PhaseSeed: Precise Call Graph Construction for Split-Phase Applications using Dynamic Seeding

Tapti Palit
UC Davis
tpalit@ucdavis.edu

Seyedhamed Ghavamnia
Bloomberg
sghavamnia@bloomberg.net

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

***Abstract*—Precise and sound call graph construction is crucial for many software security mechanisms. Unfortunately, traditional static pointer analysis techniques used to generate application call graphs suffer from imprecision. These techniques are agnostic to the application's architecture and are designed for broad applicability. To mitigate this precision problem, we propose PhaseSeed, a novel technique that improves the accuracy of pointer analysis for *split-phase* applications, which have distinct initialization and processing phases. PhaseSeed analyzes the initialization phase dynamically, collecting the points-to relationships established at runtime. At the end of the initialization phase, it then *seeds* this information to a static analysis stage that performs pointer analysis for all code that stays in scope during the processing phase, improving precision. Our observations show that, given the same runtime configuration options, the points-to relationships established during the initialization phase remain constant across multiple runs. Therefore, PhaseSeed is sound with respect to a given initial configuration. We apply PhaseSeed to three security mechanisms: control flow integrity (CFI), software debloating, and system call filtering. PhaseSeed provides up to 92.6% precision improvement for CFI compared to static call graph construction techniques, and filters nine additional security-critical system calls when used to generate Seccomp profiles.**

1. Introduction

Application call graph construction is essential for multiple security defense mechanisms, including control flow integrity (CFI), software debloating, and system call filtering. Indirect control flow transfers require the use of static pointer analysis techniques to resolve their targets. Therefore, the security guarantees of these defense mechanisms rely on the accuracy of the underlying static pointer analysis technique. For example, CFI restricts the targets of indirect function calls to precomputed functions. The effectiveness of a CFI mechanism depends on the *precision* of this restriction—the *fewer* targets permitted at an indirect call site, the higher the security guarantees. On the other hand, to ensure correct application execution, CFI must be *sound* and never restrict valid indirect function calls. Precise and sound pointer analysis techniques are thus crucial for the effectiveness and correctness of security defense mechanisms.

Despite decades of research, achieving high precision while maintaining soundness remains challenging for static pointer analysis techniques [44], [69]. The traditional approach for improving the precision of pointer analysis is by increasing its *sensitivity* and incorporating additional program information For instance, context-sensitive pointer analysis [70], [78], [82] treats every invocation of a function as distinct, while path-sensitive analysis [52], [73] analyzes every *branch* of an `if` statement independently of the other. Increasing the sensitivity of pointer analysis, however, also increases the complexity of the analysis, limiting the scalability of these techniques to only small applications. Consequently, many software defenses [13], [18], [37], [50], [53], [54], [83] resort to using the type information of function call arguments to improve call graph precision by filtering potential targets of indirect call sites. However, as we demonstrate in Section 2.2, *statically* deriving accurate type information for all objects in C/C++ codebases is error-prone and results in unsound call graphs for type-based CFI approaches, such as Clang-CFI [3], [51].

Another promising direction is to augment static analysis with dynamic analysis [29], [39], [40] by running the application with certain inputs and reusing the dynamically observed points-to relationships to improve precision. However, naively combining static and dynamic analysis results and excluding all dynamically executed code from static analysis can lead to unsound results. The dynamically analyzed functions can be re-invoked with different arguments from calling contexts *unseen* during the training phase, resulting in new points-to relationships that are not captured. While such techniques are suitable for *best effort* approaches such as software testing, which can tolerate unsoundness, they cannot be applied to software defense mechanisms where soundness is critical.

To mitigate these challenges, we propose PhaseSeed, a novel technique for improving the precision of call graph extraction for *split-phase* applications. Split-phase applications [37], [68] consist of a distinct *initialization* stage followed by a *processing* stage. During initialization, the application instantiates various objects and function pointers that persist through its lifetime, whereas during the processing phase, the application serves various types of requests. Unlike traditional call graph construction approaches that are agnostic to the application's architecture, PhaseSeed leverages the split-phase nature of certain applications to improve precision and ensure soundness.

Instead of naively combining static and dynamic analysis results, PhaseSeed dynamically executes only the initialization phase, while statically analyzing the processing phase, *seeding* the static analysis phase with the dynamically resolved pointer targets of the executed paths. Once the dynamic results are seeded, the rest of the static pointer analysis proceeds unmodified. Unlike previous techniques that unconditionally exclude the dynamically executed code from the static analysis [29], [39], [40], PhaseSeed ensures that dynamically executed code is excluded from the static analysis *only* if it is *guaranteed* to be inaccessible from any remaining parts of the code that were not executed during the dynamic analysis. Consequently, the resulting analysis is always sound with respect to the provided runtime configuration, and retains the precision improvements provided by dynamic analysis, whenever possible.

During the initialization phase, the application instantiates various objects and function pointers that persist through the application's lifetime. Unlike the processing phase, which shows significant variation depending on the request types and the application state, as we discuss in Section 7, the points-to relationships established during the initialization phase are fully deterministic given an initial runtime configuration and do not depend on user inputs. Moreover, this initial runtime configuration can also potentially *disable* certain options, which allows the analysis to completely *remove* the corresponding code from the analysis scope.

Dynamically analyzing an application's initialization also provides complete type information for the heap objects allocated during this phase. Modern applications rely on complex memory allocation paradigms (e.g., memory pools, heap allocation wrappers) that obfuscate the flow of *type information* to heap allocation sites, making the static derivation of type information for these objects challenging. During dynamic execution, the obfuscated data flows corresponding to these complex memory allocation paradigms are easily resolved, and the type information is easily determined. Furthermore, PhaseSeed uses this type information in a *strictly* sound manner, instead of using it to filter function targets at indirect call sites.

PhaseSeed requires the programmer to just annotate the function that denotes the start of the processing phase and to provide the initial runtime configuration to launch the application. Given this information, PhaseSeed dynamically analyzes the application's initialization phase till the transition point is reached. Then, it inspects the application's memory state to collect the points-to relationships and type information for all in-memory objects. By performing an iterative analysis of the functions reachable from the transition point, PhaseSeed automatically and soundly derives the functions that may be accessed by the processing phase. Finally, PhaseSeed performs static analysis on these processing phase functions after *seeding* the analysis with the points-to relationships captured during dynamic analysis.

To demonstrate the benefits of PhaseSeed's improved precision we apply it to three security mechanisms: control flow integrity, software debloating, and system call filtering. Our results show that PhaseSeed improves the precision of CFI compared to standard static pointer analysis techniques by up to 92.6%, facilitates the removal of more code for software debloating, and can filter nine additional system calls for the applications in our evaluation set.

In summary, our main contributions include:

- We present PhaseSeed, a sound call graph extraction approach that improves the precision of pointer analysis by *seeding* pointer relationships obtained dynamically.
- We implemented a method for dynamically deriving heap object types, which is sound even in the presence of complex programming patterns that obfuscate type information.
- We apply PhaseSeed to improve the precision of CFI, software debloating, and system call filtering, demonstrating up to 92.6% precision improvement compared to state-of-the-art static call graph construction techniques.

2. Background

2.1. Static Pointer Analysis

Static pointer analysis is a technique for resolving the points-to targets of all pointers in an application at compilation time. Static pointer analysis consists of two stages. The first stage collects all pointer operations in the application, along with their operands, and stores them in a *constraint graph*. The second stage solves these relationships according to certain *constraint resolution rules*. The constraint resolution rules for the commonly used Andersen's pointer analysis [17] are specified in Appendix A.

Various works [48], [60], [62], [71], [73], [79] have attempted to improve both the precision and scalability of pointer analysis algorithms. However, achieving sound, high-precision results within reasonable analysis time remains a challenge. The fundamental reason for this imprecision is that due to the dependencies on runtime input, it is impossible to reason about pointer relationships with full precision statically [44]. Moreover, decisions such as whether to analyze each invocation of a function distinctly (context-sensitivity), and whether to consider the order of the program statements (flow-sensitivity), also impact the precision of the points-to results.

Indeed, context and flow sensitivity improves the precision of the pointer analysis. However, maintaining the calling context and flow information for complex code involving arbitrarily deep function calls and deeply nested loops and branches significantly increases the number of constraints in the constraint graph, and thus reduces the scalability of the analysis. Therefore, many pointer analysis clients [24], [37], [38] opt to use a context- and flow-insensitive pointer analysis instead, thus sacrificing precision.

2.2. Type Ambiguity in C/C++

Accurate type information can assist in pointer analysis. The type information is trivial to derive for stack and global variables because the type information is embedded in the variable definition, in the case of these variables. However, statically deriving the accurate type information for heap objects is challenging.

Type Punning C/C++ does not enforce type safety. Type-casting pointers of various types to and from `void*` is commonly used in most C/C++ codebases. Moreover, some C applications emulate object-oriented features such as type inheritance and polymorphism using a technique called *type punning*. Using this technique, the programmer can create a parent-child relationship between two `struct` types by embedding the *parent* struct type as the first field of the *child* struct type. Alternately, the child type could simply be defined to contain the same fields as its parent type as its initial subset of fields. This allows a pointer to the parent struct type to safely access an object of the child type as long as it only operates on the embedded parent type fields, facilitating the implementation of polymorphic functions in C.

For example, consider the simplified code snippet from the codebase of the popular Lighttpd web server shown in Figure 1. The type `data_array` *inherits* `data_unset` by embedding a field of type `data_unset`. In the function `initializeServer`, the pointer `yy1` (line 11) of type `data_unset` is used to initialize a heap object of type `data_array`. The object is cast to the *actual* `data_array` type on Line 16, when the function returns.

Heap object type information is often derived by performing a data flow analysis from the heap allocation call site (e.g., `malloc`) to the pointer where it is stored (e.g., `data_unset* yy1`)) and tracking every type-cast operation along the path. In the case of this example, however, such an analysis will only report that the *partial* object type, `data_unset`, instead of the *full* type, `data_array`. Note that while in this sample code snippet, it is feasible to perform an interprocedural data flow analysis to identify such type-casting and derive the true types of heap objects, in real, complex codebases the heap allocation site and the type-casting statement are often separated by thousands of lines of code across multiple functions, thus making a simple static data flow analysis ineffective.

Heap Allocation Wrappers Many modern server applications use heap allocation wrappers to perform custom checks and record-keeping of each heap allocation. Because multiple heap allocations are often performed by the same heap allocation wrapper, it is no longer possible to *statically* assign a single type to the heap object created by this pattern without employing expensive context-sensitive analysis. Moreover, in many cases, such heap allocation wrappers are invoked via function pointers. To derive the type information of heap objects allocated via indirect function calls to such heap allocation wrappers, the analysis must first resolve these function pointers. These challenges further complicate the accurate recovery of type information for heap objects.

Impact on Precision Improvements Recent works [13], [18], [37], [54] have proposed using the argument types at an indirect call site to *filter* potential function targets reported

```
1  typedef struct data_unset {
2   buffer *key;
3   data_type_type type;
4  } data_unset;
5  typedef struct {
6   data_unset common;
7   array *value;
8  } data_array;
9  data_unset* initializeServer(int sz) {
10  ...
11  data_unset* yy1 = calloc(1, sz);
12  return yy1;
13 }
14 void main(int) {
15 data_array* darr = (data_array*)
16    initializeServer(sizeof(data_array));
17 }
```

Figure 1: Type-punning in the code of Lighttpd to implement inheritance and polymorphism.

by the static pointer analysis. Type ambiguity arising from type punning and heap allocation wrappers complicate the use of argument types to improve the precision of the call graph construction. With type punning, the same object can be treated as multiple different types, depending on the emulated inheritance pattern. Therefore, a naive precision improvement technique that does not accurately derive these emulated type hierarchies and simply removes functions with mismatched argument types would therefore result in unsoundness. Similarly, when heap allocation wrappers are used, the same `malloc` program statement can allocate objects of different types and the argument type matching mechanism must be aware of these different types allocated by the same program statement.

To avoid these pitfalls of using statically derived type information, PhaseSeed restricts the use of type information *only* to objects for which it has derived complete type information through dynamic execution.

3. Motivating Example

Figure 2 shows the control flow graph of a simple server with a configuration that enables `gzip` compression. At startup, the server first checks if the `gzip` option is enabled or not, and initializes `srv->init` to `gzip_init` or `default_init` using the `setup` function. The subsequent `start_processing` annotation indicates the beginning of the processing phase. In the processing phase, the server makes two indirect calls through `srv->init` and `srv->cb`, before invoking the `setup` function again to initialize the `req->init` function pointer to `req_handler`.

A fully static context-insensitive pointer analysis technique would result in the addition of all three callback functions `request_handler`, `gzip_init`, and `default_init` to the points-to sets for both `srv->init` and `request->init`, because the same function `setup` is used to initialize all three callbacks. Moreover, even if the initial configuration disables the `gzip` option, static analysis would still include the `gzip_handler` function to the points-to sets of pointers `srv->init` and `req->init`

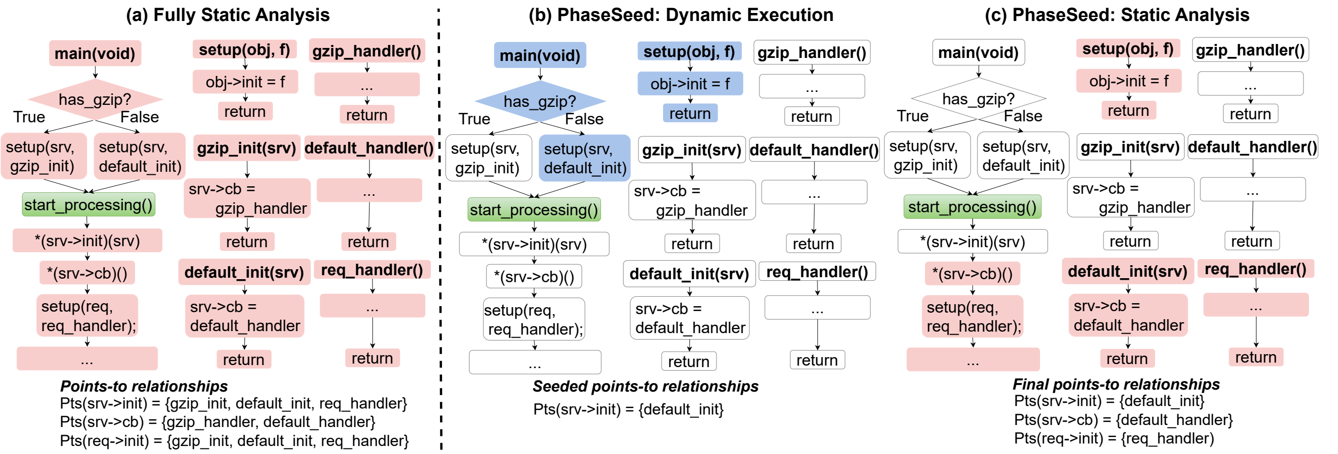


Figure 2: A fully static analysis approach compared to PhaseSeed. The `start_processing` annotation indicates the transition between the initialization and processing phase for PhaseSeed. Colored boxes show the statements operated on by any given stage. The fully static approach derives multiple spurious points-to relationships due to imprecision, while PhaseSeed's dynamic seeded approach can precisely derive the points-to relationships for all pointers in the sample code.

because it does not have access to the runtime information necessary to determine that the `has_gzip` check failed and that only the *false* branch of the `if` statement is executed.

Dynamically executing the initialization phase and seeding the resulting points-to relationships into the static analysis phase mitigates the imprecision caused due to multiple sources. These sources fall under the three major categories which we discuss below.

3.1. Unreachable Initialization Code

Server applications typically contain multiple configuration options that can enable or disable various features [38]. Dynamically executing the initialization phase allows PhaseSeed to exclude any initialization phase code that is unreachable under the given runtime configuration. If this initialization code sets up callbacks via function pointers, then excluding this code allows us to exclude the function targets that are unreachable under the given runtime configuration, thereby improving the precision of the call graph.

For example, in the case of Figure 2, if the initial configuration disables *gzip*, then the `true` branch of the `has_gzip` check will not be executed, and the dynamic execution will resolve `srv->init` to point only to `default_init`. Since `gzip_init` is not accessed from the processing phase *at all*, PhaseSeed's static analysis will also skip analyzing it. Note that we exclude a function from the static analysis only if we can guarantee that it will not be accessed from the processing phase. To achieve this, we use a conservative and sound analysis, discussed in detail in Section 4.3.

On the other hand, if a function is accessed from the initialization phase and is also deemed accessible from the processing phase, then we reanalyze it during static analysis. For example, in Figure 2, the function `setup` is invoked from both the initialization and the processing phase. Therefore, PhaseSeed's static analysis phase will reanalyze it to ensure soundness.

3.2. Fully Sensitive Dynamic Execution

As discussed in Section 2.1, performing full context, flow, and path-sensitive analysis statically poses a huge analysis cost. Dynamically executing the initialization phase allows PhaseSeed to perform a full context, flow, and path-sensitive analysis for the initialization phase code, even if the subsequent static analysis is context, flow, and path insensitive. Because typical servers are initialized in a matter of seconds, PhaseSeed gains this precision improvement with *minimal* analysis time increase.

Specifically, dynamic execution allows PhaseSeed to assign unique calling contexts for all dynamically executed functions invoked from multiple call sites. The precision benefits of this calling context separation is significant if the called function manipulates function pointers or allocates new objects. For example, the function `setup` is invoked from three different call sites with three different arguments. Under a context-insensitive static pointer analysis algorithm, the points-to sets of `srv->init` and `req->init` will be merged, resulting in the loss of precision. On the other hand, dynamic execution allows PhaseSeed to maintain full context sensitivity for the initialization phase code. We describe the details of our approach in Section 4.4.

3.3. Compounding Precision Improvement

The precision improvements facilitated by the previous two categories often have a compounding effect during the final static analysis which improves precision further. This is especially true for function pointers, as any imprecision in the call graph can potentially propagate to the *arguments* of these functions. For example, when PhaseSeed seeds the

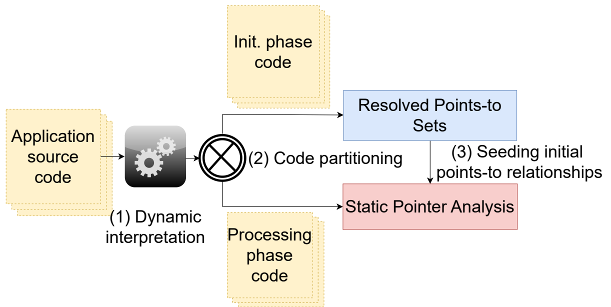


Figure 3: PhaseSeed pipeline stages.

initial points-to results, it can correctly identify that the `srv->init` function pointer can only point to the function `default_init`. This allows the subsequent static analysis phase to determine correctly that the indirect function call `*(srv->init)()` can only invoke the `default_init` function. Thus, the function `gzip_init`, which initializes the pointer `srv->cb` to `gzip_handler`, will be completely excluded from the analysis. Therefore, only `default_handler` will be added to the points-to set of `srv->cb`, when the function `default_init` is statically analyzed. This compounding precision improvement effect is significant in real-world, large, and complex codebases.

4. Design

PhaseSeed provides a custom source code annotation named `start_processing`, which developers must use to specify the beginning of the processing phase. The start of the processing phase is typically easily identifiable for split-phase applications, and is usually denoted by a blocking network system call such as `accept`, or an event loop that uses `epoll_wait`. Automated techniques [16] can further simplify the identification of this transition point.

Provided with this developer input, PhaseSeed begins to dynamically execute the server application from the initialization phase until it reaches the annotated code point. Then, it collects the dynamically observed points-to relationships and switches over to pure static analysis for the remaining code of the processing phase. Figure 3 provides an overview of the stages in the PhaseSeed pipeline.

4.1. Dynamic Interpretation

The first challenge in seeding dynamic execution results into static pointer analysis is the semantic gap between dynamic execution of binary code and static analysis, which is typically performed on source code. Critical information such as `struct` object boundaries that are essential for the precision of static analysis are lost when the source code is compiled to the binary format. Furthermore, due to various compiler optimizations, it is challenging to precisely map the assembly instructions in application binaries to source code statements. To bridge this semantic gap, we perform both the dynamic execution and the static pointer analysis

```c
static void *md2_ctx_alloc( int sz)
{
    void *ctx = calloc( sz );
    return( ctx );
}
```

Figure 4: MbedTLS example illustrating difficulty in statically determining heap object types.

on an intermediate representation of the C source code. The dynamic execution is essentially an *interpretation* of this intermediate representation, thus allowing PhaseSeed to gain fine-grained control of this execution.

PhaseSeed's dynamic interpreter begins by loading the program's intermediate representation into its memory. Then, starting from the application's `main` function, it executes each program instruction, one by one, till it reaches the `start_processing` annotation. Then, it inspects the application's in-memory state and collects the pointer relationships between the different objects. These dynamically established pointer relationships are seeded to the final static analysis phase. During interpretation, the interpreter maintains metadata for all application variables, including heap objects, created and accessed during the initialization phase, including ❶ a mapping between the variable's name and its memory address, ❷ a reverse mapping that maps a memory address to the variable name, and ❸ a mapping between the variable's name and its type. PhaseSeed uses this metadata to traverse the in-memory application objects to collect the dynamic points-to relationships.

4.2. Accurate Type Derivation

Collecting the pointer targets of complex in-memory `struct` and array objects, requires the knowledge of the offsets of the pointers embedded in them. This, in turn, requires the correct type information of these objects. Identifying the type of heap objects is challenging, especially if the heap allocation function provides no type information. For example, in the case of the simplified code snippet from the MbedTLS library shown in Figure 4, the newly allocated object is stored into a generic `void*` pointer and the `sizeof` operand is an opaque integer containing no type information. Furthermore, the heap allocator function `md2_ctx_alloc` could potentially be invoked via an indirect function call, making it even more challenging to track the object size and type passed.

To derive the full types of each object initialized during the initialization phase, PhaseSeed tracks the type-cast operations applied on each object during interpretation. Our system hooks each call to known Libc memory allocation functions, such as `malloc`, `calloc`, and `mmap`, to record both the address and the size of the allocated object. Then, when a type-cast operation operating on a heap object is interpreted, PhaseSeed updates the heap object type according to the cast type.

Type Punning and Type Upcasts. In some cases, naively updating the type of the heap object for *every* type-cast

```
1  typedef struct data_unset {
2     buffer *key;
3     data_type_t type;
4     ...
5  } data_unset;
6
7  typedef struct {
8     data_unset common;
9     buffer *value;
10  } data_string;
11
12 void array_insert_value(array *hdrs) {
13   data_unset* du = malloc(...);
14   ...
15   data_string *ds = (data_string*) du;
16   array* hdrs;
17   ...
18  array_replace(hdrs, (data_unset *)ds);
19 }
```

Figure 5: Lighttpd example illustrating a cast *up* from a more expressive type to a less expressive type.

```
1 struct plugin* plugins =
2         (struct plugin*) malloc(n*sizeof(struct plugin));
3 ...
5 for (i = 0; i < N; i++) {
6    struct plugin* p = plugins + i;
7    ...
8 }
```

Figure 6: Heap array accesses using pointer arithmetic.

operation can result in the loss of type information. This is particularly true for applications that use type punning, as shown in Figure 5. The `data_string` type *inherits* from the `data_unset` type using type punning. In Line 15, the heap object pointed to by the pointer `du`, is first *downcasted* from `data_unset` to the child type `data_string`. Then in Line 18, the object is *upcasted* to the parent type `data_unset`. Thus, naively updating the type of each heap object with the target type for each type-cast operation would result in the association of the type `data_unset` with the heap object, instead of type `data_string`, resulting in the loss of the complete type information for the object.

PhaseSeed uses the notion of *type descriptiveness* to handle this potential loss of type information. We define type descriptiveness as the *total* number of fields contained in a `struct` type after all nested `struct` type fields are expanded. Thus, in the Figure 5 code snippet, the type `data_string` is more descriptive than `data_unset`, as it describes the field `value` in addition to all of the fields of `data_unset`. During interpretation, PhaseSeed updates the type of a heap object *only if* the object is type-casted to a *more* descriptive type. Thus, when the function `array_replace_value` casts an object of type `data_string` back to `data_unset`, which is less descriptive, thus PhaseSeed ignores this type-cast operation, and retains the type `data_string` for the object, thus maintaining correctness.

Heap Arrays. The equivalence of pointer arithmetic and array indexing in C/C++ further complicates accurate type identification. In C/C++, a pointer can be initialized to the first element of an array and each subsequent element can be accessed by using increment operations performed on that pointer. In the case of heap arrays, the array can thus be type-casted to the type of the *first* element of the array, instead of being type-casted to the array type. Thus, simply extracting the type information from the type-cast operation is not enough to correctly derive the complete heap array type. Specifically, the array size information is lost. The simplified code snippet in Figure 6 shows how Lighttpd allocates a variably-sized array of plugins on the heap and accesses each array element via the pointer `plugins` that is initialized to the first plugin.

To accurately derive the size of heap arrays, the interpreter first records the size of each heap object at its allocation site. Then, when the type-cast operation is interpreted, PhaseSeed *derives* the number of elements in the array by comparing the size of the allocated heap object with the size of pointer's *base* type. For example, in the case of Figure 6, the interpreter divides the size of the heap object, allocated in Line 2 by `sizeof(struct plugin)` to determine the array size, thereby deriving the complete array type.

4.3. Code Partitioning

When the interpreter encounters PhaseSeed's `start_processing` annotation that marks the transition point, the application code is partitioned into functions accessible from the initialization and the processing phase. Performing this partitioning correctly is critical for the soundness of the analysis. To achieve sound partitioning, PhaseSeed uses an iterative process that relies on the following criteria for identifying the functions accessible from the processing phase.

Direct Function Calls. PhaseSeed maintains a list F of functions accessible from the processing phase. It starts populating this list by collecting the functions accessible via direct function calls from the transition point. To achieve this, PhaseSeed traverses all basic blocks that are reachable from the transition point through direct function calls and adds the targets of all encountered direct function calls to the function list F.

Initialization Phase Address-Taken Functions. The initialization phase might potentially store the addresses of certain functions to function pointers. To handle such address-taken functions PhaseSeed scans the application's global, stack, and heap objects for any values that correspond to function addresses and adds them to the accessible list of functions F. These are the functions whose addresses are taken and stored during the initialization phase and can be invoked via indirect call sites during the processing phase. Function pointers contained in global variables require special handling. We consider a global variable (and any functions it refers to) to be *accessible* only if it either (a) is directly accessed from an accessible function, or (b) another accessible stack, heap, or global variable holds a reference to it. In other words, if the global variable cannot be accessed *at all* from the processing phase, we exclude any function pointers it contains from the processing phase code.

Processing Phase Address-Taken Functions. Finally, the processing phase might store and update function pointers. To handle these cases, PhaseSeed scans the program statements of all functions in the accessible list F, for any instructions that store the address of any *new* function to a function pointer. All such functions are added to the list F.

Putting these strategies together, we develop an iterative algorithm that recursively collects all functions that might be accessed from the processing phase till a *fixed point* is reached and no new accessible function is discovered. This ensures that our algorithm can soundly compute all functions that remain in scope at the end of the initialization phase, with respect to the given runtime configuration. The complete algorithm for this process is described in Appendix B.

4.4. Seeding Initial Points-to Relationships

After identifying which functions can be invoked from the processing phase, PhaseSeed proceeds to build the points-to graph of all pointers and objects initialized during the initialization phase and *seeds* the static analysis phase with this graph. The goal of this step is to *pre-solve*, with full precision, the pointers and objects that are created till the transition point. Note that the points-to relationships established in this phase are not final—the subsequent static analysis phase processes and potentially updates these relationships.

To build the initialization phase points-to graph, PhaseSeed iterates over every pointer in application memory and reads the value stored in its memory. Then, using the interpreter's internal metadata maps and collected type information, it resolves the points-to relationships by comparing the pointer values to the addresses of in-memory objects. These dynamic points-to relationships are then seeded to the final static pointer analysis stage.

Context Sensitive Heap Seeding. Seeding the dynamic points-to relationships to the static analysis phase requires mapping the in-memory addresses to IR program statements that define or create objects. This process is straightforward in the case of global variables and stack objects as the object address has a 1-to-1 mapping with a program statement that defines the corresponding object. But in the case of heap objects, the same program statement (e.g., a call to the `malloc` function) can be invoked multiple times from different *calling contexts*, resulting in the creation of multiple heap objects. Each of these objects can have their own unique points-to relationships. Mapping *all* of these *unique* objects to the same heap allocation program statement would prevent the static analysis from distinguishing between these unique heap objects, resulting in loss of precision.

PhaseSeed uses *cloning* to preserve the precision of the heap objects created during the initialization phase. During interpretation, the interpreter maintains the mapping between each memory allocation program statement (`malloc, mmap, calloc`, etc.) and the heap objects it allocates at runtime. For each heap-allocated object at a given call site, PhaseSeed *clones* the memory allocation program statement, and ensures that a unique heap allocation statement corresponds to each heap object created during dynamic execution.

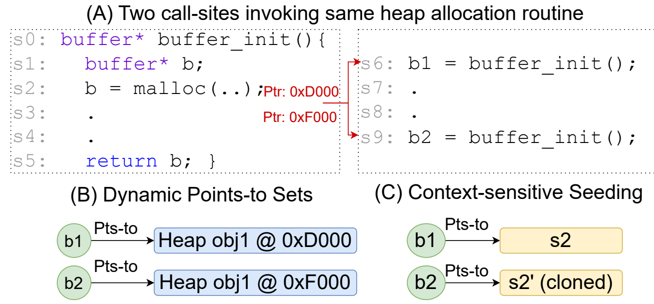


Figure 7: Context-sensitive seeding of heap objects' pointer relationships.

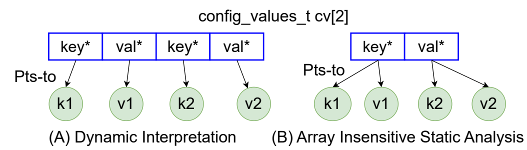


Figure 8: Array-index insensitive pointer relationship seeding.

Figure 7 shows a simplified illustration from the Lighttpd codebase, where the `malloc` invocation in statement `s2` returns two different heap objects with addresses `0xD000` and `0xF000`, to the statements `s6` and `s9`, respectively. Cloning the statement `s2` allows PhaseSeed to separate the calling contexts and precisely seed the points-to relationships for pointers `b1` and `b2`.

Array Element Seeding. The static pointer analysis framework used by PhaseSeed is array-index insensitive, and does not distinguish between different elements of an array. However, the points-to relationships captured by the interpretation phase distinguish between every array element. Therefore, when seeding the pointer relationships of array elements, we must bridge this gap between the array-index sensitive dynamic interpretation and array-index insensitive static pointer analysis.

We bridge this gap by *collapsing* all array elements during the seeding phase. Consider the simplified code snippet from Lighttpd in Figure 8. The object `cv` is an array of type `config_values_t`. This array is initialized during server startup, and therefore at the end of the initialization phase we collapse the corresponding fields of each array element into a single field as shown. This results in a loss of precision but maintains soundness, and is unavoidable without using a more expensive array-index sensitive static analysis.

Once the processing phase code is identified and the dynamic points-to results are seeded, the static analysis phase builds the final points-to sets for all the pointers in the application iteratively. PhaseSeed uses a standard field-sensitive, context and array-index insensitive Andersen's-style pointer analysis [17], [72].

4.5. Alternate Runtime Configurations Options

Given the substantial increase in the use of containers for deploying applications [2], obtaining commonly utilized runtime settings has become feasible. Furthermore, due to the increasing complexity of the runtime settings of modern server applications [81], developers often rely on the pre-configured server applications that ship with popular Docker images [1], [7], [8] and launch their web applications on top of fully configured web servers such as Nginx. PhaseSeed can harden these web servers using these fixed runtime configuration options.

To support multiple runtime configurations, we extend PhaseSeed with a lightweight runtime mechanism that allows the binary to adapt its CFI instrumentation based on the runtime configuration options with which it is launched. The binary is shipped with the interpreter and, on launch, checks whether it has already been executed under the current configuration. If not, it invokes the interpreter to execute the initialization phase, perform pointer analysis, and generate the final CFI results. These results are written to a `cfi.map` file and used to patch the binary before transferring control. After update, the `cfi.map` is changed to read-only. For previously seen configurations, the corresponding `cfi.map` is loaded at launch and the binary is patched directly. This approach enables PhaseSeed to apply its precision benefits across diverse runtime environments without sacrificing soundness.

5. Implementation

We developed PhaseSeed using the LLVM 12 [10] toolchain and the popular SVF [72] static analysis library. First, we generate the Intermediate Representation bitcode for the application using Link Time Optimization (LTO). The LLVM toolchain provides the `lli` interpreter that can interpret LLVM Intermediate Representation, which we use for dynamic interpretation. We enhance the interpreter to record the address and type metadata for the heap objects, by instrumenting `lli`'s handling of the LLVM `BitCastInst` and `CastInst` instructions. While `lli` supports most common LLVM IR instructions, we had to add support for the `atomicrmw` and `cmpxchg` instructions to fully support the applications in our test suite. We use the default field-sensitive and context-insensitive Andersen's style pointer analysis algorithm from the popular SVF [72] library for static pointer analysis. Additional details including the handling of multithreaded applications is provided in Appendix C.

6. Use Cases

We apply PhaseSeed to three use cases—CFI, software debloating, and system call filtering. In each of these cases, we use PhaseSeed to resolve the indirect call sites and generate the call graph for the application's processing phase. Then, using this call graph, we either implement the CFI checks, remove unneeded functions, or filter the unneeded system calls, depending on the use case.

Control Flow Integrity. We implement a forward-edge CFI mechanism based on the call graph generated by PhaseSeed. This call graph provides the list of permitted targets for all indirect call sites invoked during the processing phase. Using these lists of permitted targets, we instrument each indirect call site to check if the target observed at runtime is permitted or not.

Special handling is required if the same indirect call site is invoked from both the initialization and the processing phase with different permitted targets. In these cases, we clone and specialize the functions containing such indirect call sites into two versions, one for each phase respectively. Then, we apply the corresponding CFI filters for each clone. This allows us to enforce different CFI profiles for both the specialized versions correctly.

Software Debloating. Software debloating aims to reduce an application's attack surface by identifying and removing unneeded application code. Generating a sound and precise call graph for the application and its libraries is essential for performing software debloating statically. An unsound call graph can break the program, and an imprecise call graph limits the security guarantees. Therefore, we use PhaseSeed to generate a more precise and sound call graph.

Previous works [14], [67] that debloat code statically only remove code from the application's dependent libraries. These techniques consider the entire application a monolithic entity and assume *all* its code is required, and only specialize the dependent libraries according to the application's requirements. However, by using the processing phase call graph generated by PhaseSeed we can also debloat application functions that are unneeded under the given initial configuration, once the application has finished initialization.

System Call Filtering. Filtering unused system calls is an effective mechanism for reducing the application's attack surface [31], [36], [37], [68] Similarly to software debloating, system call filtering also requires the precise call graph of the application and its libraries to identify the unreachable system calls. Temporal Specialization [37] is a technique that uses static analysis to perform system call filtering. Temporal Specialization also focuses on two-phase applications, and uses static analysis to partition the application code into functions accessible from the initialization and processing phases. Therefore, this technique's security guarantees depend on the precision of the initialization and processing phase call graphs.

To improve precision, Temporal Specialization augments pointer analysis and call graph construction with argument type matching, i.e., it removes all targets in the points-to set of an indirect call site whose formal argument types do not match the actual call site argument types. However, as discussed in Section 2.2, the pointer types derived statically are often unreliable. Therefore, performing this type-based filtering technique can potentially lead to unsoundness.

TABLE 1: Applications used for evaluation and whether they use type-punning and `void*` pointers. (✓: uses pattern, ✗: does not use pattern)

Application	Transition Point	Type-punning	`void*` pointers
Nginx	`ngx_worker_process_cycle`	✗	✓
Lighttpd	`server_main_loop`	✓	✓
MbedTLS	`mbedtls_net_accept`	✗	✓
Monkey	`mk_server_loop`	✓	✓
Ssh-agent	`prepare_poll`	✓	✓
Memcached	`event_base_loop`	✓	✓
Wget	Basic block in `main`	✗	✓
Curl	Basic block in `operate`	✗	✓

We replaced the static call graph generation component of Temporal Specialization with PhaseSeed. The rest of the toolchain proceeds unmodified and extracts the system call profiles for each stage of the application. Moreover, as discussed in Section 4.2, because PhaseSeed only performs conservative and sound type-based optimizations, the generated call graph is guaranteed to be sound.

7. Experimental Evaluation

We evaluate our system with eight popular server and desktop split-phase applications. Across all of these applications, we verified that the points-to relationships established at the end of the initialization phase indeed remained constant across multiple runs. We also verified that the applications' functionality was not affected after hardening them using PhaseSeed's points-to results, demonstrating the soundness of our approach.

7.1. Applications

Table 1 presents the applications we used to evaluate our system, along with the transition function that marks the start of the processing phase. Nginx [6], Lighttpd [9], and Monkey [64] are web servers supporting pluggable modules. Of these, Monkey is multi-threaded while Nginx and Lighttpd use asynchronous I/O using `epoll` for concurrency. Memcached [11] is an event-driven key-value store that uses LibEvent [55]. MbedTLS [56] is an SSL/TLS library, and we use the sample `ssl_server2` application for our evaluation. Wget [5] and Curl [4] are command-line utilities for downloading web content. Both programs provide a variety of command-line options that determine how the URL is parsed and handled. Each of these applications are executed with their default initial configuration. Four out of the eight applications in our evaluation set use type-punning, while all of them use `void*` pointers to point to complex `struct` type objects. Therefore, using a type-based precision improvement technique would potentially lead to unsoundness for these applications.

7.2. Pointer Analysis Statistics

Pointer analysis imprecision results in the inclusion of spurious objects in the points-to sets, i.e., the analysis derives pointer relationships that are not possible during actual runtime execution. Therefore, the size of the points-to sets is an indication of the level of imprecision of the pointer analysis. Table 2 shows the average and maximum points-to set size for PhaseSeed and for the standard field-sensitive and context-insensitive pointer analysis provided by the popular SVF framework [72]. PhaseSeed significantly reduces both the average and maximum points-to set sizes. For all applications except Curl and MbedTLS, PhaseSeed shows a significant precision improvement. We discuss the reasons for this behavior in Section 7.3. Appendix F evaluates the analysis time for the baseline fully static pointer analysis and PhaseSeed. Across all applications, PhaseSeed takes less time than the baseline fully static pointer analysis.

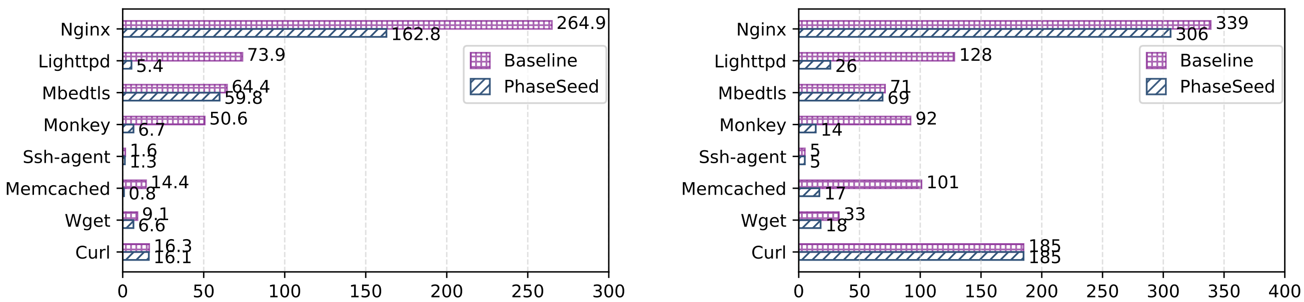
TABLE 2: Points-to set sizes for whole program static pointer analysis using SVF vs. PhaseSeed.

Application	Max Pts-to Set Size SVF	Max Pts-to Set Size PhaseSeed	Avg Pts-to Set Size SVF	Avg Pts-to Set Size PhaseSeed
Nginx	1837	1465	372.46	316.24
Lighttpd	1433	911	70.14	11.86
Mbedtls	244	244	31.21	28.63
Monkey	795	247	198.97	48.11
Ssh-agent	182	105	1.46	1.26
Memcached	757	298	38.43	16.24
Wget	489	460	17.19	14.87
Curl	2059	1782	212.98	147.50

7.3. Control Flow Integrity

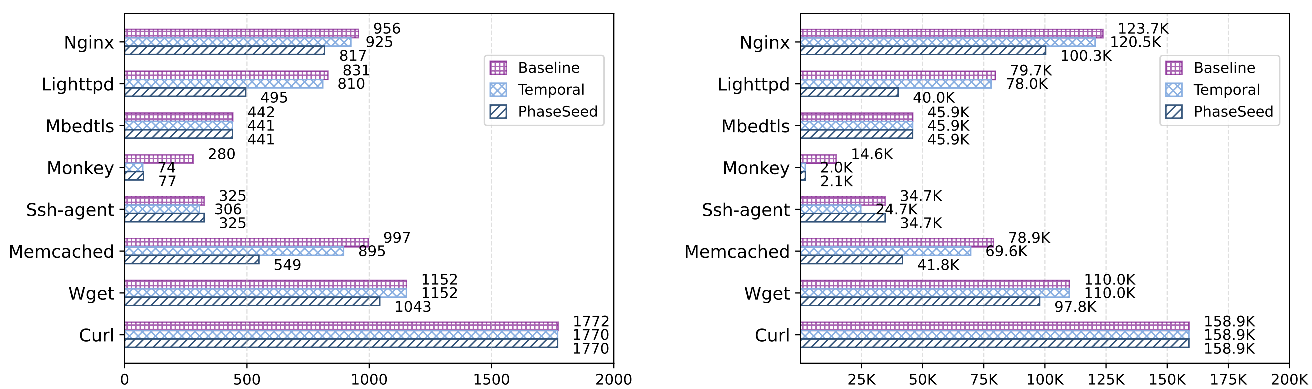
Figure 9 reports the maximum and average number of targets for each indirect call site using the naive SVF-based approach and our technique. The targets for each indirect call site are considered an *Equivalence Class (EC)*. All server applications except Curl and MbedTLS, show an improvement in the precision of the resolved targets ranging from 43% to 92.6%. The web servers Nginx and Lighttpd, support configurable *modules* which encapsulate custom functionalities that can be enabled by the initialization configuration. These modules are created and stored on the heap during the initialization phase, and thus they benefit from the fully precise dynamic analysis and context-sensitive seeding of heap objects, as discussed in Section 4.4. These are examples that clearly show the benefit of dynamic seeded pointer analysis. Note that in the case of Memcached, the average EC size is less than 1 because some indirect call sites in the Libevent library have no targets because those Libevent features are not used by Memcached.

In the case of MbedTLS, however, the precision improvement is only marginal. This is because, in the case of MbedTLS, the initialization phase simply loads and configures the private keys and does not set up any function pointers. The main use of function pointers in the MbedTLS codebase occurs in the encryption/decryption routines of the different protocols. These protocols however are stored in different global *protocol objects* and are invoked depending on the type of client request received, which is resolved during the processing phase. Therefore, all global protocol



(a) Average EC Size (b) Maximum EC Size

Figure 9: Average and Maximum EC (Equivalence Class) size for CFI.



(a) Program Function Count (b) Program IR Instruction Count

Figure 10: Number of functions and IR instructions debloated.

objects remain *in scope* during the processing phase, and PhaseSeed is forced to conclude that all encryption/decryption protocol functions are valid at every indirect call site. Curl also performs limited operations during initialization, mainly parsing configuration settings without allocating any objects that persist into the processing phase. Therefore, PhaseSeed provides lower precision improvements for this application. Appendix D provides an ablation study to determine the precision improvements provided by each PhaseSeed component.

7.4. Software Debloating

Similar to prior works on software debloating, we report the amount of code that remains accessible to assess PhaseSeed's attack surface reduction benefit. Since previous works [14], [67] on code debloating through static analysis do not improve the precision of the application's call graph, we cannot use them to perform a proper comparison. Instead, we use the call graphs generated by Temporal Specialization [37] as our point of comparison. Figure 10 compares the number of accessible functions and LLVM IR instructions for the 8 applications in our dataset using the call graph generated by the baseline SVF, Temporal Specialization, and PhaseSeed.

PhaseSeed provides a significant code reduction benefit for Lighttpd, Memcached, and the Monkey server, reducing the number of accessible functions from 947 to 573 for Lighttpd, 1000 to 549 for Memcached, and 355 to 83 for the Monkey server. The number of accessible IR instructions after applying PhaseSeed is reduced by 48%, 47%, and 86% for Lighttpd, Memcached, and the Monkey server, respectively. PhaseSeed also reduces the accessible functions and instructions for Nginx and Wget. The code reduction for MbedTLS, Ssh-agent, and Curl is limited, which occurs for the same reasons described in Section 7.3.

7.5. System Call Filtering

We evaluated the number of system calls that can be filtered by PhaseSeed compared with the original Temporal Specialization [37] toolchain. As shown in Table 3, PhaseSeed filters additional system calls for Nginx and Lighttpd compared to Temporal Specialization [37], offering a considerable attack surface reduction. These include system calls, such as `bind` and `select` for Lighttpd, which have been proven to be security-critical by previous work [37], as discussed in Appendix E. Note that Lighttpd supports multiple mechanisms for processing incoming requests,

TABLE 3: Additional system calls filtered by PhaseSeed compared to Temporal Specialization.

Application	Additional System Calls Filtered
Nginx	`sched_yield`, `pwritev`, `rename`, `utimes`
Lighttpd	`bind`, `listen`, `select`, `pipe2`, `poll`

including `epoll`, `select`, and `poll`. The default initial configuration for Lighttpd enables only `epoll`, which allows PhaseSeed to filter the `select` and `poll` system calls. For the other applications, PhaseSeed removes the same number of system calls as Temporal Specialization. Moreover, PhaseSeed's call graph analysis avoids the soundness issues of Temporal Specialization's type-based matching.

8. Discussion

Dynamic vs. Static Boundary. PhaseSeed assumes that the transition point splits the program into two phases. While this separation is adequate to illustrate the benefits of PhaseSeed, further precision improvements can potentially be obtained by the addition of deeper phases. This is especially true for applications such as Curl, which continue parsing configuration options past PhaseSeed's selected transition point. In the case of such applications, moving the transition point deeper into the application code would improve the precision further. Placing the transition point beyond the start of the serving phase would cause unsoundness, but PhaseSeed can detect this because in that case, the developer-placed transition point would not be reached during the dynamic interpretation as the application waits for a user request. Thus, PhaseSeed will flag this error and protect against potential unsoundness.

Other Use Cases. PhaseSeed is a generic technique that improves pointer analysis precision and can be applied to any use case that requires pointer analysis. DSR [19] and DynPTA [59] use static pointer analysis to obtain the superset of all pointers that might point to sensitive application data to randomize or encrypt it. DPP [15] uses pointer analysis to identify security-critical application data. Similarly, fuzzing techniques [24], [25] use pointer analysis to identify and prioritize interesting mutations. PhaseSeed can be used to improve the precision of these mechanisms.

9. Related Work

Pointer Analysis Precision Improvement. Traditional techniques for improving precision require adding *more* program information to the pointer analysis. Previous works [77], [78] improve context sensitivity using cloning-based and summarization-based approaches. Hardekopf and Lin [42] proposed using Binary Decision Diagrams (BDDs) to implement flow and context-sensitivity. Pearce et al. [61] presented a field-sensitive pointer analysis. Lei et al. [48] improved the precision field derivations by using object type information. Hasti et al. [43] iteratively converted memory to SSA form to gradually add flow-sensitivity to their points-to results. Li et al. [50] propose a hybrid approach that combines data flow analysis and type-based analysis to identify writes to global variables in the Linux kernel. Integrating PhaseSeed with these techniques can further improve their precision. Recent works [13], [21], [53], [83] leverage a type-based analysis to generate call graphs, including for large codebases such as the Linux kernel [83]. These approaches suffer from the limitations discussed in Section 2.2. Kallgraph [49] diagnoses soundness and completeness gaps in type-based call graph construction and remedies them by combining type-based analysis with demand-driven pointer analysis. PhaseSeed instead provides a general refinement framework for pointer analysis, of which call graph generation is just one application.

Past-Sensitive pointer analysis [74] integrates pointer analysis with symbolic execution to improve its precision. Hybrid pruning [29] applies dynamic profiling to derive pointer relationships, but does not provide any soundness guarantees. Optimistic Hybrid Analysis [32] uses predicated static analysis to accelerate dynamic analysis. Smaragdakis et al. [39] use dynamic heap snapshots to improve the soundness of static analysis to account for dynamic loading and cross-language libraries. In contrast with these techniques, PhaseSeed presents a sound technique for seeding the results of dynamic execution to improve pointer analysis precision.

Control Flow Integrity. CFI was originally introduced by Abadi et al. [12] to protect against control flow hijacking attacks. Since then, many works have discussed its shortcomings [23], [27], [34] and proposed methods to improve its precision and effectiveness [26], [28], [33], [41], [45], [47], [58], [75], [76]. More recent works [33], [41], [45], [75] leverage hardware features (e.g., Intel PT) to obtain runtime information and reduce the valid targets for each indirect call site. Such techniques are orthogonal to PhaseSeed. TypeArmor [76] uses type-based matching, and OSCFI [47] leverages data flow analysis to reduce the number of targets of indirect call sites. However, previous work [51] shows that these techniques are unsound and result in compatibility issues making their usage impractical.

Software Debloating. Various works have focused on attack surface reduction using static analysis [30], [31], [37], [38], [57], [67], [68], dynamic analysis [20], [35], [65], or hybrid approaches [22], [36], [63], [66], [80]. Callgraph construction underpins many of the recent techniques [14], [22], [31], [46], [67]. Saffire [57] performs argument-level specialization and requires an accurate call graph to resolve the argument flows via indirect calls. Configuration-to-Code (C2C) [38] maps runtime settings to the application's code through static analysis and filters unnecessary system calls using this mapping. PhaseSeed can generate a more precise call graph improving these works' effectiveness.

10. Conclusion

We presented PhaseSeed, a novel technique for precise call graph construction for split-phase applications. We applied PhaseSeed to control flow integrity (CFI), software debloating, and system call filtering, and showed that PhaseSeed provides up to 92.6% precision improvement compared to state-of-the-art static call graph construction techniques.

References

[1] Bitnami Wordpress - Docker Hub. https://hub.docker.com/bitnami/wordpress.

[2] Cncf 2023 annual survey. https://www.cncf.io/reports/cncf-annual-survey-2023/.

[3] Control flow integrity - fsanitize-cfi - clang - llvm. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[4] Curl: command line tool and library. https://curl.se/.

[5] Gnu wget. https://www.gnu.org/software/wget/.

[6] Nginx. https://www.nginx.com/.

[7] Wordpress - Docker Hub. https://hub.docker.com/_/wordpress.

[8] Zend container registry. https://cr.zend.com/.

[9] Lighttpd. https://www.lighttpd.net/, 2020.

[10] The LLVM compiler infrastructure. https://llvm.org/, 2020.

[11] Memcached. https://memcached.org/, 2020.

[12] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, page 340–353, 2005.

[13] Muhammad Abubakar, Pedro Ahmad, Adil Fonseca, and Dongyan Xu. Shard: Fine-grained kernel specialization with context-aware hardening. In *Proceedings of the 2021 Usenix Security Symposium*, 2021.

[14] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.

[15] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N Asokan, and Daphne Yao. Not all data are created equal: Data and pointer prioritization for scalable protection against data-oriented attacks. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association.

[16] Mohannad Alhanahnah, Philipp Schubert, Thomas Reps, Somesh Jha, and Eric Bodden. slash: A technique for static configuration-logic identification. *arXiv preprint arXiv:2310.06758*, 2023.

[17] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[18] Markus Bauer, Ilya Grishchenko, and Christian Rossow. Typro: Forward cfi for c-style indirect function calls using type propagation. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 346–360, 2022.

[19] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 1–22, 2008.

[20] Priyam Biswas, Nathan Burow, and Mathias Payer. Code specialization through dynamic feature observation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, CODASPY '21, page 257–268, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Yuandao Cai, Yibo Jin, and Charles Zhang. Unleashing the power of type-based call graph construction by using regional pointer information. In *33nd USENIX Security Symposium (USENIX Security 24)*, 2024.

[22] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. pages 139–151, 2021.

[23] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[24] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. *arXiv preprint arXiv:2007.15943*, 2020.

[25] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[26] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 38–49, 2016.

[27] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963, 2015.

[28] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE symposium on security and privacy*, pages 292–307. IEEE, 2014.

[29] Dipanjan Das, Priyanka Bose, Aravind Machiry, Sebastiano Mariani, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. Hybrid pruning: Towards precise pointer and taint analysis. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2022.

[30] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. Towards automated application-specific software stacks. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, 2019.

[31] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[32] David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 348–362, 2018.

[33] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *USENIX Security Symposium*, pages 131–148, 2017.

[34] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[35] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 2019.

[36] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[37] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766, 2020.

[38] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. C2c: Fine-grained configuration-driven system call filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[39] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.

[40] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 198–208, 2018.

[41] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 173–184, 2017.

[42] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. *ACM SIGPLAN Notices*, 44(1):226–238, 2009.

[43] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. *ACM SIGPLAN Notices*, 33(5):97–105, 1998.

[44] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[45] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1470–1486, 2018.

[46] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. Protect the system call, protect (most of) the world with bastion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 528–541, 2023.

[47] Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *USENIX Security Symposium*, pages 195–211, 2019.

[48] Yuxiang Lei and Yulei Sui. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings 26*, pages 27–47. Springer, 2019.

[49] Guoren Li, Manu Sridharan, and Zhiyun Qian. Redefining indirect call analysis with kallgraph. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2734–2752. IEEE Computer Society, 2025.

[50] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4211–4228, 2023.

[51] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1821–1835, 2020.

[52] V Benjamin Livshits and Monica S Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 317–326, 2003.

[53] Kangjie Lu. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1270. IEEE, 2023.

[54] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[55] Nick Mathewson, Azat Khuzhin, and Niels Provos. libevent - an event notification library. https://libevent.org/.

[56] ARM MBED. Mbed TLS. https://tls.mbed.org/, 2020.

[57] Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization against code reuse attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[58] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[59] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937. IEEE, 2021.

[60] David J Pearce, Paul HJ Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 3–12. IEEE, 2003.

[61] David J Pearce, Paul HJ Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):4–es, 2007.

[62] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization*, pages 126–135. IEEE, 2009.

[63] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–180, 2020.

[64] Monkey Project. Monkey http server. https://github.com/monkey/monkey.

[65] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[66] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. Slimium: Debloating the chromium browser with feature subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 461–476, 2020.

[67] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, pages 869–886, 2018.

[68] Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktas, Jun Xu, and Georgios Portokalidis. Syspart: Automated temporal system call filtering for binaries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1979–1993, 2023.

[69] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[70] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices*, 41(6):387–400, 2006.

[71] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[72] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[73] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. Spas: Scalable path-sensitive pointer analysis on full-sparse ssa. In *Asian Symposium on Programming Languages and Systems*, pages 155–171. Springer, 2011.

[74] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. Past-sensitive pointer analysis for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 197–208, 2020.

[75] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940, 2015.

[76] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[77] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[78] Robert P Wilson and Monica S Lam. Efficient context-sensitive pointer analysis for C programs. *ACM Sigplan Notices*, 30(6):1–12, 1995.

[79] Robert P Wilson and Monica S Lam. Efficient context-sensitive pointer analysis for c programs. *ACM Sigplan Notices*, 30(6):1–12, 1995.

[80] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. LIGHTBLUE: Automatic Profile-Aware debloating of bluetooth stacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 339–356. USENIX Association, August 2021.

[81] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319, 2015.

[82] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th IEEE/ACM Symposium on Code Generation and Optimization (CGO)*, pages 218–229, 2010.

[83] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium*, pages 1205–1220, 2019.

Appendix A. Static Pointer Analysis Constraints

Static pointer analysis consists of two stages. In the first stage, all pointer-related operations are converted to constraints. The second stage solves these constraints according to the constraint resolution rules. This section describes the constraints and the constraint resolution rules for Andersen's style [17] pointer analysis.

Four types of pointer operations result in the creation of constraints. These are described below. Note that p and q in these examples can be single-level (`int *p`) or multi-level (`int **p`) pointers.

1) $p := \&x$ (*Address-of*)
2) $p := q$ (*Copy*)
3) $p := *q$ (*Dereference*)
4) $*p := q$ (*Assign*)

The points-to set for a pointer p is depicted by $pts(p)$. The constraint resolution rules for Andersen's inclusion-style analysis are as follows:

1) $p := \&x \Rightarrow x \in pts(p)$
2) $p := q \Rightarrow pts(p) \supseteq pts(q)$
3) $p := *q \Rightarrow pts(p) \supseteq pts(pts(q))$
4) $*p := q \Rightarrow pts(pts(p)) \supseteq pts(q)$

The pointer analysis algorithm iteratively applies these rules to update the points-to set of each pointer in the application, until the *fixed point* is reached and no new pointer relationships are observed.

Appendix B. Code Partitioning Algorithm

The complete algorithm for the code partitioning stage described in Section 4.3 is shown in Listing 1. The algorithm iteratively accumulates all functions potentially reachable from the transition point, until a fixed point is reached and no new functions are discovered.

Appendix C. Multithreaded Applications

PhaseSeed uses LLVM's `lli` interpreter to interpret the applications. The `lli` interpreter is a single-threaded application and does not support the interpretation of multithreading applications. Therefore, to support the analysis of multithreaded applications, PhaseSeed interprets *only* the main thread till the transition point and treats all child threads as part of the processing phase. In other words, *all* code in the child threads will always be statically analyzed.

The child threads are usually created using well-known interfaces such as `pthread_create` which accept a function argument that acts as the thread *entry-point*. We modify `lli`'s handling of function calls to these interfaces to record the thread entry-point specified in such calls. When partitioning the code, as described in Section 4.3, we add all such thread entry-point functions to the initial list of functions accessible from the processing phase. Therefore, this approach ensures soundness but potentially loses precision because all child threads are always completely statically analyzed. Adding multithreaded capabilities to the `lli` interpreter would mitigate this potential loss of precision. We leave this task for future work.

Appendix D. Ablation Study

In this section, we present the results of the ablation study where we study the precision improvement impact of each

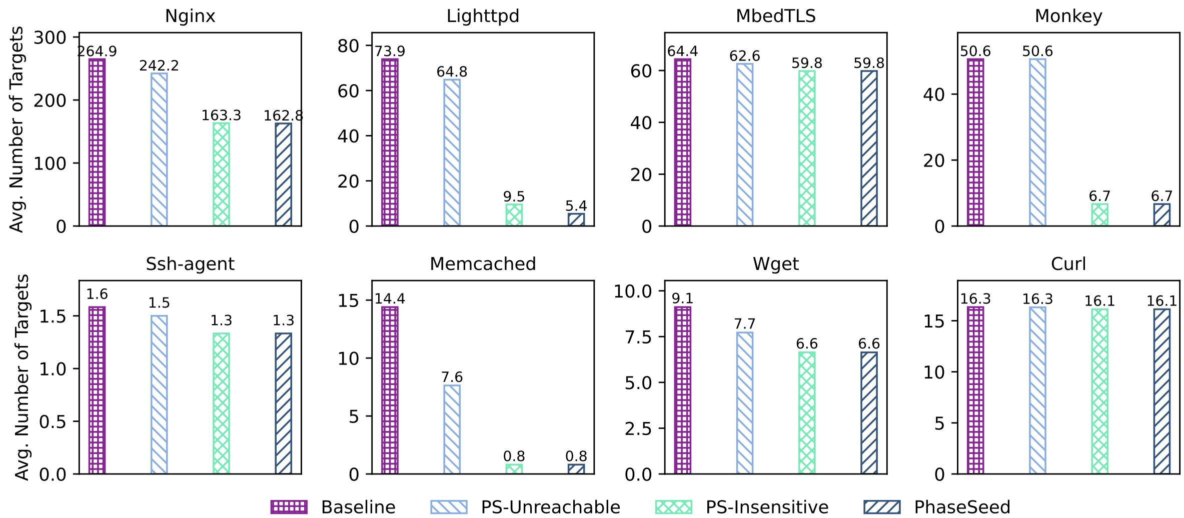


Figure 11: Ablation Study: Average number of CFI targets.

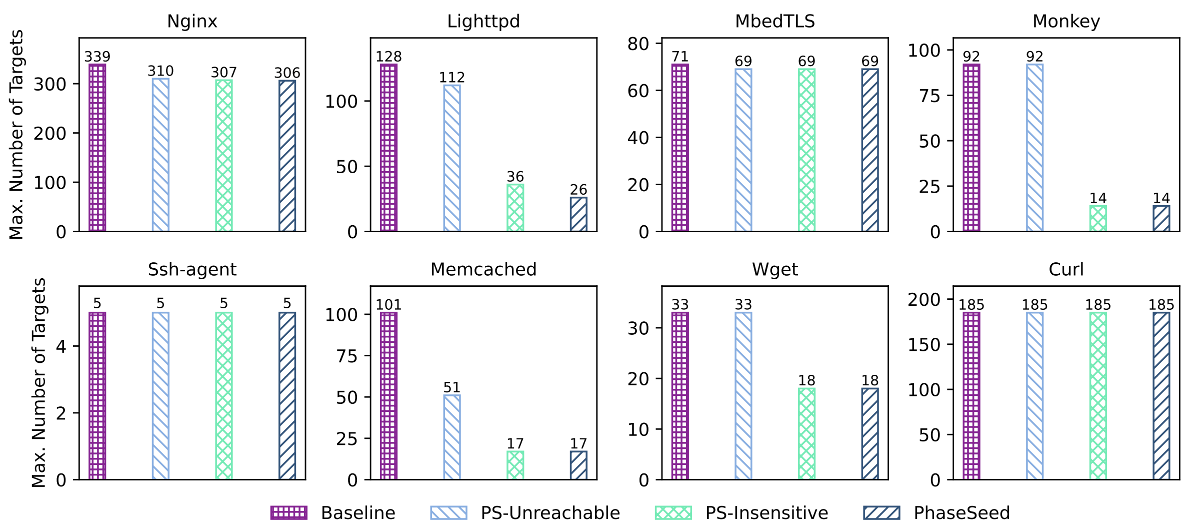


Figure 12: Ablation Study: Maximum number of CFI targets

component of PhaseSeed for the Control Flow Integrity use case. We perform the ablation study with three configurations, in addition to the baseline system, and report both the average and the maximum sizes of equivalence classes.

Configurations. The first configuration, *PS-Unreachable*, isolates the precision improvement due to the removal of the code that becomes unreachable because of the runtime configuration disabling certain code paths, as discussed in Section 3.1. The second configuration, *PS-Insensitive*, disables the fully-sensitive dynamic execution discussed in Section 3.2 and only captures the effect of the compounding precision improvement. In this configuration, the points-to heap relationships are not captured in a context-sensitive manner, at the end of the dynamic execution. Finally, the third configuration, *PhaseSeed*, enables the fully-sensitive dynamic execution and captures the total precision improvement due to all components of our system.

Effect of Unreachable Code. As shown in Figure 11 and Figure 12, in the case for all applications, except Curl, disabling unreachable code alone does not account for the total precision improvement provided by PhaseSeed. For example, in the case of Nginx, filtering the unreachable code alone reduces the average EC size from 264.9 to 242.2. But the compounding precision improvement and fully-sensitive dynamic execution reduces the average number of targets to 162.8.

Algorithm 1: Algorithm: Find all functions accessible from the processing phase.

```
Input  : HeapObjects = In scope heap objects at
         the transition point
         StackObjects = In scope stack objects at
         the transition point
         GlobalVariables = All global variables in
         the application
Result: F = set of functions accessible from the
        processing phase.
F = ∅ ;
GlobalToFnMap = ∅ ;
AccessibleGVs = ∅ ;
foreach H_obj ∈ HeapObjects do
    Inspect H_obj to find all contained function
      references F_h;
    F = F ∪ F_h;
    Inspect H_obj to find all global variable
      references G_h;
    AccessibleGVs = AccessibleGVs ∪ G_h;
end
foreach Stk_obj ∈ StackObjects do
    Inspect Stk_obj to find all function references
      F_stk;
    F = F ∪ F_stk;
    Inspect Stk_obj to find all global variable
      references G_stk;
    AccessibleGVs = AccessibleGVs ∪ G_stk;
end
foreach GVar in AccessibleGVs do
    Inspect GVar to find all function references F_g;
    F = F ∪ F_g;
end
foreach G ∈ GlobalVariables do
    Inspect G to find all function references F_g;
    GlobalToFnMap[G] = GlobalToFnMap[G]
      ∪ F_g;
end
F_new = F;
do
    F = F ∪ F_new;
    F_new = ∅;
    foreach func ∈ F do
        foreach insn ∈ func do
            if insn is a direct call to function T
              then
                F_new = F_new ∪ T;
            end
            if insn takes address of function T then
                F_new = F_new ∪ F;
            end
            if insn accesses the global variable G
              then
                F_new = F_new ∪
                  GlobalToFnMap[G];
            end
        end
    end
while F_new ≠ ∅;
F = F ∪ F_new;
```

TABLE 4: Analysis time for the baseline and PhaseSeed

	Analysis Time (in seconds)	
Application	**SVF**	**PhaseSeed**
Nginx	1385.74	1382.96
Lighttpd	190.41	136.84
Mbedtls	16.15	8.41
Monkey	114.14	31.64
Ssh-agent	16.75	10.27
Memcached	72.68	43.01
Wget	32.12	12.03
Curl	911.81	511.32

Fully-sensitive Dynamic Execution. The fully sensitive dynamic execution shows significant precision benefits only for Lighttpd where it reduces the average EC size from 9.5 to 5.4. We analyzed the results and found that in the case of Nginx, we found that while the fully sensitive dynamic execution provided initial improvements in precision, those precision gains were almost completely lost by subsequent imprecision in the static analysis stage. In the case of Monkey, ssh-agent, Memcached, and Wget, they compounding precision improvements already accounted for most of the possible precision improvements. As discussed in Section 7.3, Curl and MbedTLS show minimal improvement under PhaseSeed due to their program structure.

This shows that the different components of PhaseSeed offers different degrees of precision improvement depending on the application's structure.

Appendix E. Security Critical System Calls

To evaluate the protection that system call filters can realistically provide, earlier research first defined a list of "security-critical" system calls by systematically mining large repositories of real-world exploit payloads. Temporal Specialization [37] extracted 53 Metasploit and 514 Shellstorm payloads and, after adding equivalent calls, analyzed 1,726 payloads. In that corpus, `bind` and `select` were the most common, appearing 316 and 293 times, respectively; this metric underlies later syscall-filtering evaluations [68]. We use this data-set for our system call evaluation in Section 7.5.

Appendix F. Analysis Time

Table 4 presents the total analysis time for both the baseline SVF analysis and the complete PhaseSeed toolchain. Across all applications, PhaseSeed requires less analysis time than the baseline fully static pointer analysis technique. This is because the baseline analysis spends multiple iterations deriving imprecise pointer relationships, whereas PhaseSeed reaches the fixpoint faster. This shows that PhaseSeed not only improves the analysis precision but also improves its scalability by reducing the analysis time.