q \). Finally when C3 is solved, it adds the derived *Copy* constraint edge C4 from \( p \) to \( r \). By solving C4, the pointer analysis algorithm finally concludes that \( r \) can point to \( q \).

Thus, if imprecision causes the spurious addition of objects to the points-to set of a double or triple pointer, it results in the addition of spurious derived constraint edges. Although the constraint graph in this example is relatively simple, the constraint graph of real-world applications is significantly more complex, often contain hundreds of thousands of constraint nodes and edges, and determining these spurious derived constraint edges requires advanced techniques.

### 2.2 Precision Challenges in Static Pointer Analysis

Static pointer analysis is performed at compile time and therefore, lacks runtime information. Certain information that is easily available at runtime, such as values of offsets added to pointers, are challenging to derive statically as they can depend on user input. This causes imprecision in the final points-to results. We briefly discuss four of the common imprecision sources for C programs [46] and how they compound, resulting in potential amplification of the analysis imprecision.

**Context Sensitivity.** A context-sensitive pointer analysis qualifies each program statement in a function with information about the calling context. The calling-context information is critical for precision in cases where the function is invoked from multiple callites, with different arguments, thus establishing different candidate relationships at each callite. However, maintaining the calling context information for arbitrarily deep function calls significantly reduces the scalability of the analysis. Thus, pointer analysis clients [20, 27, 28] opt to use a context-insensitive pointer analysis instead, thus sacrificing precision.

**Flow and Path Sensitivity.** Similar to context-sensitivity, a flow-sensitive pointer analysis qualifies each program statement with its order of execution. The execution order information allows the analysis to determine when a pointer is being overwritten, thus allowing it to invalidate stale pointer relationships at each program statement, thus ensuring precision. Similarly, a path-sensitive analysis qualifies each program statement with its branch information and can distinguish between the pointer relationships established along the true and false branches of if-else statements. Thus, the lack of flow and path sensitivity causes imprecision.

The key property of these various imprecision sources is that they often compound. Figure 3 shows a simplified example from the popular MbedTS SSL library. During pointer analysis, context insensitivity first causes the SSL object to be added to the points-to set of pointer \( s \), which is updated using arbitrary pointer arithmetic in a statement \( *(s+i) = *(ptr+i) \). This causes the SSL object to become field insensitive. The SSL object, in turn, contains the function pointers \( f_send, f_recv, \) and \( f_recv_timeout \). Turning the object field insensitive causes *each of these* function pointers to share the same points-to set, making the analysis (wrongly) conclude that *each of these* function pointers can point to either \( mbedtls_net_send, mbedtls_net_recv, \) and \( mbedtls_net_recv_timeout \). Moreover, the imprecision explosion propagates to all arguments and return values of the corresponding indirect callites.
Table 1. Pointer analysis constraints and resolution rules for a field-sensitive Andersen’s algorithm [15, 43]. Pointers (e.g., \( p \) and \( q \)) may be single-level (i.e., \( \text{int}^* \)) or multi-level (e.g., \( \text{int}^{**}, \text{int}^{***} \)). \( \text{Pts}(p) \) represents the points-to set for the pointer \( p \).

<table>
<thead>
<tr>
<th>Program Statement</th>
<th>Constraint</th>
<th>Constraint Type</th>
<th>Resolution Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = &amp; x )</td>
<td>Addr-Of</td>
<td>Primitive</td>
<td>( x \in \text{Pts}(p) )</td>
</tr>
<tr>
<td>( p = *q )</td>
<td>Load</td>
<td>Primitive</td>
<td>( \forall u \in \text{Pts}(q), \text{add Copy constraint } p = a )</td>
</tr>
<tr>
<td>( *p = q )</td>
<td>Store</td>
<td>Primitive</td>
<td>( \forall u \in \text{Pts}(p), \text{add Copy constraint } a = q )</td>
</tr>
<tr>
<td>( p = q )</td>
<td>Copy</td>
<td>Primitive or Derived</td>
<td>( \text{Pts}(p) = \text{Pts}(p) \cup \text{Pts}(q) )</td>
</tr>
<tr>
<td>( p = &amp; (q.k) )</td>
<td>Field-Of</td>
<td>Primitive</td>
<td>( \text{Generate a new constraint node } f \text{ for field } q.k, \text{Pts}(p) = \text{Pts}(p) \cup f )</td>
</tr>
</tbody>
</table>

Figure 3. MbedTLS code snippet showing how imprecision compounds.

3 Invariant-guided Optimistic Analysis

Traditional points-to analysis techniques rely on static analysis to attempt to identify the points-to set of each pointer. Unfortunately, as discussed in §2.2, there are fundamental challenges that make this static-time process imprecise. This work explores a different angle to this problem. Instead of attempting to statically identify the points-to set of each pointer, Kaleidoscope’s approach makes optimistic assumptions about the program during static analysis that it can verify and, if necessary, correct by falling back on a conservative analysis during runtime. We name this approach invariant-guided optimistic (IGO) points-to analysis.

Goals and Requirements. IGO aims to soundly reduce the points-to set sizes, increasing the effectiveness of common point-to-analysis use cases. In security-related use cases, such as CFI and debloating, effectiveness increases are particularly important since they translate into security increases. However, to ensure effectiveness, the IGO approach has to satisfy two requirements. First, the analysis has to remain, in practice, sound (i.e., during runtime no pointer can point to an object not in the points-to set). Ensuring soundness is important because use cases such as CFI require a sound analysis for correctness. Second, the system must reduce the points-to set sizes while still ensuring sufficient application performance. The effectiveness gains may justify some performance reduction, especially in security-sensitive uses cases, but performance impact should be reasonable.

Insight. The key observation leveraged by IGO is that some program properties that are hard to validate during static time, significantly impact the points-to set sizes. Hence, traditional points-to analysis techniques assume a worst-case scenario to ensure soundness. However, making an optimistic assumption during static time would allow an analysis framework to reduce the points-to set, as long as there is a fallback mechanism for disproven assumptions.

Challenges. Realizing this idea requires addressing two main challenges:

- How to identify program properties that are (a) likely to hold (i.e., likely invariants) and (b) significantly reduce the points-to set size?
- How to design an efficient monitor and fallback mechanism for the unlikely case that the likely invariants do not hold?

Overview. Depending on the use case, the pointer analysis results are used to instrument or transform the application code. For instance, a CFI implementation will add branch checks to the instrumented code, which dynamically blocks unsafe jumps. Hence, the IGO fallback mechanism for disproven invariants requires multiple instrumented versions of the program, i.e., memory views, and a mechanism to switch between them dynamically. IGO performs the points-to analysis in three stages. First, IGO performs the standard pointer analysis and generates the memory view corresponding to this analysis. We call this memory view the Fallback Memory View. Then we perform the pointer analysis assuming that the selected likely invariants hold, generating the Optimistic Memory View. Finally, IGO inserts runtime monitors for each likely invariant that monitor violations of the likely invariants and switch from the optimistic MV to fallback MV. Figure 4 presents the high-level interaction between these different phases.

Figure 5 shows how the typical pointer analysis algorithm is modified to perform IGO pointer analysis using likely invariants. As discussed in §2.1, the analysis generates new derived constraints as well as new pointer relationships during constraint solving. Assume that the application program has
two pointers $p$ and $q$, and two objects $o_1$ and $o_2$. A particular iteration of the points-to solver derives the relationship $o_2 \in Pts(q)$. If the IGO analysis believes that this derivation is likely imprecise, it simply removes this derivation from the points-to set of $q$ and inserts a runtime monitor that monitors the program execution for the condition where $q = \& o_2$. After this filtration, the pointer analysis proceeds without further changes until the next likely invariant is encountered.

4 Kaleidoscope Design

This section presents the design of Kaleidoscope’s pointer analysis. We first discuss our systematic introspection method for selecting IGO likely invariants. Then, we discuss the three types of likely invariants policies: the arbitrary pointer arithmetic, the positive weight cycles, and the context sensitivity policies.

4.1 Pointer Analysis Introspection

In order to understand the exact points during pointer analysis that cause an imprecision explosion, we instrumented SVF [10], the state-of-the-art pointer analysis framework to track all updates to the points-to sets of the various pointers under analysis. SVF uses a standard field-sensitive, flow- and context-insensitive Andersen’s [15] algorithm. To gain better visibility into the sources of imprecision, we modified the generation of derived constraint edges to record the original primitive constraint edges that were instrumental in their derivation, thus recording their origin information. If there are multiple paths along which the same derived constraint is generated, we retain the five most recent paths, which we found to be enough for our purposes. The overhead of recording this information is non-trivial, but because this introspection is performed only to inform our choice of likely invariants, it does not have any significant impact.

A pointer’s points-to set can be updated by solving a primitive Copy or Field-Of constraint. Similarly, cycle detection and collapse [31], an optimization technique, also updates the points to set of the pointers in the cycle. During cycle collapse, the points-to sets of all pointer nodes in the cycle are merged, and the cycle is replaced by a single pointer node. Thus, to introspect how the points-to sets of different pointers evolve during multiple iterations of the pointer analysis algorithm, we instrument the resolution rules and the cycle collapse code to record the number of objects that are added to the target pointer’s points-to set.

Our introspection framework triggers alerts when it encounters situations that are indicative of imprecision. In particular, it alerts if the number of new objects added to the points-to set of a pointer crosses a pre-configured threshold (configured to vary between 100 and 1000 depending on the program size). The addition of objects of different unrelated types to the same points-to set also indicates imprecision. Therefore, if the points-to set is updated to contain objects of more than a preconfigured number of different types (configured to vary between 10 and 50), our system also registers an alert. If the triggering constraint edge is a derived constraint, our introspection framework automatically backtracks from the particular derived constraint till it reaches the primitive constraint that resulted in the addition of the derived constraint. To ensure backtracking termination, we impose a limit of five levels.

The alerts generated by this automatic introspection are then manually analyzed to identify the pointer analysis precision bottlenecks. We performed a preliminary study using this introspection framework on two large codebases, the Nginx web server, and a tiny build of the Linux kernel. This study informed our selection of likely invariant policies.

Observation. Our first observation was that a significantly high percentage of imprecision was caused by the loss of field sensitivity for struct objects under certain circumstances. Particularly, if struct objects containing function pointers lose field sensitivity, it introduces imprecision in the callgraph that further compounds imprecision. We further observed that objects lose field sensitivity because of either (a) arbitrary pointer arithmetic operations determined to operate on these struct objects, or (b) collapsing of positive weight cycles in the constraint graph.

We also observed that invocations of a small set of functions from different calling contexts cause a significant loss in precision due to context insensitivity. These functions typically copy one of the function arguments to a field of an object referred to by another function argument. The next sections discuss these cases that cause field and context sensitivity loss and how Kaleidoscope mitigates them.

4.2 Arbitrary Pointer Arithmetic

Ideally, a field-sensitive static pointer analysis would always distinguish between individual fields of an object because these individual fields might themselves hold pointer values. Achieving field sensitivity is fairly trivial if the fields are accessed by their field name, such as $obj.f1$, but, unfortunately, other non-obvious ways to access fields complicate this analysis.
Most complex real-world applications extensively use pointer arithmetic, where a pointer is used to incrementally traverse the elements of an array of simple or complex types. While less commonly used, it is also legal in the C language to ignore the actual struct types and use arbitrary pointer arithmetic, in the form \( *(p+i) = \&k \), to access individual fields of a struct object, when \( p \) is a char\* or int\* pointer pointing to the base of the struct object, and \( i \) is a variable with an arbitrary value \([2, 3]\). So pointer analysis frameworks must support this programming construct even for struct object types. However, in these cases, pointer analysis techniques typically find it difficult to distinguish between different fields, especially if the value of \( i \) cannot be easily computed statically and simply turn the entire struct object field-insensitive.

Figure 6 shows a simplified example from the Lighttpd codebase. The function `http_write_header` copies elements from a char array pointed to by pointer `ds->ptr` to another char array pointed to by pointer `s`. Now imagine that because of imprecision, the analysis determines that the pointer `s` might also point to the plugin objects `mod_auth` and `mod_cgi`. Without any further mitigation, the analysis has no option but to assume that the statement \( *(s+i) \) is accessing individual fields of the objects, `mod_auth` and `mod_cgi`, and must, therefore, turn these objects field-insensitive (if it cannot determine the value of \( i \)). Because these objects contain function pointers, turning these objects field insensitive has a cascading effect on imprecision.

**Key Insight.** Analysis imprecision has a critical side effect on how pointer analysis techniques handle pointer arithmetic. The pointer analysis might imprecisely conclude that the individual fields of a struct object `obj` are accessed using arbitrary pointer arithmetic, when at runtime, the pointer in question does not even point to `obj` at all. Nevertheless, because the pointer analysis algorithm has no way of pinpointing these imprecise derivations, `obj` unnecessarily has to be turned field insensitive, resulting in further precision loss. This, in turn, makes all other accesses to `obj` field insensitive, including explicit field accesses (e.g., `obj.fl`).

We observe that while it is challenging for the static pointer analysis to determine if a points-to relationship is imprecise, this information is easily available during program execution. Moreover, because such field-level accesses using arbitrary pointer arithmetic are rare, the likelihood of it happening at runtime is very low.

**Likely Invariant.** In order to limit the compounding of imprecision due to arbitrary pointer arithmetic on struct objects, we define our first likely invariant as follows:

A pointer, to which an arbitrary offset is added or subtracted, is only used to access elements of an array (consisting of primitives or complex objects), and not fields of a struct object.

If the analysis concludes that such a pointer is used to access individual fields of a struct object, the IGO optimistic analysis considers this points-to relationship to be the result of imprecision and ignores it. Note that because our likely invariant is focused primarily on the imprecision introduced by field insensitivity, it makes optimistic assumptions only about these pointers not accessing individual fields of objects. No assumptions are made about arbitrary pointer arithmetic used to iterate over individual elements within arrays of struct objects.

When performing the optimistic analysis, we filter all struct type objects from the points-to sets for the pointers operated on by arbitrary arithmetic and insert likely invariants that are monitored at runtime. For example, in Figure 6, based on the type information of the object, the optimistic analysis filters the imprecise targets `mod_auth` and `mod_cgi` from the points-to set of `s` as these are of struct type. This allows the optimistic MV to retain field sensitivity for these objects, thereby ensuring higher precision as long as the likely invariant holds.

**Runtime Monitor.** Kaleidoscope instruments the pointer on which the pointer arithmetic is performed (in this case, the pointer `s`) with runtime monitors that verify that the pointer does not point to any of the filtered objects during program execution. Thus, in Figure 6, the runtime monitor checks if the pointer `s` points to `mod_auth` and `mod_cgi`. If at runtime, the monitor observes `s` to refer to `mod_auth` or `mod_cgi`, Kaleidoscope will detect that the likely invariant has been violated and proceed to switch to the fallback MV. Our current implementation supports only two memory views, the optimistic MV, where all likely invariants hold, and the fallback MV, where none of the likely invariants hold. We discuss the use-case specific details of memory view switching in \S 5.

4.3 Positive Weight Cycles in the Constraint Graph

As discussed in §2.1, during the pointer analysis, as new points-to relationships are discovered, new derived constraint edges are added to the constraint graph. Occasionally, such derived constraint edges result in the formation of cycles in the pointer constraint graphs. If such cycles consist solely of Copy constraint edges, the points-to relationship derivations eventually converge when all the pointers involved have the same objects in their points-to sets. In fact, using cycle detection and elimination techniques \([25, 31, 45]\),

![Figure 5. Overview of IGO Analysis using likely invariants.](image-url)
the pointer analysis technique can safely detect and collapse these cycles into a single representative pointer node.

The PWC Challenge. A particularly insidious form of such a cycle is the positive weight cycle [43] (PWC). A PWC is a cycle that includes at least one Field-Of constraint edge (in the form of \( q = \& (p->f1) \)). Even though the exact field index is known, the presence of such a Field-Of constraint edge inside a cycle presents additional challenges. As shown by Pearce et al. [43], if no further mitigations are in place, this potentially leads to infinite derivations, as the points-to sets of the pointers involved do not ever converge, repeatedly accessing deeper and deeper nested fields of the objects in the points-to set of \( p \).

For example, in Figure 7, the PWC consists of \( C1, F1 \), and \( C2 \). Imagine the heap pointer \( H1 \) points to object \( obj \). The constraints \( C1, F1 \), and \( C2 \) will be solved repeated, accessing deeper nested fields, \( obj.f2, obj.f2.f2 \), and so on, and never converge. Note that if the Field-Of constraint edge was a Copy constraint edge, the points-to sets would have converged with all the points-to sets containing \( obj \).

To avoid this problem, typically, pointer analysis frameworks turn all targets of such Field-Of constraint edges inside a PWC, into field insensitive. This allows the analysis to convert the Field-Of constraint edge itself to a Copy constraint edge, thereby converting the PWC into a simple cycle, that can be safely collapsed into a single node. However, this approach leads to precision loss, and potentially sets the stage for the generation of further PWCs down the line.

**Key Insight.** Similar to the case of arbitrary pointer arithmetic, the pointer analysis technique is unable to determine if a PWC is the result of actual points-to relationships that hold at runtime or is caused purely because of imprecision. While it is theoretically true that if such a PWC were to actually occur at runtime, it would cause infinite derivations, the very fact that a PWC can lead to an improbable situation such as infinite derivations, itself indicates that the formation of a PWC is very likely the result of imprecision.

**Likely Invariant.** The second likely invariant targets imprecision resulting from the generation of PWCs, as follows:

Positive weight cycles (PWCs) in the pointer constraint graph stem from imprecision and do not occur at runtime.

Through experimentation, we found that it was not feasible to predict which derived Copy constraint edge in the PWC was likely to be added due to imprecision. However, we observe that turning the pointee objects of the Field-Of constraint edges can be deferred until the PWC is actually observed to occur at runtime. Therefore, during the optimistic analysis, when Kaleidoscope observes the formation of PWCs, it simply solves each constraint edge involved in the PWC, but defers collapsing the PWC and turning any of the involved objects to field insensitive. Thus, the optimistic MV does not contain any imprecision resulting from PWCs. Non-PWC cycles that do not contain a Field-Of edge are collapsed and merged normally.

Figure 7 shows a simplified PWC from the LibPNG [7] library. In the case of this PWC, heap imprecision at the heap allocation function \( png_alloc \) causes the same heap object \( H1 \) to be returned at both call sites, \( P6 \) and \( P7 \), even though \( P6 \) allocates a pointer to a \( compression_state \) object, and \( P7 \) allocates a pointer to a \( int \) primitive variable. This results in the addition of the same object \( H1 \) in the points-to sets of both \( s1 \) and \( q \). When the \( Load \) and \( Store \) constraint edges \( L1 \) and \( S1 \), corresponding to program statements \( P10 \) and \( P12 \) are solved, this ultimately results in the addition of derived Copy edges \( s1 \) and \( s2 \). These derived Copy constraints along with the Field-Of edge (corresponding to program statement \( P11 \)) from \( a \) to \( b \) completes the PWC.

A PWC can form at runtime only if a field address generated by a program statement such as \( k = \& (p->f) \), is reused as the base pointer to generate the address of a deeper nested field, by a subsequent execution of the same program statement. For example, the statement \( P11 \) would result in a PWC only if the generated field address \( b \) is reused as the base pointer \( c2 \) to derive the field address of a nested inner field. Multiple such field accesses in a cycle can interact with each other and form a PWC. Thus, the formation of PWCs at runtime can be detected by observing the base pointers and the field addresses generated by the field accesses involved in the PWC. In the case of Figure 7, the heap contexts represented statically by \( H1 \) will be separated at runtime. Therefore, the PWC will not form, and the likely invariant will continue to hold.

**Runtime Monitor.** Kaleidoscope inserts runtime monitors to record both the base pointer and the field address generated by the field access instructions in the PWC. If these runtime monitors observe a field address generated by an field access inside a PWC, being reused as the base pointer in the computation of a nested field address, it detects the formation of the PWC. This, in turn, invokes the use-case specific memory switcher to switch to the fallback MV.
4.4 Context Sensitivity

Implementing full context-sensitivity requires each function to be re-analyzed at each callsite where it is invoked. This requires significant recomputation of the same function under different calling contexts, especially in the case of nested function calls, leading to scalability issues, ultimately preventing its adoption for large codebases [24, 48].

**Key Insight.** Our key insight stems from the observation that the context-insensitive analysis of only a few program statements, in a handful of functions, results in a significant loss of overall precision. For example, the typical case of updating global values via pointer arguments of a function does not cause imprecision. Only in cases where there exists a data flow originating from a pointer argument to the value returned by the function, will invoking the function from different callsites with cause imprecision. This data flow can be via an explicit return statement or by copying the input pointer argument to another argument. The precision loss occurs because the pointer arguments at each of the callsites will be returned to each of the return sites. We call pointer arguments that are part of such data flows **Precision Critical Arguments**.

Therefore, instead of analyzing all instructions of all functions in a context-sensitive manner, Kaleidoscope makes optimistic assumptions about these precision critical arguments, thereby allowing us to maintain significant precision without the cost of a full context-sensitive analysis.

For example, consider the simplified code snippet from the Libevent [5] library, shown in Figure 8. The Libevent library provides support for handling events asynchronously. The application first sets up one or more event bases, and then sets up callback functions for them. The function `ev_queue_insert` stores the callback argument `cb`, to a field `cbs` of event base argument `b`. The Libevent source code invokes this function from two different callsites, to set up two different event bases, `global_base` and `evdns_base` with their respective callback functions. Due to the program statements being analyzed in a context-insensitive way, the pointer analysis will resolve both `global_base.cbs` and `evdns_base.cbs` to point to both the callback objects `cb1` and `cb2`. Therefore, in this case, the formal arguments `b` and `cb` are the precision critical arguments for the function `ev_queue_insert`.

Complex pointer operations involving such precision critical arguments are challenging to reason about without expensive analysis. In particular, these arguments can themselves have their addresses stored in other double or triple indirection pointers, and be modified to point to other objects via these multi-level pointer references. However, a lightweight data flow analysis of these pointer arguments can identify the simple patterns where a pointer argument is either returned by the function, or copied to another pointer argument.

**Likely Invariant.** In order to mitigate the imprecision caused by the context-insensitive analysis of the precision critical arguments, we define a third likely invariant as follows:

**Precision critical arguments of a function are not updated to point to any other object inside the called function.**

Thus, in the case of the code snippet in Figure 8, Kaleidoscope will assume that the calls to `INCR_EVENT_COUNT` in statement P12 does not update the argument pointers `b` or `cb` to point to any other objects other than the objects they are initialized with at the callsites at P8 or P9.

During optimistic analysis, Kaleidoscope directly connects the actual arguments at the callsites, bypassing the store or return statements inside the function. To achieve this, in the

---

**Figure 7.** Example showing how heap imprecision leads to a positive weight cycle [43].

**Figure 8.** Example showing how context insensitivity leads to imprecision.
constraint graph, we create new dummy nodes at each call-site. We then connect the actual arguments, at each call-site, via these dummy nodes. In the case of Figure 8, the actual arguments at each call-site are represented by the nodes b@P8, b@P9, cb@P8, and cb@P9. To represent the store instruction P16, we generate dummy field nodes cbs0 and cbs1 for nodes b@P8 and b@P9 respectively, and connect them to cb@P8 and cb@P9. Then, while solving the constraint graph, the store constraint edge corresponding to original store statement P16 is ignored. This ensures that the pointer relationships that are updated by the statement P16 are analyzed in a context-sensitive manner.

Runtime Monitor. To observe the state of the likely invariants at runtime, we instrument the call-sites of the target function to record the actual arguments. The return and store statements involving precision-critical arguments are instrumented with runtime monitors to check if the argument pointers are modified to point to different objects than what was recorded at the call-site. If the values are observed to not match, it triggers a switch to the fallback MV.

5 Case Study: Control Flow Integrity

Control flow integrity (CFI) protects against control flow hijacking attacks by ensuring that the runtime control flow follows valid, precomputed paths, including indirect call-sites. However, imprecision in static pointer analysis makes the generation of effective CFI policies challenging. To mitigate this precision challenge, we use the IGO pointer analysis to resolve the targets of the function pointers and generate significantly tighter control flow integrity policies.

Consider the simplified example from the MbedTLS codebase shown in Figure 9. The function mbedtls_ctr_drbg_reseed has an indirect call site ctx->f_entropy. The indirect function call is instrumented with the CFI check, which ensures that the function pointer can only refer to precomputed valid function targets. The set of valid function targets for each function pointer is derived by the IGO pointer analysis. Therefore, in the CFI case, a memory view consists of the set of valid functions that a function pointer can refer to. We use Kaleidoscope to generate both the optimistic MV and the fallback MV consisting of the function pointer targets for each such indirect call-site.

When the application starts, it begins with the optimistic MV, and the indirect call-site is permitted to invoke only mbedtls_entropy_func. During runtime, if a likely invariant is violated, the memory view is switched via a secure MV Switch operation, and the indirect call-site is now permitted to invoke mbedtls_entropy_func, mbedtls_net_send, and mbedtls_net_receive. We implemented Kaleidoscope using the LLVM 12 toolchain and SVF framework.

Security Analysis. Kaleidoscope strengthens the traditional CFI properties by narrowing down the set of valid jump targets through the increased analysis precision. This benefit is dependent on Kaleidoscope preserving during run-time the integrity of the memory view, even under attack. Kaleidoscope ensures that corrupting a function pointer by a buffer overflow does not cause the memory view to switch to the fallback MV. This is because Kaleidoscope does not use likely invariants on the targets of function pointers. Instead, it uses likely invariants on the pointer relationships, which have a cascading imprecision effect, eventually adding a target function to the equivalence class of a function pointer during static analysis. For example, a PWC might cause certain objects to lose field sensitivity, eventually leading to the addition of a function to the set of valid targets of an indirect call-site. In this case, the likely invariant is added on the PWC, and the attacker must perform a series of data-only attacks first to violate the likely invariant and create the PWC at runtime. And only then can they mount the control flow attack by overwriting the function pointer.

6 Implementation

We implement Kaleidoscope using the LLVM 12 toolchain. The Kaleidoscope IGO pointer analysis is based on the SVF framework, which applies Andersen’s pointer analysis algorithm on LLVM IR bitcode. Kaleidoscope modifies SVF’s pointer analysis algorithm to insert the likely invariants and their monitors.
Table 2. Evaluation Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbedTLS</td>
<td>SSL Library</td>
<td>73528</td>
</tr>
<tr>
<td>Libtiff</td>
<td>Library for manipulating TIFF files</td>
<td>34221</td>
</tr>
<tr>
<td>Curl</td>
<td>Web Downloader</td>
<td>21258</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>HTTP Web Server</td>
<td>77912</td>
</tr>
<tr>
<td>Memcached</td>
<td>Key-value Store</td>
<td>75049</td>
</tr>
<tr>
<td>LibPNG</td>
<td>Library for manipulating PNG files</td>
<td>58831</td>
</tr>
<tr>
<td>Libxml</td>
<td>Library for manipulating XML files</td>
<td>97929</td>
</tr>
<tr>
<td>Wget</td>
<td>Webpage Downloader</td>
<td>65490</td>
</tr>
<tr>
<td>TinyDTLS</td>
<td>Library for Datagram Transport Layer Security</td>
<td>10207</td>
</tr>
</tbody>
</table>

Heap Type Detection Pointer Arithmetic likely invariants for heap objects required special handling. Kaleidoscope determines the type of heap objects so that they can be filtered by the pointer arithmetic likely invariant. To determine the type of the object created at the callsite, we extract the type name passed to the sizeof operator at these heap allocation calls. Because the sizeof operator is lowered to constant integer values by the Clang front-end, we modified the Clang front-end to retain this type information as metadata in the LLVM IR bitcode. We use an interprocedural analysis to propagate the heap-type information. If the type information for a heap allocation cannot be determined, then the objects allocated at that callsite are never filtered, thus ensuring soundness.

7 Evaluation

In this section, we describe our evaluation of Kaleidoscope.

Evaluation Setup. We ran all experiments on machines with an AMD EPYC 7402P (24 core) CPU and 128 GB of RAM. These machines ran Ubuntu 22.04, with Linux kernel 5.15.0-27. For server-client experiments, both machines were on the same local network.

Applications Table 2 lists the applications used to evaluate Kaleidoscope. In cases where the application depended on other libraries, we used Link Time Optimization (LTO) to generate the linked LLVM Intermediate Representation bitcode of the application and its dependent libraries. For library targets, such as Libtiff and Libpng, we used a representative application that uses the library to evaluate Kaleidoscope.

7.1 Precision Improvements

Table 3 reports the average and maximum points-to set size for each application under Kaleidoscope and the baseline analysis. A smaller points-to set size indicates a higher degree of precision. Kaleidoscope’s likely invariants results in improved field and context sensitivity, and this causes an improvement in precision. Across all applications, the average points-to set size reduced by 13.15x and the maximum points-to set size reduced by 1.25x, demonstrating the effectiveness of Kaleidoscope. Figure 10 presents the box-plot distribution of the points-to set sizes of all pointer variables in the applications. Kaleidoscope reduces both the median points-to set size as well as the outliers, thus mitigating the extreme imprecision cases where certain pointer variables have very large points-to sets. Depending on the programming patterns used by the application, some of the likely invariants are more impactful than others. For example, in the case of Libtiff, the context-sensitivity and the PWC likely invariants provide the majority of the precision improvements, whereas in the case of MbedTLS, all the likely invariants must be enabled to observe a significant reduction in the points-to set sizes.

In the case of Wget and TinyDTLS, Kaleidoscope prevents imprecision that causes multiple pointers to share the largest points-to set, but it does not decrease the size of the largest points-to set. Therefore, the maximum points-to set size does not show any improvement under Kaleidoscope, but the average points-to set size reduces by 1.83x and 3.89x compared to the baseline analysis, respectively.

7.2 Case Study: Control Flow Integrity

Figure 11 shows the average number of targets for indirect callsites for each application under IGO’s optimistic analysis. As expected, in the case of all 9 applications, Kaleidoscope provides a lower number of indirect call-site targets, thus ensuring a higher degree of security. All applications except Curl, Wget, and Lighttpd show significant precision improvements. In the cases of these applications, we observed that even though Kaleidoscope identifies opportunities to make optimistic assumptions and insert likely invariants, the precision improvement provided by these likely invariants are eventually negated by certain programming patterns.

Lighttpd and Wget use function pointers stored in arrays to implement callbacks. Lighttpd uses these callbacks to implement a plugin architecture, while Wget uses callbacks to implement the functionalities of the command line options. Because our baseline analysis itself is array-index insensitive, Kaleidoscope is forced to treat each of these function pointers as the same, thus losing all benefits of preserving field sensitivity. In the case of Curl, heap allocation functions such as malloc and calloc accessed via function pointers, account for the majority of the imprecision. Resolving these function pointers itself requires complete pointer analysis, thus Kaleidoscope’s context-sensitivity likely invariants do not sufficiently handle such patterns.

Figure 12 provides a detailed view of the distribution of the number of indirect callsite targets. Note that the reduction in the points-to set sizes of all pointers shown in Figure 10 does not necessarily correlate with the reduction in the indirect callsite targets because imprecision can sometimes be localized only to function pointers. Our observation shows that the points-to sets of function pointers often converge and typically multiple pointers end up sharing the same points-to set, resulting in a narrow interquartile range (IQR). As depicted by the results, Kaleidoscope reduces both the
median number of indirect callsite targets and also the larger outliers.

**Performance Overhead.** We benchmark the CFI-hardened applications using standard benchmarking tools. For the MbedTLS SSL server, which is an SSL server provided by the MbedTLS library that can serve HTTP requests over a secure SSL connection, we use the SSL client that is provided with the library to send 100000 requests. To benchmark Lighttpd, we use ApacheBench [1] to request a 4KB file 10000 times. We benchmark Memcached using the memaslap [6] tool and make 200000 requests with a 90:10 get/set ratio [41]. We also enabled multi-get requests. For Wget and Curl, we downloaded a 4KB file 10000 times from a local web server. For Libtiff, we used the tiffcrop utility to crop 4KB TIFF images, and for Libpng, we used the pngcmp tool to copy 4KB PNG images. For Libxml, we used the xmllint tool to validate an 8KB XML file. We performed 10000 requests to the TinyDTLS server. Each experiment was performed 10 times, and the average was reported.
Whenever possible, we use a diverse set of inputs, which we derive via fuzzing, to increase the code coverage and ensure that the application is not under-exercised. We present the details of our fuzzing setup in §7.3. In the case of Lighttpd and MbedTLS, due to the limitations of the benchmarking tools, which limit the types of requests that can be sent, we could not incorporate all possible inputs during our evaluation. ApacheBench [1] does not support benchmarking multiple URLs with different command line options. Similarly, the memaslap tool does not support commands such as stats and flush. Table 4 reports the branch and runtime monitor coverage statistics of the inputs used in our CFI evaluation. On average, 33.08% of all code branches were executed. Moreover, 50.72% of all runtime monitors were executed, demonstrating that the applications were not under-exercised, and the reported performance overhead is representative of the overhead of Kaleidoscope.

Figure 13 shows the runtime performance overhead of Kaleidoscope over a CFI framework hardened by the baseline points-to analysis. As shown, the performance overhead of Kaleidoscope is minimal compared to the baseline, with a maximum overhead of 9.67% observed for Memcached. The average throughput overhead was observed to be 5.45%. This indicates that the cost of the runtime monitor checks is low. And indeed, the maximum number of monitor checks performed at runtime is 4.78% of all memory operations (in the case of Curl). We observe that none of the likely invariants selected by Kaleidoscope were violated at runtime. Therefore, the CFI policies generated using the optimistic analysis are always in place, thereby providing higher security.
Table 4. Branch and runtime monitor coverage for CFI evaluation.

<table>
<thead>
<tr>
<th>Application</th>
<th>Code Branches</th>
<th>Runtime Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbedTLS</td>
<td>36096 11810 32.72%</td>
<td>2172 842 38.77%</td>
</tr>
<tr>
<td>Libtiff</td>
<td>13853 5392 38.92%</td>
<td>821 671 81.73%</td>
</tr>
<tr>
<td>Curl</td>
<td>23752 7921 33.35%</td>
<td>1871 991 52.97%</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>19784 3531 17.85%</td>
<td>1498 492 32.84%</td>
</tr>
<tr>
<td>Memcached</td>
<td>20422 3531 17.85%</td>
<td>1498 492 32.84%</td>
</tr>
<tr>
<td>LibPNG</td>
<td>16065 8810 54.84%</td>
<td>474 177 37.34%</td>
</tr>
<tr>
<td>Libxml</td>
<td>31465 5391 17.13%</td>
<td>1254 638 50.88%</td>
</tr>
<tr>
<td>Wget</td>
<td>13028 7811 59.96%</td>
<td>586 328 55.97%</td>
</tr>
<tr>
<td>TinyDTLS</td>
<td>8027 1121 13.97%</td>
<td>101 38 37.62%</td>
</tr>
</tbody>
</table>

Table 5. Branch and runtime monitor coverage for likely invariant validation through fuzzing.

<table>
<thead>
<tr>
<th>Application</th>
<th>Code Branches</th>
<th>Runtime Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>MbedTLS</td>
<td>36096 20388 56.48%</td>
<td>2172 1289 59.35%</td>
</tr>
<tr>
<td>Libtiff</td>
<td>13853 6291 45.41%</td>
<td>821 689 83.92%</td>
</tr>
<tr>
<td>Curl</td>
<td>23752 8621 37.14%</td>
<td>1871 1269 67.82%</td>
</tr>
<tr>
<td>Lighttpd</td>
<td>19784 7807 39.46%</td>
<td>1498 876 58.48%</td>
</tr>
<tr>
<td>Memcached</td>
<td>20422 7807 39.46%</td>
<td>1498 876 58.48%</td>
</tr>
<tr>
<td>LibPNG</td>
<td>16065 10326 64.28%</td>
<td>474 198 41.77%</td>
</tr>
<tr>
<td>Libxml</td>
<td>31465 6876 21.85%</td>
<td>1254 998 79.59%</td>
</tr>
<tr>
<td>Wget</td>
<td>13028 8976 68.90%</td>
<td>586 345 58.87%</td>
</tr>
<tr>
<td>TinyDTLS</td>
<td>8027 2184 27.21%</td>
<td>101 38 37.62%</td>
</tr>
</tbody>
</table>

7.3 Likely Invariant Validation through Fuzzing

To further demonstrate that the likely invariants hold under a variety of inputs we used the AFL++ [26] fuzzier to generate a variety of test cases and increase the code coverage. Every application was seeded with a list of inputs covering the popular options provided in their man page. The branch-level coverage and the executed runtime monitor statistics after running a 24-hour [18, 44] fuzzing session are reported in Table 5. Across all applications, 46.47% of total branches and 66.56% of all runtime monitors were executed and none of the likely invariants were violated. This increases our confidence in the selection of the likely invariants.

8 Discussion

Finer Grained Fallback Mechanisms. Finer grained fallback mechanisms can potentially allow Kaleidoscope to gradually degrade precision as likely invariants are observed to fail. A possible approach for such a mechanism is to pre-generate the memory views corresponding to each likely invariant. This would potentially lead to an increase in the application binary size. Therefore alternatively, incremental pointer analysis techniques [38] can be used on likely-invariant violations to update the points-to sets on the fly.

Other Use Cases. Applying Kaleidoscope to use cases where the fallback MV can be statically determined is straightforward. This includes security use cases such as bug finding [29, 30, 39] and software debloating [12, 27, 28]. However, in certain use cases, the fallback MV can depend on runtime state whose generation itself is predicated on the fallback MV’s pointer analysis. Such cases must be handled on a per use-case basis to recover this runtime state. This includes compiler optimizations which optimistically transform code to elide the generation of certain runtime state that is later required by the fallback MV. Systems that only use pointer analysis to statically prove properties of the system are not a good fit for Kaleidoscope.

Kaleidoscope can augment bug-finding and fuzzing tools to provide more precise, optimistic points-to results. In the case of debloating, Kaleidoscope can rely on dynamic debloating mechanisms, which simply mark the optimistically
debloated code as inaccessible [12–14], instead of removing them entirely. If a likely invariant is violated at runtime, the fallback mechanism can restore the executable access to this code.

Other Language Support. Kaleidoscope operates at the LLVM IR level and can natively support any language that has an LLVM frontend. Likely invariants for these languages can be derived using our pointer introspection framework.

9 Related Work

Scalability Improvements. Andersen’s pointer analysis [15] is the most precise class of solvers but has a cubic runtime. Optimizations such as cycle detection [31], wave propagation [45], and partial update solving [40] use graph optimizations to improve the performance of Andersen’s analysis. Steensgaard’s [49] pointer analysis runs in linear time but is imprecise. Other techniques [22] attempt to reduce the imprecision in such unification-based approaches. DEA [36] proposed a faster technique for solving PWC cycles, but does not improve precision.

Sensitivity Improvements. Various techniques [32, 34, 52] have been proposed to improve the scalability of flow and context sensitive pointer analysis algorithms. Introspective analysis [47] uses feedback from the pointer analysis process to fine-tune context sensitivity. Pearce et al. [43] extend Andersen’s analysis to support field sensitivity. CClyzer [16] presents a structure sensitive points-to analysis. Zipper [37] proposes a precision-guided context sensitivity for Java programs. Lu et al. [33] present a tunable technique for object context sensitivity in Java programs.

Selective context sensitivity techniques offer similar capabilities as Kaleidoscope’s context sensitivity likely-invariant, but because they target Java applications, they are not directly applicable to C/C++ codebases. For example, Zipper [37] uses the Java Class information to identify the precision loss patterns, whereas the type information is often obfuscated in C applications. Moreover, Kaleidoscope allows the functions with precision critical arguments to contain indirect function calls. Applying selective context sensitivity techniques to such functions is challenging because this requires a context sensitive analysis for all possible targets of such indirect callsites. Unlike Java, where the number of targets of virtual function calls is limited by the class hierarchy, in C/C++ codebases, each indirect call site can have a significantly large number of targets, each of which must be analyzed in a context-sensitive manner, thus complicating the analysis.

Hybrid Approaches. Various hybrid approaches which combine other techniques with pointer analysis to improve precision have been proposed. Past-Sensitive pointer analysis [50] uses symbolic execution to improve the precision of pointer analysis. Similarly, Iodine [17] and Hybrid pruning [21] applies dynamic profiling to derive data flow and pointer relationships, respectively. Optimistic Hybrid Analysis [23], uses predicated static analysis to accelerate dynamic analysis.

10 Conclusion

We implemented Kaleidoscope, a system that improves pointer analysis precision by making optimistic assumptions during the pointer analysis that it later validates at runtime. We showed that Kaleidoscope (a) reduces the average points-to set size by 13.15× across a range of benchmarks and (b) enables CFI implementations with significantly more restrictive policies.

11 Acknowledgments

We thank all anonymous reviewers for their feedback, which greatly improved the paper. We also thank Iulian Neamtiu for shepherding our paper. This work was funded in part by award CNS-2127309 to the Computing Research Association for the CIFellows Project. In addition, this work was funded in part by the National Science Foundation (NSF) under grants CNS-2140305 and CNS-2145888.

A Artifact Appendix

A.1 Abstract

The artifact comprises the workflow for Kaleidoscope. Kaleidoscope is based on a custom LLVM compiler and the SVF pointer analysis framework. In this artifact, we provide the source code and scripts to build the custom LLVM compiler, the Kaleidoscope codebase, and scripts to compile and analyze applications using Kaleidoscope. The source code for the custom LLVM compiler and the Kaleidoscope codebase is publicly available on Github.

A.2 Artifact check-list (meta-information)

- Program: kaleidoscope-artifact.
- Binary: The custom LLVM 12 compiler artifacts clang, llvm-link, opt, and the Kaleidoscope binary.
- Run-time environment: Ubuntu 22.04, Docker
- Hardware: 64 GB RAM, 300 GB hard-disk
- Output: CSV files containing the points-to sets and CFI policies
- How much disk space required (approximately)?: ~100 GB
- How much time is needed to prepare workflow (approximately)?: ~3-4 hours
- How much time is needed to complete experiments (approximately)?: ~3 hours
- Publicly available?: Yes
- Code licenses (if publicly available)?: Apache License
- Archived (provide DOI)?: https://zenodo.org/records/10841643

A.3 Description

We evaluated Kaleidoscope on a server machine with 24-core AMD EPYC 7402P core, 64 GB RAM, and 1 TB hard disk. Kaleidoscope requires a customized LLVM 12 compiler
Kaleidoscope: Precise Invariant-Guided Pointer Analysis

Figure 10 was tested on a machine running Ubuntu 22.04.

A.5 Evaluation and expected results

The artifact evaluation will cover the following aspects that are automatically pulled by the provided scripts.

A.3.1 How to access

The artifact is available at https://github.com/rssys/kaleidoscope-artsafs/. The source code for the modified LLVM 12 compiler and the Kaleidoscope pointer analysis framework are provided as git sub-modules which are automatically pulled by the provided scripts.

A.3.2 Hardware dependencies

The artifact evaluation requires 64 GB RAM and ~100GB hard disk.

A.3.3 Software dependencies

The artifact evaluation was tested on a machine running Ubuntu 22.04.

A.4 Installation

Please see the README file at https://github.com/rssys/kaleidoscope-artsafs/blob/main/README.md for instructions on how to install the artifact.

A.5 Evaluation and expected results

The artifact evaluation will cover the following aspects that serve as the key results of this paper: (1) the reduction in the maximum and average points-to set sizes using the full Kaleidoscope system (Table 3), (2) the average reduction in the average number of CFI targets (Figure 11), and (3) the distributions of the points-to set sizes and the numbers of CFI targets (Figure 10, Figure 12), for sample applications.

The artifact provides the scripts to automatically run the analysis pipeline and reproduce the results. For more details regarding the evaluation, please refer to the README.md file in https://github.com/rssys/kaleidoscope-artsafs/.

References


