Call Graph Construction for Operating System Kernels

Sergey Ignatov
Software Engineering Department
Ivannikov Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia
ignatov.sa@ispras.ru

***Abstract*—This study focuses on the challenging area of pointer analysis in C, with a particular emphasis on constructing complete and accurate call graphs using point-to analysis for function pointers. Recognizing the NP-completeness of pointer analysis, our study aims to overcome the challenges of accurately and efficiently analyzing programs written in pointer-based languages such as C. We start with a thorough review of existing methodologies for constructing call graphs, highlighting different approaches and their applicability to different scenarios, from optimization to error detection. We introduce a flow-insensitive, context-insensitive points-to analysis algorithm specifically designed for constructing call graphs. The presented algorithm is based on the mathematical abstraction of set constraints and is efficiently solved in near linear time. Encountering the realities of parsing real code bases, we describe the problems encountered, such as pointer arithmetic, inline assembler directives, type casting, and self-modifying code - elements that together present significant barriers. Through comparative analysis, we match our proposed methodology against the existing type-based approaches, providing insights into its relative strengths of reducing the number of indirect call edges of the call graph by a factor of one. The performance of our algorithm is evaluated on a large code base such as the Linux kernel, with the ultimate goal of reducing cycles in the call graph and dividing the kernel into independent components suitable for formal verification.**

***Index Terms*—call graph, points-to analysis, interprocedural program analysis**

I. Introduction

Analyzing programs written in pointer-based languages like C requires specific approaches to handle pointers effectively, especially considering that pointer analysis itself is an NP-complete problem [1]–[3]. In many cases, pointer analysis is the foundational step that leads the way to a host of other problem solutions, as other analyses often build on its results. The one of such dependent problems is complete call graph construction that includes both direct and indirect calls, the latter executed through the use of pointers, to ensure a holistic understanding of the program's execution flow. Furthermore, building a call graph do not merely serve for understanding the execution flow; they are also crucial prerequisite for the decomposition of programs into their independent components.

There are numerous approaches to analyzing pointers, each differing in terms of accuracy, execution time, and memory requirements. Various applications of pointer analysis necessitate different approaches and methods [4]. For instance, optimization tasks may benefit from certain approaches, while error detection or program understanding may require others.

It's important to note that pointer analysis doesn't lend itself to linear scalability. To address the complexities associated with analyzing substantial program code for call graph construction problem, practitioners often turn to heuristics like type-based analysis. Nevertheless, this approach suffers from insufficient accuracy, although it has advantages in scalability, computational speed and ease of implementation. The main reason for using type-based analysis is its simplicity of implementation and absence of false-negative errors, in the worst case the result will be a large number of non-existent edges in the call graph, but all true branches will be present there. Therefore, this method is often used to analyze critical components, such as operating system kernels, in order to control security policy [5]–[9].

In this study, our focus is on points-to analysis for function pointers, with the aim of constructing a complete and precise call graph. Subsequently, we conduct a comprehensive survey of extant methodologies employed for call-graph construction. Following this, we introduce a flow-insensitive, context-insensitive point-to analysis tailored for call-graph construction. We describe the algorithm and its rationale in detail to provide a clear understanding of the operational framework. In addition, we discuss the problems inherent in dissecting real code, including such complexities as pointer arithmetic, embedded assembler directives, complex type conversion, and the presence of self-modifying code. Furthermore, we engage in a comparative analysis between our suggested algorithm and data type-centric analysis approaches. Lastly, we subject the constructed call-graph to evaluation to determine its effectiveness in addressing the task of decomposing the Linux kernel into manageable components suitable for formal verification.

A. Points-to Analysis for Function Pointers

Points-to analysis is a technique used to determine the possible values of pointers in a program. When dealing with function pointers, points-to analysis identifies the set of functions that a pointer may refer to during program execution. This information is crucial for constructing an accurate call graph, particularly if this graph is to be employed in the task of decomposing a codebase into isolated components suitable

for formal verification. Extraneous and false edges may lead to fictitious cycles in the graph, making it impossible to isolate small subgraphs.

II. Constructing call graphs

A. Call graph problem

When considering the problem of constructing a call graph, we must understand the ways a function can be called indirectly.

1) Call by a **function pointer**. Commonly used in C for dynamic dispatch and passing callbacks.
2) C++ **virtual method** invocation using vtables containing pointers to virtual methods allows dynamic dispatching depending on the object type.
3) Smalltalk-style **send-method** dynamic dispatch. Involves dynamic type lookup through class hierarchy for method selection in languages like Objective-C and JavaScript.
4) C++ **exceptions**.

B. Dynamic analysis

This class of methods represents a cost-effective and straightforward approach for constructing call graphs, particularly in the absence of source code. Indeed, its relevance is crucial for analyzing binary-level programs [10]. This can include various instrumentation techniques for recording call directions during the program execution. Expanding beyond binary analysis, this technique exhibits versatility by also addressing challenges in languages without strong typing, like JavaScript [11]. Even with Java's strong typing facilitating static analysis, the JIT compilation process introduces a need for optimizations, where dynamic analysis [12] plays a crucial role in balancing compilation speed and execution performance.

C. Type matching

Another class of simple methods is used when it is necessary to eliminate false negatives, and speed and reliability of implementation are more important than accuracy of analysis. The basic idea is to map call-sites to functions with corresponding type signatures, that works well in strong typed languages. In some cases type signatures are used in combination and improve the result of pointer analysis [13]. The significance of improvement indicate that type signatures is a fairly unique entity in programs and for this reason the results obtained by this analysis are not much worse than the exact analysis.

In the realm of security applications, employing type matching emerges as a cost-effective strategy for applying security policies and ensuring control flow integrity [5]–[9], [14], [15].

D. Static analysis

All program analyses are in general NP-complete problems [1], and pointer analysis is no exception. Therefore, there are many variants of pointer analysis [16]–[18] that operate with different generalization approaches to achieve the right balance between the complexity of the analysis and the accuracy of the results. These differences in approaches, can be categorized into the following classes.

1) flow-sensitivity: Flow-sensitive analysis considers different execution paths in a function, deriving unique solutions for different locations within a procedure. In contrast, flow-insensitive analysis makes no such distinctions, combining solutions from all paths reachable at a given point. A more relaxed version of flow-insensitive analysis accepts all expressions as possible even without considering their order of execution.

2) context-sensitivity: Context-sensitive analysis considers all variants of a function call and analyzes functions for each call context, which definitely leads to an exponential growth in the number of states and requires specialized unification techniques. Context-independent analyses do not distinguish between function call contexts, this definitely leads to a reduction in computational complexity, but loses in accuracy and leads to false positives.

3) field-sensitivity: Field-sensitive analysis looks at different parts of a structure separately, while field-insensitive analysis treats everything as one. Consider a structure with two pointer fields: field-sensitive analysis maintains separate information for each field, while field-insensitive analysis considers both fields to point to the same memory locations.

Another well-known approach, Steensgaard's algorithm [17], is to combine the stored information of a pointer graph. If two variables a and b point to A therefore the common variable a/b points to A. The opposite is also true, if a points to A and B, we can merge and represent what now points to A/B. This significantly reduce the memory and computation required for the analysis.

III. Algorithm

Our analysis focuses on function pointer values within programs. Function pointers, by nature, offer systematic simplification for analysis due to restrictions imposed by the C standard [19]. These include prohibitions against multiplication, stringent conditions for addition or subtraction, and limitations on bitwise operations (exceptions are detailed in sections III-C1). Since our goal is a comprehensive program analysis, we ignore internal control structures and represent the program as a sequence of function pointer operations. In the same way, we also abstract interprocedural pointer relations with respect to the memory in which the values of these pointers are stored. If a global variable is assigned a value at a certain point in the program, then it can have that value in all places where that variable is used.

A. Definitions

Let P is a program, where:

$$P = \{F, I_{static} \times I_{dynamic}\} \tag{1}$$

$$F_i = \{S\} \tag{2}$$

- F is a set of functions.
- S is a set of statements.

TABLE I: Abstract Pointer Operations

statement	set relations
a = &func	$a \supseteq \{func\}$
a = &b	$a \supseteq ref(b)$
a = *b	$a \supseteq deref(b)$
*a = b	$deref(a) \supseteq b$
a[i] = b	$a \supseteq b$
a = b[i]	$a \supseteq b$
a.field = b	$field \supseteq b$
a = b.field	$a \supseteq field$
func() { return a }	$func \supseteq a$
a = func()	$a \supseteq func$
func(..., i:a, ...)	$func_{param_i} \supseteq a$

- $I_{static} = I_{arch} \cup I_{fp}$ is a set of static inputs known at compile time such as architecture dependent values (I_{arch}) and function labels (I_{fp}).
- $I_{dynamic}$ is a set of runtime inputs and parameters.

Assuming that $I_{dynamic}$ has no impact on I_{fp} or the destinations of indirect control transfer allows us to isolate the subset of the program P_{fp} that involves only operations on function pointers $S_{fp} \subseteq S$.

$$P_{fp} = \{F_{fp}, I_{fp}\} \tag{3}$$
$$F_{fp} = \{S_{fp}, I_{fp}\} \tag{4}$$
$$S_{fp} \mapsto S_{abstract} \tag{5}$$

B. Abstract Pointer Operations

We can represent pointer relations as a system of set constraints [20] and turn the pointer analysis problem into a constraint set problem (CSP) by assume pointer operations as an inclusion between a set of pointers and a set of function values. In the abstract constraint-set language (see Table I), we define the *ref* and *deref* operations - taking the address and taking the pointer value, respectively.

Let us give an explanation of the semantics of our abstract operations.

1) $\{\}$ - a set of function labels.
2) a - a program's function pointer variable.
3) $ref(a)$ - operator of taking a reference to a variable.
4) $deref(a)$ - operator of obtaining a pointer value.
5) $func$ - a set of return values of a function.
6) $func_{param_i}$ - a set of values of a function parameter.

We define pointer operations, such as pointer reference, dereference, and pointer assignment, through sets relations.

$$[reference] \quad \frac{a \supseteq ref(b) \quad b \supseteq x}{a \supseteq x} \tag{6}$$
$$[dereference] \quad \frac{a \supseteq deref(b) \quad b \supseteq x}{a \supseteq x} \tag{7}$$
$$[assignment] \quad \frac{deref(a) \supseteq b \quad a \supseteq x}{x \supseteq b} \tag{8}$$

The efficient solution of CSP can be obtained thought representing the sets inclusions as a directed graph that we call *Set Transition Graph*; In which nodes are representations of subsets of corresponding elements, such as variables, function return values, function parameters, structure fields, and edges represent inclusion of one set into another.

EXAMPLE 1: Set Transition Graph of the simple program. Demonstrates the inference of set constraints for trivial operations and what kind of graphs this leads to.

```
1  typedef void (*func_t)();
2  void fn1() {}
3  void fn2() {}
4  void fn3() {}
5  struct obj_t {
6      func_t fn;
7      func_t *fp;
8  }
9  void work(struct obj_t *obj, func_t f) {
10     func_t a, p;
11
12     a = fn1;                 a ⊇ {fn1}
13     obj->fp = &a;            obj_t.fp ⊇ ref(a)
14     p = *obj->fp;            p ⊇ deref(obj_t.fp)
15
16     p();                     icall1 : p
17     obj->fn();               icall2 : obj_t.fn
18     f();                     icall3 : work.param2
19 }
20 void main() {
21     struct obj_t obj = {
22         .fn = fn2,           obj_t.fn ⊇ {fn2}
23         .fp = 0
24     };
25     work(&obj, fn3);         work.param2 ⊇ {fn3}
26 }
```

$$icall_1 \longrightarrow p \xrightarrow{deref} obj\_t.fp \xrightarrow{ref} a \longrightarrow \{fn1\}$$

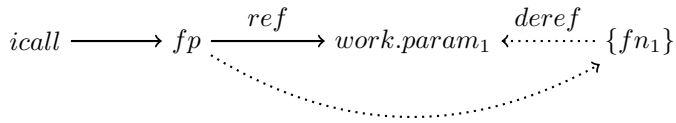
$$icall_2 \longrightarrow obj\_t.fn \longrightarrow \{fn2\}$$

$$icall_3 \rightarrow work.param_2 \rightarrow \{fn3\}$$

Let us examine a simple program (see example 1) and the corresponding set constraint for it. In the function `work` the value of the function `fn1` is assigned to the variable `a`, which in terms of sets means that `fn1` is a subset of `a`. Then the address of `a` is assigned to the field `obj->fp` of the structure `obj`. Thus, all values of `a` can be addressed via `obj->fp`, which means that `a` is a subset of `obj->fp`. Obviously, the set of a particular instance of `obj->fp` will be a subset of the type `obj_t.fp`, which generalizes access to the fields of the structure to their type. In a similar way, we also unify array elements by considering an array and every its elements as one common set. Later in the example, the pointer field `obj->fp` is assigned to the variable `p`, followed by an indirect function call $icall_1$ from the value in `p`. The set of destination

EXAMPLE 2: Propagation of pointer assignment.
Demonstrates the inference of set constraints in case of pointer assignments.

```
1 void work(func_t * fp) {
2     *fp = fn1;          deref(work.param_1) ⊇ {fn1}
3 }
4 void main() {
5     func_t fp;
6     work(&fp);          work.param_1 ⊇ ref(fp)
7     fp();               icall : fp
8 }
```



addresses for the given indirect call site is the same as `p`, which includes at least the set of `obj->fp`. For indirect calls $icall_2$ and $icall_3$ the destination addresses are sets of the field `obj_t.fp`, since we generalize field access, and the sets of `f`, which is the function's parameter $work.param_2$.

The operator $deref$ has no transitivity over $\supseteq$, for this reason, inclusion something as a subset of $deref(\tau)$ requires resolution of the set of τ, in other words, in C terms, assignment by a pointer requires finding the set of its values. In the example 2, pointer assignment turns into inclusion of the assigned value `fn1` in the set of $deref(work.param_1)$. Therefore, to solve the set system, we should take all sets of $work.param_1$ in *Set Transition Graph* and propagate $deref$ edge into the assigned value.

C. Distinguishing Abstract Model from Real C Implementation

It's important to acknowledge the differences between the abstract model and its counterpart in original implementation and actual hardware execution.

1) Pointer arithmetic: is a fundamental feature in C, allowing directly manipulate memory addresses. This capability is crucial for efficient data processing and system programming. However, the situation becomes notably different when we deal with function pointers, which refer to the locations of functions in memory. First off, it's important to understand that arithmetic on function pointers is generally undefined behavior in C. Primarily because function pointers don't work in the same way as data pointers. For data pointers, arithmetic operations like addition or subtraction make sense because data elements can be stored sequentially in memory, like the elements of an array. This is not the case for functions. Functions are not laid out back-to-back in memory; their arrangement is determined by the compiler and the operating system. As a result, adding or subtracting values to or from a function pointer doesn't carry a clear or consistent meaning.

But yet, operations on function pointers might be used in specific, low-level contexts, like managing where a piece of code resides in memory for hardware interactions. Addressing these unique cases requires specialized knowledge tailored to the specific hardware being dealt with and is beyond the scope of this article.

2) Inline assembler: as a compiler extension can be considered as a special function with inputs, outputs and clobbered registers. If the latter serves as critical information for the compiler, the explicit specification of inputs and outputs makes it possible to model inline assembler statement as a separate extension of our abstract model and only if the inputs or outputs somehow get into the Set Transition Graph, otherwise it can be ignored. Importantly, the current work and its discussions do not address this facet.

3) Type casting: in C can lead to behaviors that are difficult to model abstractly. However, the immutable nature of function pointers allows us to ignore type casting over function pointers themselves. Likewise, the approach of generalized modeling of object fields by their types allows us to avoid this problem if a program preserves the semantics of types, but for some reason uses type casting only when passing from one place to another. At the same time, if the program behavior violates the type semantics for data pointers, it may lead to loose links in the set transition graph. To ensure that this situation does not occur, we verify the integrity of the graph by validating that all function labels in the graph are reachable from all call sites and that there are no leaf nodes in the graph other than a function label.

Another problem arises when a program performs type conversion by offset from beginning of an object, including negative offset. This can be called type conversion in an arbitrary place of objects. Similarly for casting scalar types to a pointer. These problems require further study.

4) Self-modifying code: For performance reasons, the Linux kernel extensively relies on self-modifying code to implement features such as *STATIC_CALL*, *JUMP_LABEL*, *DYNAMIC_FTRACE* and *BPF*. The first is used to override the target of a particular call site, which helps avoid CPU stalls due to branch prediction. It functions similarly to an indirect call by retrieving a function pointer from a global variable and updating that variable accordingly. The latter is intended to dynamically change the macros likely/unlikely for conditional branches and does not affect the call graph. FTRACE, designed for function tracing, is optional in the kernel and not discussed further in this analysis, as we disabled this config option during our investigation. The last is the feature allows for placing dynamically generated code into the kernel. However, invoking BPF programs occurs through a single entry point. The code is generated from the BPF virtual machine and includes a set of functions allowed to be called by BPF programs, known as BPF helpers. We create artificial edges in the call graph from the entry point to the set of BPF helpers. Anyway, self-modifying code is a challenging task to analyze and requires careful investigation of each use case.

TABLE II: Performance of the call graph construction

Program	Lines of code	#of functions	#of indirect call-sites	#of indirect functions	#of CG direct edges	#of type-based indirect edges	#of static-analisys indirect edges
linux-5.15.153 tinyconfig	734.2K	23 308	939	2393	51 195	6355	2150
linux-5.15.153 defaultconfig	3.1M	107 022	8286	37 387	404 749	159 896	35 276
linux-5.15.153 allyesconfig	23M	513 335	61 237	328 874	2 643 969	5 049 734	579 091
linux-6.1.83 tinyconfig	775.6K	24 848	968	2483	52 030	6518	2203
linux-6.1.83 defaultconfig	3.3M	113 679	8500	39 477	385 376	169 919	36 876
linux-6.1.83 allyesconfig	26.4M	503 600	58 804	315 881	2 159 046	4 763 771	558 579

IV. Results

The accuracy, and scalability of our proposed points-to analysis algorithm were validated through empirical evaluation on Linux kernel versions 5.15.153 and 6.1.83. For our evaluation, we implemented algorithm of set transition graph construction in the CodeQL query language. We conducted experiments with our algorithm on a system featuring a 2.4GHz 8-core Intel Core i9 processor and 64GB of RAM. The results was compared to signature matching approach, the evaluation criterion being the reduction of the number of edges in the call graph.

Summary: Compared to the type-based approach, our algorithm has reduced the number of indirect edges of the call graph by 4-8 times (see table II). Moreover, the larger the code base size is, the more efficient the algorithm is. This is due to the fact that with increasing code size, in the type-based approach, the number of indirect edges grows faster than the size of the code, accumulating false positives.

V. Conclusions and future work

Preliminary results demonstrate the algorithm's potential for decomposing complex code structures such as the Linux kernel. Representing the pointer operation to a set relation turns the problem into a comprehensible mathematical problem. Generalization of fields and access to arrays allows to make the analysis simple while maintaining an acceptable level of accuracy and avoiding exponential growth of computational complexity.

Looking ahead, there are several directions for future exploration and development:

- **Comparative Analysis of Static Analysis Methods:** The next step is to compare our methodology with a number of other static analysis approaches and different sensitivities. This can provide an understanding of the necessary sufficiency to obtain the preferred results. Moreover, extending the application scope of our method to user space software and libraries will be critical in determining its generalized applicability across different software paradigms.
- **Generalization of Dynamically Allocated Objects and Arrays:** An important area for further research is generalization strategies for arrays and dynamically allocated objects. Evaluating the impact of such generalizations on the performance, accuracy and precision of our algorithm will contribute to its usefulness.
- **Modeling Inline Assembler and Hardware Semantics:** The presence of inline assembly code introduces additional complexity to program analysis. Although we can ignore the inline assembler in some cases, in general we need to model the operations and take into account the semantics of the hardware. Future work will focus on developing extensions that accurately address these aspects.
- **Addressing Type Conversion Challenges:** Casting types at arbitrary locations poses significant challenges. Consequently, we aim to thoroughly explore these scenarios, developing robust modeling techniques that accurately reflect program semantics.

References

[1] V. T. Chakaravarthy and S. Horwitz, "On the non-approximability of points-to analysis," *Acta Informatica*, vol. 38, no. 8, pp. 587–598, Jul. 2002. [Online]. Available: http://link.springer.com/10.1007/s00236-002-0081-8

[2] W. Landi and B. G. Ryder, "Pointer-induced aliasing: a problem taxonomy," in *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '91*. Orlando, Florida, United States: ACM Press, 1991, pp. 93–103. [Online]. Available: http://portal.acm.org/citation.cfm?doid=99583.99599

[3] S. Horwitz, "Precise flow-insensitive may-alias analysis is NP-hard," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, pp. 1–6, Jan. 1997. [Online]. Available: https://dl.acm.org/doi/10.1145/239912.239913

[4] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. Snowbird Utah USA: ACM, Jun. 2001, pp. 54–61. [Online]. Available: https://dl.acm.org/doi/10.1145/379605.379665

[5] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, vol. 50, no. 1, pp. 1–33, Jan. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3054924

[6] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Edinburgh United Kingdom: ACM, Jun. 2014, pp. 577–587. [Online]. Available: https://dl.acm.org/doi/10.1145/2594291.2594295

[7] Y. Zhang, X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma, "ReCFA: Resilient Control-Flow Attestation," in *Annual Computer Security Applications Conference*. Virtual Event USA: ACM, Dec. 2021, pp. 311–322. [Online]. Available: https://dl.acm.org/doi/10.1145/3485832.3485900

[8] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. Saarbrucken: IEEE, Mar. 2016, pp. 179–194. [Online]. Available: http://ieeexplore.ieee.org/document/7467354/

[9] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 6, pp. 1535–1550, Jun. 2018. [Online]. Available: http://ieeexplore.ieee.org/document/8269390/

[10] R. Jalan and A. Kejariwal, "Trin-trin: Who's calling? a pin-based dynamic call graph extraction framework," *International Journal of Parallel Programming*, vol. 40, pp. 410–442, 2012.

[11] T. R. Toma and M. S. Islam, "An efficient mechanism of generating call graph for javascript using dynamic analysis in web application," in *2014 International Conference on Informatics, Electronics & Vision (ICIEV)*. IEEE, 2014, pp. 1–6.

[12] T. Xie and D. Notkin, "An empirical study of java dynamic call graph extractors," *University of Washington CSE Technical Report*, pp. 02–12, 2002.

[13] D. Atkinson, "Accurate Call Graph Extraction of Programs with Function Pointers Using Type Signatures," in *11th Asia-Pacific Software Engineering Conference*. Busan, Korea: IEEE, 2004, pp. 326–335. [Online]. Available: http://ieeexplore.ieee.org/document/1371935/

[14] B. Niu and G. Tan, "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. Scottsdale Arizona USA: ACM, Nov. 2014, pp. 1317–1328. [Online]. Available: https://dl.acm.org/doi/10.1145/2660267.2660281

[15] D. Zeng, B. Niu, and G. Tan, "MazeRunner: Evaluating the Attack Surface of Control-Flow Integrity Policies," in *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. Shenyang, China: IEEE, Oct. 2021, pp. 810–821. [Online]. Available: https://ieeexplore.ieee.org/document/9724388/

[16] L. O. Andersen, "Program analysis and specialization for the c programming language," 1994.

[17] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96*. St. Petersburg Beach, Florida, United States: ACM Press, 1996, pp. 32–41. [Online]. Available: http://portal.acm.org/citation.cfm?doid=237721.237727

[18] X.-X. S. Zhang, *Practical pointer aliasing analysis*. Rutgers The State University of New Jersey, School of Graduate Studies, 1998.

[19] "American national standard for information systems - ansi x3.159-1989 ; programming language - c," 1989. [Online]. Available: https://api.semanticscholar.org/CorpusID:197537566

[20] A. Aiken, "Set constraints: Results, applications and future directions," in *International Workshop on Principles and Practice of Constraint Programming*. Springer, 1994, pp. 326–335.