Flow-Sensitive Type-Based Heap Cloning

Mohamad Barbar
University of Technology Sydney, Australia
CSIRO's Data61, Sydney, Australia

Yulei Sui[1]
University of Technology Sydney, Australia

Shiping Chen
CSIRO's Data61, Sydney, Australia

Abstract

By respecting program control-flow, flow-sensitive pointer analysis promises more precise results than its flow-insensitive counterpart. However, existing heap abstractions for C and C++ flow-sensitive pointer analyses model the heap by creating a single abstract heap object for each memory allocation. Two runtime heap objects which originate from the same allocation site are imprecisely modelled using one abstract object, which makes them share the same imprecise points-to sets and thus reduces the benefit of analysing heap objects flow-sensitively. On the other hand, equipping flow-sensitive analysis with context-sensitivity, whereby an abstract heap object would be created (cloned) per calling context, can yield a more precise heap model, but at the cost of uncontrollable analysis overhead when analysing larger programs.

This paper presents TypeClone, a new type-based heap model for flow-sensitive analysis. Our key insight is to differentiate concrete heap objects lazily using type information at use sites within the program control-flow (e.g., when accessed via pointer dereferencing) for programs which conform to the strict aliasing rules set out by the C and C++ standards. The novelty of TypeClone lies in its lazy heap cloning: an untyped abstract heap object created at an allocation site is killed and replaced with a new object (i.e. a clone), uniquely identified by the type information at its use site, for flow-sensitive points-to propagation. Thus, heap cloning can be performed within a flow-sensitive analysis without the need for context-sensitivity. Moreover, TypeClone supports new kinds of strong updates for flow-sensitive analysis where heap objects are filtered out from imprecise points-to relations at object use sites according to the strict aliasing rules. Our method is neither strictly superior nor inferior to context-sensitive heap cloning, but rather, represents a new dimension that achieves a sweet spot between precision and efficiency. We evaluate our analysis by comparing TypeClone with state-of-the-art sparse flow-sensitive points-to analysis using the 12 largest programs in GNU Coreutils. Our experimental results also confirm that TypeClone is more precise than flow-sensitive pointer analysis and is able to, on average, answer over 15% more alias queries with a no-alias result.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Heap cloning, type-based analysis, flow-sensitivity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.24

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at
`https://doi.org/10.4230/DARTS.6.2.1`.
`https://github.com/SVF-tools/SVF/wiki/TypeClone`

Funding This research is supported by Australian Research Grant DP200101328.
Mohamad Barbar: supported by a PhD scholarship funded by CSIRO's Data61.

Acknowledgements We would like to thank the anonymous reviewers for their helpful comments.

[1] Corresponding author

1 Introduction

Pointer analysis aims to determine the objects which a pointer may point to at runtime. It is an enabling technology which forms a basis for other program analysis tasks such as compiler optimisation [28, 13], program slicing [42, 43], enforcing control-flow integrity [23, 16], and software security analysis [38, 32].

1.1 Background

Flow-sensitivity is an essential precision dimension of pointer analysis. Unlike flow-insensitive analysis which considers instructions to be unordered, flow-sensitive analysis accounts for program execution order to better approximate runtime behaviour and achieve a more precise result. Traditional flow-sensitive points-to analysis computes and maintains points-to relations at each program point. These points-to relations are propagated along the program's control-flow graph (CFG) by solving an iterative data-flow problem until a fixed point is reached.

In recent years, there have been significant advances in making flow-sensitive analysis for C and C++ programs more efficient. Rather than propagating all relations to/from each program point on the CFG, SPARSE analysis [19, 20], a flow-sensitive and context-insensitive analysis, pre-computes over-approximated value-flows (def-use chains) to produce a value-flow graph upon which the main phase analysis can propagate points-to relations sparsely. This reduces both time and space overhead while maintaining precision. Selective flow-sensitive analysis [30] performs strong updates for stack and global variables at stores where flow-sensitive singleton points-to sets are available, but falls back to flow-insensitive results otherwise, making a trade between efficiency and precision. SPARSE analysis incorporates these strong updates. As an alternative to whole-program analysis, demand-driven flow-sensitive analysis [39] aims to answer points-to queries flow-sensitively by only analysing specific parts of a program with CFL-reachability on the pre-computed value-flow graph.

1.2 Motivation and insights

Most recent advances in C/C++ flow-sensitive pointer analysis focus on improving efficiency. Apart from those which employ expensive context-sensitivity, existing solutions exclusively use an *allocation-site-based* heap abstraction where an abstract object is created per memory allocation site to represent the set of concrete objects created at that allocation site during runtime. This is especially coarse for heap-intensive programs when allocation sites are contained within allocation wrapper functions [7] since any pointers pointing to objects originating from such wrapper functions will be regarded as having a may-alias relation despite originating in different contexts at runtime. Thus the heap abstraction significantly reduces the benefit of flow-sensitivity by restricting most of the precision improvement to stack and global variables only. For example, given two concrete heap objects o_1 and o_2 which originate from the same allocation wrapper and are accessed along two different control-flow branches, the flow-sensitive points-to results of the heap objects, and of any pointers which point to them, would be as (im)precise as flow-insensitive results since the heap abstraction would treat both concrete objects as a single abstract object. Whenever a single allocation wrapper is used exclusively, the heap abstraction is as precise as having one abstract object represent all concrete heap objects allocated by the program.

Developing a precise and efficient flow-sensitive heap abstraction is challenging since the infinite-sized heap needs to be partitioned into a finite a number of abstract objects. A straightforward solution is to equip flow-sensitive analysis with context-sensitivity. A

context-sensitive analysis performs heap cloning based on calling contexts: a new heap object (a clone) is created for each calling context reaching the object's allocation site. However, given the billions of calling contexts in large programs [44], adding context-sensitivity to flow-sensitive analysis incurs uncontrollable overhead even with *k-limiting* enabled [3].

Rather than heap cloning according to calling context, we would like to investigate the use of type information and the initialisation of an object's type (e.g. through dereferencing a pointer to that object) to perform heap cloning. Our key insight is based on the strict aliasing rules laid out by the C [2, §6.5 ¶7] and C++ [1, §6.10 ¶8] standards whereby a pointer may only access an object with a compatible type (or else, the program exhibits undefined behaviour). For example, reading an object of type `float` through a pointer of type `int *` is undefined behaviour. Thus, a pointer of type `int *` and a pointer of type `float *`, for example, cannot be referring to the same object when being dereferenced. We can then use such type information to "separate" concrete objects from within an abstract object (of the allocation-site-based abstraction) into multiple abstract objects, each used for accesses of a specific type. More concretely, an untyped heap object o can be cloned into multiple typed objects based upon the accesses (or dereferences) of o. Therefore, type-based heap cloning could more precisely distinguish heap objects while avoiding expensive context-sensitive heap modeling.

1.3 Existing efforts and limitations

Type-based heap modelling [24, 26, 37] has been used in strongly typed languages, such as Java, to produce a cost-effective heap abstraction for context-sensitive but flow-insensitive pointer analysis. However, there are very few attempts at type-based pointer analysis for C and C++ [5, 7, 13]. C and C++ introduce new challenges:

- C and C++ are weakly typed so reasoning about the types of objects, especially heap objects, is difficult.
- High-level C/C++ type information is not preserved in the intermediate representation (IR) of modern compilers, like Clang/LLVM [14], upon which static analyses typically operate on.
- C and C++ allow for the address of fields to be taken and for pointer arithmetic within an object, which means fields' types must too be resolved during the analysis.

It is hard to design a fully sound pointer analysis, incorporating types, for non-conforming programs which violate the strict aliasing rules. Generally, a fully sound analysis would be unscalable or imprecise almost to the point of uselessness on its own [31]. Type-based alias analysis (TBAA) [13] made the initial attempt at exploiting the strict aliasing assumption (or its equivalent in Modula-3) to produce a fast alias analysis. The almost stateless nature of TBAA makes it especially useful in resolving alias queries that would otherwise require inter-proecedural analysis to non-trivially answer. Modern compilers like LLVM and GCC implement TBAA behind the `-fstrict-aliasing` flag (enabled by default). In reality, programs which wish to safely benefit from optimisations enabled by TBAA must conform to the strict aliasing rules. Programs which fail to do so either risk miscompilation (since the program exhibits undefined behaviour the compiler assumes is not present) or must notify the compiler that the program violates the strict aliasing rules and do without those optimisations. Violating these rules (or invoking any other undefined behaviour) is typically strongly discouraged, particularly where safety is a factor, as in the MISRA C and C++ coding standards, for example.

Recently, structure-sensitive analysis [7] (hereon referred to as `cclyzer-ss`) presented a type-based flow-insensitive points-to analysis that enhances the precision of Andersen's analysis [4]. `cclyzer-ss` lazily infers the types of heap objects by leveraging LLVM type casts to filter out spurious field derivations (i.e., field derivations introduced by static imprecision that can never happen during runtime).

1.4 Our solution

Inspired by TBAA and `cclyzer-ss`, the scope of this work is to produce a more precise flow-sensitive heap model for C and C++ programs which follow the strict aliasing rules. We propose TypeClone, a flow- (and field-) sensitive analysis with a new type-based heap abstraction which yields better precision than a traditional flow-sensitive analysis. We aim to perform lazy heap cloning by incorporating lightweight type information within standard flow-sensitive pointer analysis. Rather than performing heap cloning per calling context for each allocation, for untyped memory object o allocated at program point ℓ, we lazily clone to create typed object o_t at each of its type initialisation points ℓ' where the object o has a type t assumed. To maintain soundness, each clone o_t is back-propagated to any pointer that may have actually been accessing the concrete objects now represented by o_t through o.

Intuitively, the type initialisation point is treated as the real allocation site during our lazy heap cloning, thus distinguishing different sets of concrete objects where necessary in a lazy rather than eager manner. From the type initialisation point ℓ' forward, we only propagate the clone object o_t rather than the untyped object o. Not only are untyped objects prevented from propagating past type initialisation points like ℓ', but any object of type t accessed by a pointer with incompatible element type t' will be strongly updated (i.e. killed, filtered) because such a points-to relation is impossible in a program which adheres to the strict aliasing rules and must be a result of static imprecision. This reduces the number of spurious objects in points-to sets during points-to resolution, especially when a killed object would have otherwise created spurious field sub-objects via field-sensitivity. This gives us a flow-sensitive analysis that is still scalable yet more precise than that which uses standard heap abstractions. We see our work more as an addition to flow-sensitive analysis rather than as competition to other heap cloning techniques like context-sensitivity. Our technique is neither strictly superior nor inferior to context-sensitive analysis and can work with context-sensitivity to achieve a more precise result. Our key contributions are summarised as follows:

- We present TypeClone, a flow-sensitive pointer analysis which can perform lazy heap cloning using types without context-sensitivity for C and C++ programs which do not violate the strict aliasing rules.
- We present new forms of strong updates, namely type-based semi-strong updates and type-based strong updates, which improve precision.
- We have implemented TypeClone and compared it to sparse flow-sensitive analysis. We have found that TypeClone can answer over 15% more alias queries with a no-alias result, on average, than Sparse.

2 A motivating example

Figure 1a gives an example of a common heap allocation wrapper extracted from GNU Coreutils [18] and usage of that wrapper. This example aims to demonstrate the key idea of TypeClone and compare it with existing C and C++ points-to analyses: a recent flow-insensitive analysis (`cclyzer-ss`) and a flow-sensitive and context-insensitive analysis

```
1  int main(void) {
2    int *p = (int *)xmalloc(4);
3    *p = 1;
4    float *q = (float *)xmalloc(4);
5    *q = 1.0;
6    // Alias(p, q)?
7  }
8
9  void *xmalloc(size_t n) {
10   void *x = malloc(n);
11   if (!x && n != 0) xalloc_die();
12   return x;
13 }
```

(a) Usage of GNU Coreutils `malloc` wrapper.

Analysis	Points-to relations	ℓ_6: Alias(p,q)
`cclyzer-ss`	$\{o, o_{\texttt{int}}, o_{\texttt{float}}\} \in pt(\texttt{p})$ $\{o, o_{\texttt{int}}, o_{\texttt{float}}\} \in pt(\texttt{q})$	True
Sparse	$\{o\} \in pt(\ell_2\text{–}, \texttt{p})$ $\{o\} \in pt(\ell_4\text{–}, \texttt{q})$	True
TypeClone	$\{o, o_{\texttt{int}}, o_{\texttt{float}}\} \in pt(\ell_2, \texttt{p})$ $\{o_{\texttt{int}}\} \in pt(\ell_3\text{–}, \texttt{p})$ $\{o, o_{\texttt{int}}, o_{\texttt{float}}\} \in pt(\ell_4, \texttt{q})$ $\{o_{\texttt{float}}\} \in pt(\ell_5\text{–}, \texttt{q})$	False

(b) Points-to relations.

Figure 1 A motivating example.

(Sparse). Points-to results are shown in Figure 1b. Flow-sensitive analyses maintain different points-to information for the same pointer at different program points, so pt takes an extra argument indicating the program point (ℓ– means ℓ onwards). The results show that TypeClone is more precise than `cclyzer-ss` and Sparse when performing an alias query on `p` and `q` at line 6 and maintains only one object at lines 3 and 5 onward.

`cclyzer-ss` performs heap cloning at cast instructions, i.e., $p = (t)\ q$, such that the untyped object o that q points to is cloned to create a new object o_t with type t. The o_t is then propagated back to the allocation site of o in order to maintain soundness since some preceding program points may have become aliases of q, through points-to relations with the untyped object, before the clone was created. `cclyzer-ss`'s main goal is to enable more precise field-sensitivity through these typed heap objects. Given a field constraint, i.e., $p = \&q \rightarrow f$, a new field is derived only when (1) q points to a typed object o_t, and (2) the type of q is $t*$. This prevents the generation of spurious field objects. `cclyzer-ss` can only improve precision by preventing the generation of spurious field objects since every clone is back-propagated to its original allocation site which causes every object which originally pointed to the untyped object from that allocation site to point to every typed clone. For example, the clone object $o_{\texttt{int}}$, created at line 2, is back-propagated to line 10, which makes pointer `q`, soundly but imprecisely, also point to $o_{\texttt{int}}$ at every program point.

Sparse [20] performs flow-sensitive points-to analysis using allocation-site-based heap modeling. Therefore, the program only has one heap object o which is then pointed to by both pointers, `p` and `q`, at line 6.

TypeClone clones heap objects lazily only at the type initialisation point in a flow-sensitive manner. A type initialisation point is any program point in which an object must be of a certain type (or one of a set of types) for the program to be legal (i.e., not exhibit undefined behaviour). In this example, the pointer dereferences at lines 3 and 5 must be referring to objects of type `int` and `float`, respectively, for the program to avoid undefined behaviour. This is in contrast to `cclyzer-ss` which clones at cast instructions eagerly. The cloned objects are then propagated back to their allocation sites for the same reason that `cclyzer-ss` does and so `x` at line 10 would then point to the two cloned objects (as well as the original untyped object). When the clone objects $o_{\texttt{int}}$ and $o_{\texttt{float}}$ are re-propagated to line 3 following the control-flow, $o_{\texttt{float}}$ will be filtered by TypeClone's type-based strong updates since our analysis only expects an `int`-typed object ($o_{\texttt{int}}$) to be accessed by the

Table 1 Domains and LLVM-like instructions used by our pointer analysis.

Analysis domains			LLVM-like instruction set	
ℓ	$\in \mathcal{L}$	instruction labels	STK/GLOB	$p = \&o$
$t, \bullet$	$\in \mathcal{T}$	types	HEAP	$p = malloc_o$
k	$\in \mathcal{C}$	constants	PHI	$p = \phi(q, r)$
i	$\in \mathcal{S}$	stack virtual registers	FIELD	$p = \&q \rightarrow f_k$
g	$\in \mathcal{G}$	global variables	CAST	$p = (t)\ q$
p, q, r, x, y	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	top-level variables	LOAD	$p = *q$
$\dot{o}$	$\in \mathcal{O}$	abstract objects	STORE	$*p = q$
$o.f_k$	$\in \mathcal{F}$	abstract field objects	CALL	$p = q(r_1, \ldots, r_n)$
a, o	$\in \mathcal{A} = \mathcal{O} \cup \mathcal{F}$	address-taken variables	FUNENTRY	$fun(r_1, \ldots, r_n)$
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{A}$	program variables	FUNEXIT	$ret_{fun}\ p$

dereference `*p`. Similarly, a type-based strong update is also performed at line 5 to kill the incompatible typed object $o_{\texttt{int}}$. Thus, TYPECLONE is able to more precisely answer the alias query at line 6 than both SPARSE and `cclyzer-ss`.

3 Program representation and type model

This section describes the program representation, our type model (based on the C and C++ standards), and the value-flow representation used for our flow-sensitive analysis.

3.1 Program representation

Like [7, 20, 30, 39], we perform our pointer analysis on top of the LLVM IR of a program. The instructions relevant to our analysis and the domains are given in Table 1. The set of all variables $\mathcal{V}$ is separated into two subsets: $\mathcal{A} = \mathcal{O} \cup \mathcal{F}$ which contains all possible abstract objects and their fields (i.e., *address-taken variables* of a pointer), and $\mathcal{P}$ which contains *top-level variables*, including stack virtual registers (symbols starting with `%` in LLVM) and global variables (symbols starting with `@`). Top-level variables in $\mathcal{P}$ are explicit and directly accessed, and address-taken variables in $\mathcal{A}$ are implicit and indirectly accessed at LLVM LOAD or STORE instructions through top-level variables. Since our analysis is type-based, we require types: $t \in \mathcal{T}$. $\bullet \in \mathcal{T}$ is the undefined type which is necessary because heap objects can be untyped before being initialised.

After SSA (static single assignment) conversion, given that $p, q, r_1, \ldots, r_n \in \mathcal{P}$ and $o \in \mathcal{A}$, a program is represented by ten types of instructions: (1) eight types of instructions which appear in the body of a function: $p = \&o$ (allocates memory for a stack or global object), $p = malloc_o$ (allocates memory for a heap object), $p = \phi(q, r)$ (selects the value of a variable at the joint point of branching control-flow), $p = \&q \rightarrow f_k$ (retrieves a pointer pointing to the field of a struct object), $p = *q$ (reads the value of an object), $*p = q$ (writes the value of an object), $p = (t)\, q$ (casts a pointer to type t), and $p = q(r_1, \ldots, r_n)$ (calls function q with arguments $r_1, \ldots, r_n$), and, (2) a FUNENTRY instruction $fun(r_1, \ldots, r_n)$ containing the parameters of fun, and a FUNEXIT instruction $ret_{fun}\ p$ representing the unique return of fun. LLVM pass `UnifyFunctionExitNodes` is executed before pointer analysis to ensure that every function has only one FUNEXIT instruction. Top-level variables are put directly in SSA form, while address-taken variables are only accessed indirectly via LOAD or STORE instructions. Parameter passing and returning are treated as COPYs.

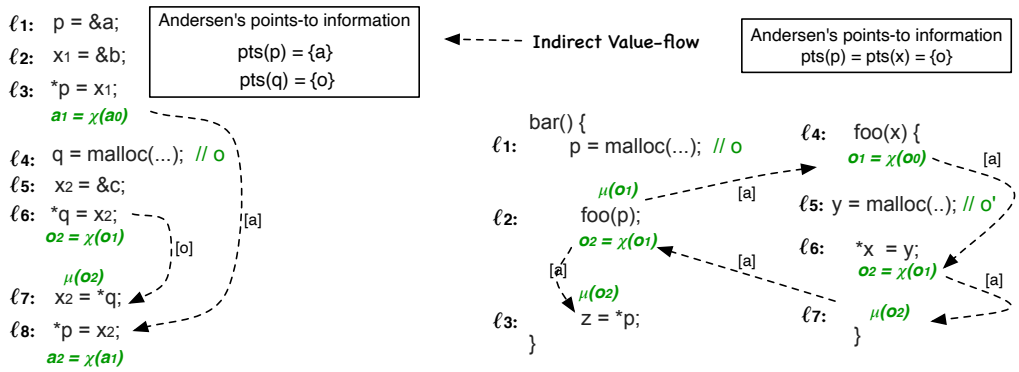
```
1   p = &a;
2   a = &b;
3
4
5   q = malloc(...);
6   *q = &c;
7
8
9   *p = *q;
10
```

(a) C code fragment.

```
1   p = &a;
2   x1 = &b;
3   *p = x1;
4
5   q = malloc(...);
6   x2 = &c;
7   *q = x2;
8
9   x3 = *q;
10  *p = x3;
```

(b) Corresponding LLVM IR.

Figure 2 C code and its corresponding LLVM IR.



(a) An intra-procedural value-flow example.

(b) An inter-procedural value-flow example.

Figure 3 Intra-procedural and inter-procedural value-flow graph examples.

C++ object allocation, like `S *x = new S()`, is generally translated into two instructions: an allocation instruction, like $x = malloc_o$, which returns an untyped object, and a call to a constructor, like `S(x)`, which calls `S`'s constructor to initialise the newly allocated object. For pointer arithmetic, as in $q = p + j$, if p points to an object o, we conservatively assume that q can point anywhere within o. This is based on the assumption that pointer arithmetic cannot cross the boundary of an object. A pointer pointing to an object produced through an integer cast aliases every other pointer (i.e. it points to every object). We treat arrays monolithically such that accessing any element of an array is the same as accessing the entire array object meaning our analysis is array-insensitive. This is an orthogonal dimension of precision.

Given $\mathtt{p}, \mathtt{q}, \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3 \in \mathcal{P}$ and $\mathtt{a}, \mathtt{b}, \mathtt{c} \in \mathcal{A}$, Figure 2 shows a code fragment and its corresponding LLVM partial SSA form. Note that address-taken variable `a` can only be indirectly accessed by introducing a top level pointer, for example through $\mathtt{x}_1$ in store `*p = x1`. Complex statements like `*q = &c` and `*p = *q` are decomposed into basic instructions by introducing top-level pointers like $\mathtt{x}_2$ and $\mathtt{x}_3$.

3.2 Value-flow representation for flow-sensitive analysis

Unlike a flow-insensitive analysis which ignores program execution order, a flow-sensitive analysis accounts for the program's control flow. Traditional flow-sensitive points-to analysis computes and maintains data-flow facts (points-to relations) at each program point. These

data-flow facts are propagated along the program's inter-procedural control-flow graph (ICFG) until a fixed point is reached [12, 30]. However, in time and space, computing and propagating the points-to information on the ICFG is costly.

To accelerate performance, the analysis is run on top of a sparse value-flow graph (VFG) instead of the ICFG [20, 33, 36, 39]. Intuitively, the VFG is a sparse def-use graph where each node represents a program statement and edges represent a def-use chain (i.e. value-flow) of a program variable. The notion of sparsity comes from the omission of statements irrelevant to the value-flow of a variable such that an edge acts as a "jump" directly from the definition of a variable to its use thus reducing redundant points-to propagation along the control-flow.

The value-flow of a top-level variable (which has a unique definition in the partial SSA form) is directly available without pointer analysis. Such value-flows are called *direct value-flows*. A directed edge from node x to node y represents a value-flow relation: a variable defined at node x is used at node y. On the other hand, value-flows of address-taken variables (which are not in partial SSA form) are obtained by building the memory SSA form following [11, 20] because their uses, at loads for example, could be defined indirectly at multiple stores

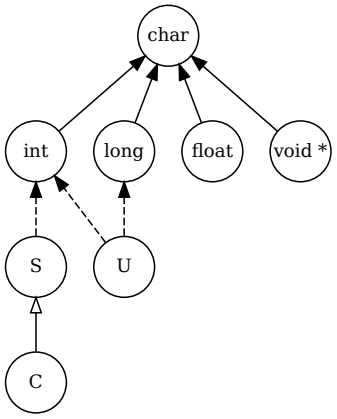
Figure 3 shows an intra- and an inter-procedural example based on the code in Figure 2. For both, a fast and imprecise flow-insensitive Andersen's analysis [4] is used to annotate indirect memory accesses at program points like loads, stores, and callsites. The results of that Andersen's analysis are shown in the two boxes in Figures 3a and 3b. Now, let us consider the intra-procedural example in Figure 3a specifically. Firstly, we annotate store instructions like $*p = x_1$ with a function $a = \chi(a)$ for each variable $a \in \mathcal{A}$ which may be pointed to by p. This represents a potential value-flow (i.e., def and use of a) at the store. If a can be strongly updated, then a receives the value on the right-hand side of the store (x_1) and the old contents in a are killed. Otherwise, a weak update takes place by adding the right-hand side to a's old contents [30]. Secondly, we annotate load instructions like $x_2 = *q$ with a function $\mu(o)$ for each variable $o \in \mathcal{A}$ that may be pointed to by q to represent a potential use of o at the load. Thirdly, we convert all address-taken variables into SSA form, treating each $\mu(o)$ as a use of o and each $o = \chi(o)$ as both a def and a use of o. Finally, we obtain the indirect value-flows for $o \in \mathcal{A}$: for a use of o, identified as o_n (where n represents the version), at load or store instruction ℓ, with its unique definition at store ℓ', $\ell' \xrightarrow{o} \ell$ represents the potentially *indirect value-flow* of o from ℓ' to ℓ. This is exemplified by $\ell_3 \xrightarrow{a} \ell_8$ and $\ell_6 \xrightarrow{o} \ell_7$ in Figure 3a.

Figure 3b shows an inter-procedural value-flow example. In addition to what is done in the intra-procedural case, we compute the side-effects of a function call by applying a lightweight inter-procedural mod-ref analysis [41, §4.2.1]. A given callsite, ℓ, is annotated with $\mu(a)$, or $a = \chi(a)$, if a may be read, or modified, respectively, inside the callees of ℓ as discovered by Andersen's pointer analysis. Additionally, appropriate χ and μ operators are also added to the FunEntry and FunExit instructions of these callees in order to mimic parameter passing and returning of address-taken variables.

To handle read side-effects within a function, we add a μ call before appropriate callsites to represent potential uses of μ's argument. The corresponding FunEntry instruction is annotated with an appropriate χ call. For example, to represent potential uses of o in foo in Figure 3b, $\mu(o)$ is added before the callsite at ℓ_2, and foo's FunEntry instruction at ℓ_4 is annotated with $o = \chi(o)$ to receive the values of o passed from ℓ_2. Similarly, for modification side-effects within a function, a call to χ is added after appropriate callsites to receive potentially modified values, and the corresponding functions' FunExit instructions are annotated with a μ call. In Figure 3b, $o = \chi(o)$ is added after the callsite at ℓ_2 to handle

```
1  struct S {
2    int i;
3    void *v;
4  };
5  class C : public S {
6    float f;
7    virtual void foo(void) { }
8  };
9  union U {
10   long l;
11   int i;
12 };
```

(a) C++ code fragment of type declarations.



(b) Corresponding type graph.

Figure 4 A type graph and the corresponding C++ code it was generated from. The open triangle arrow represents an inheritance relation, the dashed arrow represents a first-field relation, and the solid arrow represents relations derived from the strict aliasing rules of C and C++.

potential modification of o in foo, and foo's FunExit instruction at ℓ_7 is annotated with $\mu(o)$. Finally, as in the intra-procedural example, all variables in μ and χ are renamed and converted into memory SSA form when connecting their def-use relations.

3.3 Type model

The types of stack and global objects are static and are determined at their allocation sites whereas dynamically allocated heap objects (e.g., through `malloc` or C++'s `new`) are untyped. Primarily, the type of a heap object is manifested when accessed through the dereferencing of a pointer.

Before introducing the type model for our analysis, we first define the subtyping relations that may appear in a program. We use a type graph to represent subtyping relations between different types in a program. The type graph is a directed acyclic graph where each node represents a type in the target program and each edge from t' to t represents one of three subtyping relations: (1) t' inherits from t, (2) the first field of struct type t' has type t, and (3) subtyping relations derived from the strict aliasing rules of C and C++. All types appearing in the program are placed into the type graph, but we treat arrays and pointers as equivalent for this purpose and, in conformance with the strict aliasing rules, ignore signedness and qualifiers (e.g., `int` is deemed equivalent to `const unsigned int`).

Figure 4 illustrates the three subtyping relations on an example type graph (Figure 4b) generated from the code in Figure 4a. Class `C` inherits from struct `S` which has an `int` as its first field. Unions treat all their members as their first field, as is the case with `U` and its `int` and `long` members. Finally, all types which do not have any outgoing edges are connected to the `char` type in line with the strict aliasing rules which allow any object to be accessed by dereferencing a pointer to a `char`. Finally, we say that t' is a subtype of t, denoted as $t \prec t'$, if t is reachable from t' in the type graph.

Our analysis expects target programs to conform to our type model. We define the type model of our analysis with the following rules **R1**–**R5**, adopted from the provenance and strict aliasing rules of C18 [2] and C++17 [1].

R1. An abstract heap object's initial type is undefined ($\bullet$) until a non-`void` type is potentially assigned.
R2. Any pointer may point to any object regardless of its type (through pointer casting, for example). However, through a pointer of type $t*$ only objects whose type is t', such that $t \prec t'$, may be read.
R3. For pointer arithmetic $q = p + j$, q will either point within the object pointed to by p, or at one past the last element of the object if it is an array object.
R4. An object may not access any of its virtual methods until it is passed to the corresponding constructor.
R5. An object's type can be changed to a type that is not a transitive base (i.e., reuse).

R1–**R4** are easy to understand. **R5** describes object reuse in C and C++. An example use case is to reuse an already allocated object rather than freeing that object and performing a new allocation. This is commonly done using placement `new` in C++. Another use case would be custom allocators in C++ using a statically allocated pool of memory, e.g., given a static buffer (or allocated otherwise), `char buf[100]`, a placement new operation, `new(buf) T()`, would not allocate new memory, but would call `T`'s constructor with `buf` as the `this` argument to initialise the underlying object with type `T`. In C, placement `new` is unavailable. Outside well know patterns like pool allocators, object reuse is an uncommon feature that is error-prone since it may make pointers illegal to dereference, unless strict conditions are met. This is similar to introducing dangling pointers through deallocation except that deallocation is usually more explicit (e.g., through the presence of `free` and `delete`) and more commonly understood by the programmer than the possible ways of performing reuse. We discuss object reuse in more detail in Section 4.2.

4 TypeClone approach

We present a base analysis in Section 4.1 that will achieve our goals of typing heap objects and performing flow-sensitive heap cloning without context-sensitivity. The base analysis assumes no direct object reuse. We then extend the base analysis to support direct object reuse in Section 4.2.

4.1 Base analysis

This section introduces our base analysis. Central to our analysis is typing the usually untyped abstract objects. We use the notation o_t to indicate that the type of object o is t and we use o without the type subscript when the type is irrelevant. For stack and global objects, the type is assigned at allocation, whereas for heap objects, the type is undefined (denoted as $\bullet$) at allocation. According to our type model in Section 3.3, pointers with element type t can only read from objects whose underlying type t' satisfies $t \prec t'$. Therefore, such pointer accesses are an indication of the type of an object and we can use this information for heap cloning. Writes to an object are another indication of the type of an object: an object written to through a pointer with element type t is assigned type t. Figure 5 presents the inference rules for the base analysis and an explanation of each of the rules follows.

4.1.1 Memory allocation (`[HEAP]` `[STACK/GLOBAL]`)

The `[HEAP]` and `[STACK/GLOBAL]` rules handle the allocation of heap, stack and global objects. Allocation is handled as in standard flow-sensitive pointer analysis except that we associate a type with the newly allocated objects. At allocation time, the type of a heap

[STACK/GLOBAL]
$$\frac{\ell : p = \&o \quad t = T(p)}{\boxed{o_{\tilde{t}}} \in pt(\ell, p)}$$

[HEAP]
$$\frac{\ell : p = malloc_o}{\boxed{o_\bullet} \in pt(\ell, p)}$$

[LOAD]
$$\frac{\ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \ell' \xrightarrow{o} \ell \quad o_t \in pt(\ell'', q) \quad o' = \mathbf{init}(T(q), o_t)}{pt(\ell', o') \subseteq pt(\ell, p)}$$

[STORE]
$$\frac{\ell : *p = q \quad \ell' \xrightarrow{o} \ell \quad \ell'' \xrightarrow{p} \ell \quad \ell''' \xrightarrow{q} \ell \quad o_t \in pt(\ell'', p) \quad o' = \mathbf{init}(T(p), o_t)}{pt(\ell''', q) \subseteq pt(\ell, o')}$$

$$\mathbf{init}(t, o_{t'}) = \begin{cases} \boxed{o_{\tilde{t}}} & \text{if } t' \equiv \bullet \text{ [INITIALISE]} \\ o_{t'} & \text{if } \tilde{t} \prec t' \text{ [TBWU]} \\ \boxed{o_{\tilde{t}}} & \text{if } t' \prec \tilde{t} \wedge t' \not\equiv \tilde{t} \wedge h(o_{t'}) \text{ [TBSSU]} \\ \varnothing & \text{otherwise [TBSU]} \end{cases}$$

[BACK-PROPAGATE]
$$\frac{\ell : p = malloc_o \quad o_t \text{ newly cloned}}{o_t \in pt(\ell, p)}$$

[SU/WU]
$$\frac{\ell : *p = \_ \quad \ell' \xrightarrow{o} \ell \quad o \in \mathcal{A} \setminus \mathbf{kill}(\ell, p)}{pt(\ell', o) \subseteq pt(\ell, o)}$$

$$\mathbf{kill}(\ell, p) = \begin{cases} \{o'\} & \text{if } pt(\ell, p) \equiv \{o'\} \wedge o' \in \mathit{singletons} \\ \mathcal{A} & \text{if } pt(\ell, p) \equiv \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

[FIELD]
$$\frac{\ell : p = \&q \rightarrow f_k \quad \ell' \xrightarrow{q} \ell \quad o \in pt(\ell', q) \quad o' = \mathbf{init}(T(q), o) \quad t = \text{type of } t{::}f_k}{\boxed{(o'.f_k)_t} \in pt(\ell, p)}$$

[FF-NOT-IN-PT]
$$\frac{o \nsim o' \quad o \in pt(\ell, v)}{o' \in pt(\ell, v)}$$

[FF-EQ-PT]
$$\frac{o' \nsim o'' \quad o \in pt(\ell, o')}{o \in pt(\ell, o'')}$$

[PHI]
$$\frac{\ell : p = \phi(q, r) \quad \ell' \xrightarrow{q} \ell \quad \ell'' \xrightarrow{r} \ell}{pt(\ell', q) \cup pt(\ell'', r) \subseteq pt(\ell, p)}$$

[CAST]
$$\frac{\ell : p = (t)\ q \quad \ell' \xrightarrow{q} \ell \quad o \in pt(\ell', q)}{o \in pt(\ell, p)}$$

[CALL]
$$\frac{\ell : \_ = q(\dots, r, \dots) \quad \boldsymbol{\mu}(o) \quad \ell' : fun(\dots, r', \dots) \quad o{=}\boldsymbol{\chi}(o) \quad o_{fun} \in pt(\ell'', q) \quad \ell'' \xrightarrow{q} \ell \quad \ell^* \xrightarrow{r} \ell \quad \ell^\star \xrightarrow{o} \ell}{pt(\ell^*, r) \subseteq pt(\ell', r') \quad pt(\ell^\star, o) \subseteq pt(\ell', o)}$$

[RET]
$$\frac{\ell : p = q(\dots) \quad o{=}\boldsymbol{\chi}(o) \quad \ell' : ret_{fun}\ p' \quad \boldsymbol{\mu}(o) \quad o_{fun} \in pt(\ell'', q) \quad \ell'' \xrightarrow{q} \ell \quad \ell^* \xrightarrow{p'} \ell \quad \ell^\star \xrightarrow{o} \ell}{pt(\ell^*, p') \subseteq pt(\ell, p) \quad pt(\ell^\star, o) \subseteq pt(\ell, o)}$$

$T(v)$	$: \mathcal{V} \mapsto \mathcal{T}$	v's type.
$pt(\ell, v)$	$: \mathcal{L} \times \mathcal{V} \mapsto 2^{\mathcal{A}}$	v's points-to set after ℓ.
$\ell \xrightarrow{v} \ell'$	$: \mathcal{L} \times \mathcal{V} \times \mathcal{L}$	v's value flow.
$h(o)$	$: \mathcal{A} \mapsto$ Bool	o is a heap object.

$\tilde{t}$	$: \mathcal{T} \mapsto \mathcal{T}$	The element (pointee) type of t.
$t \prec t'$	$: \mathcal{T} \times \mathcal{T}$	t' is a transitive subtype of t.
$\bullet$	$: \mathcal{T}$	The undefined type.

Figure 5 Inference rules for the base analysis. We use a box to indicate that a new cloned object is created if it does not already exist.

object is unknown and will be determined later through usage of the object. Thus an untyped object $o_\bullet$ is propagated. The types of stack and global objects, however, are known, so a type (t in o_t) is immediately assigned.

4.1.2 Direct and indirect propagation (`[PHI] [CAST] [CALL] [RET]`)

Rules `[PHI]` and `[CAST]` act as copies, performing trivial direct propagation. Both simply propagate the points-to information of the pointer on the right hand side of an assignment to the pointer on the left hand side. In the case of rule `[PHI]`, the points-to sets of q and r are added to that of p. In the case of rule `[CAST]`, the points-to set of q is added to that of p as a CAST instruction acts like a copy. Despite the type-based nature of our analysis type casting has no effect on actual points-to relations since any pointer can point to any object except when that pointer is used in certain ways (recall **R2** of our type model).

Direct inter-procedural points-to propagation is done via the `[CALL]` and `[RET]` rules. Function targets are resolved on-the-fly during points-to analysis to more precisely discover callee functions at indirect callsites. Points-to values from the actual arguments at the callsite are then propagated to the corresponding formal parameters of each callee. The points-to information of an address-taken variable o is propagated via indirect value-flows (Section 3.2) with the χ and μ annotations shown in the two rules.

Indirect propagation is the propagation of address-taken variables (objects in $\mathcal{A}$) in the VFG. Indirect edges are labelled with address-taken variables, as determined by the pre-analysis, which are then propagated along those edges. Since the Andersen's analysis used to construct the VFG has no notion of typed objects, the edges are labelled with objects according to the allocation-site-based heap model (i.e. the original untyped objects). To remedy this, on the fly, when an indirect edge is labelled o, we propagate points-to information for all clones of o defined at the source of the indirect edge through the indirect propagation of o. For example, the propagation of the points-to information of both o_t and $o_{t'}$ from ℓ and ℓ' would be through the indirect edge labelled with o, i.e. $\ell \xrightarrow{o} \ell'$. Another way to resolve this is to augment the pre-analysis with type-based heap cloning and then indirect edges would be labelled with the appropriate typed objects. We forego this method for brevity and generality.

4.1.3 Loads and stores (`[LOAD] [STORE] [SU/WU]`)

LOAD and STORE instructions, through the `[LOAD]` and `[STORE]` rules, are handled in the same way as in a standard sparse flow-sensitive analysis [20] except that object initialisation (via **init**, described in Sections 4.1.4 and 4.1.5) is performed on the objects pointed to by the dereferenced pointers. The return value of the **init** function is then operated on rather than the object in the dereferenced pointer's points-to set (which was passed in to **init**).

The `[SU/WU]` rule performs standard singleton-based strong and weak updates [20, 30] for object o, pointed to by p, at STORE instruction $*p = q$. A weak update merges the points-to set of o with that of q and propagates that result onward. When o is a singleton, a strong update is performed. Strong updates discard o's old pointees, making its points-to set equivalent to that of q [30]. Singleton-based strong updates cannot take place upon local variables within recursion, arrays (treated monolithically), and heap objects. In addition to these singleton-based strong updates, TYPECLONE performs type-based semi-strong updates (Section 4.1.5.2) and type-based strong updates (Section 4.1.5.3) by leveraging our typing of abstract objects.

```
1  class S { ... };
2  class T : public S { int f; };
3  S *smalloc(size_t size) {
4    return (S*)malloc(size);
5  }
6  int main(void) {
7    T *t = (T*)smalloc(sizeof(T));
8    t->f = ...;
9  }
```

Figure 6 An example where object cloning would be performed by `cclyzer-ss` but not by TypeClone.

4.1.4 Object cloning

Object initialisation occurs whenever pointer access to an object makes an assumption regarding that object's type. The Load, Store, and Field instructions make assumptions about the type of the object being accessed. The **init** function used by rules `[LOAD]`, `[STORE]`, and `[FIELD]` handles object cloning through four different cases: `[INITIALISE]`, `[TBWU]`, `[TBSSU]`, and `[TBSU]`. It takes two arguments: the type t of the pointer pointing to the object of interest and the object $o_{t'}$ being accessed by the pointer. **init** may produce new objects if they do not already exist (i.e., it may clone objects). The potential to create a new object is denoted by a $\boxed{\text{box}}$ and the $\tilde{}$ operator, as in $\tilde{t}$, returns the element type of a pointer such that it would return `int *` when applied to `int **`, for example.

4.1.4.1 Object initialisation (`[INITIALISE]`)

In the `[INITIALISE]` case, an untyped object ($t' = \bullet$) is accessed by a pointer of type t. TypeClone will initialise the type of the heap object to be $\tilde{t}$ based on the assumption that the underlying type of the object is of type $\tilde{t}$ or a subtype of $\tilde{t}$. We can then propagate the $\tilde{t}$ typed object and stop propagating the untyped object thus differentiating it from objects of different types originating at the same allocation site. In C and C++, for example, code snippet `int *i = (int *)malloc(4); *i = 1;` makes an assumption about the type of the object returned by `malloc`–that it is of type `int` or a subtype of `int`–because pointer `i` cannot be accessing an object of any other type per our model.

Our approach to object cloning differs from that of `cclyzer-ss` in that it is less eager. When a pointer is cast to a pointer to another type, but never dereferenced as that pointer type, our approach would not perform object cloning whereas `cclyzer-ss` would. For example, consider Figure 6 where `T` is a derivative type of `S`. Despite a pointer to the allocated object o being cast to type `S *`, it is never dereferenced as such, and so we do not need to clone to create $o_{\texttt{S}}$.

4.1.4.2 Back propagation (`[BACK-PROPAGATE]`)

In the `[INITIALISE]` case, a new object is created and solely returned by **init**, causing the caller to stop propagating the original object, and to only propagate the clone. This ignores aliases made before the object was initialised and assigned a type. Figure 7 exemplifies this where pointer `a` is assigned pointer `i` before the pointed-to heap object is initialised at line 3. At lines 1 and 2, `i` would point to the untyped object $o_\bullet$, `a` would also point to $o_\bullet$ at lines 2 and 3, and `i` would correctly point (only) to the typed object $o_{\texttt{int}}$ at line 3 but would simultaneously share an incorrect no-alias relation with `a`.

```
1  int *i = (int *)malloc(...);
2  void *a = i;
3  *i = 1;
```

Figure 7 An example of an alias being made before initialisation occurs.

The [BACK-PROPAGATE] rule ensures that pre-initialisation aliases, like a in Figure 7, also point to the clone by back-propagating the newly created clone to the original object's allocation site, like [6, 7]. This is a cause of imprecision, since more than just pre-initialisation aliases will now point to the clone. Some precision is then recouped by subsequent typed-based strong and semi-strong updates.

4.1.5 Type-based weak, semi-strong, and strong updates

The **init** function acts upon types, hence we say it performs type-based updates. These updates are divided into type-based weak updates, type-based semi-strong updates, and type-based strong updates.

4.1.5.1 Type-based weak updates ([TBWU])

The [TBWU] case represents the basic case, *type-based weak updates*, or TBWUs. In this case, either the object's type exactly matches the pointer's element type ($\tilde{t} \equiv t'$), or the object's type is a derived type of the pointer's element type (an upcast took place). Both situations are covered by the statement $\tilde{t} \prec t'$. Since this access asserts the legality of such a pointer accessing such an object and makes no new assumptions about the object's type, it is simply propagated onward like a standard flow-sensitive analysis would and no cloning occurs.

4.1.5.2 Type-based semi-strong updates ([TBSSU])

The first case, [INITIALISE], results in a *type-based semi-strong update*, or TBSSU. It is type-based since the mechanism by which it occurs relies on type information, and it is semi-strong in that it kills one object and replaces it with another.

The [TBSSU] case, which models access after a downcast occurs, also results in a type-based semi-strong update. TypeClone assumes that any pointer access resulting from a downcast (i.e., $p = (t)\ q$ where $T(p) \prec T(q)$) is legal since we consider all input programs to conform to the strict aliasing rules of C and C++ (as implied by our type model). Regardless of whether an analysis accepts illegal programs or not, this is also the more conservative way of handling access after a downcast (with respect to soundness). From another point of view, we cannot know if the original type we assigned is the actual type of the object, and that the original type initialisation was actually an access through an upcast. We test that $o_{t'}$ is a heap object ($h(o_{t'})$) because non-heap objects have a declared type and cannot be changed (until we discuss reuse in Section 4.2).

Like the [INITIALISATION] case, we create a new object of type $\tilde{t}$ since an assumption about the type of the object is being made (that its real type is $\tilde{t}$ or a derivative of $\tilde{t}$, both of which being derived types of t'). It is tempting to change the object's type instead of cloning but this can cause unsoundness as the abstract object $o_{t'}$ may have been representing both concrete objects of type t' and concrete objects of type $\tilde{t}$ or other derivatives.

Handling downcasts explicitly gives a more accurate representation of an object's type which is necessary for virtual method resolution, and allows for better strong updates. To illustrate the latter, consider the C code in Figure 8, which implements a rudimentary form of

```
1  typedef struct { int i; } S;
2  typedef struct { struct S s; long l; } T1;
3  typedef struct { struct S s; float f; } T2;
4  void *smalloc(size_t size) {
5    S *base = (S *)malloc(size);
6    base->i = 1;
7    return base;
8  }
9  int main(void) {
10   T1 *p = (T1 *)smalloc(sizeof(T1));
11   p->l = 2;
12   T2 *q = (T2 *)tmalloc(sizeof(T2));
13   q->f = 3.0;
14 }
```

Figure 8 An example of an abstract object being initialised as two different "derivative" types, `T1` and `T2`, after it has been initialised as the "base" type `S`.

inheritance. Within wrapper function `smalloc`, all allocated objects are initialised to "base" type `S`. Callers of `smalloc` can then access the returned object as a "derivative" type `T1` or `T2`. If the programmer allocates the correct size, then this is legal since $\texttt{S} \prec \texttt{T1/T2}$. Since we see this as a downcast, the initialisation at line 11, for pointer `p`, would create a new object with type `T1`, stop propagating the object of type `S`, and back-propagate the new object. This would similarly occur at line 13 for pointer `q`. `p` and `q` at lines 11 and 13 would then *not* alias since they would perform a type-based strong update (discussed in the following section) on the back-propagated object which does not match their type, and neither would point to the `S` typed object any longer from the type-based semi-strong update to an object of type `T1/T2`. In essence, we have split an abstract object into more abstract objects, each of which representing a smaller set of concrete objects than the original abstract object.

4.1.5.3 Type-based strong updates (`[TBSU]`)

When an object is typed and there is no relation between the object's type and the pointer's element type, we know that the pointer is pointing to an object which would be impossible during execution (i.e., a spurious object). A conforming program cannot, for example, read an object through an unrelated pointer. In the `[TBSU]` case, we return nothing, which is, in effect, a strong update; the pointer will not regard the killed object as in its points-to set. A *type-based strong update* (TBSU) differs from a typical strong update in that it applies to any potential initialisation points, uses type information to perform it, and can occur to all forms of abstract objects.

4.1.6 Field-sensitivity (`[FIELD]` `[FF-NOT-IN-PT]` `[FF-EQ-PT]`)

Taking the address of a field of an aggregate object is handled by the `[FIELD]` rule. The `[FIELD]` rule is the same as that in a standard flow-sensitive analysis, except that (1) initialisation is performed on the objects which q points to since assumptions about the aggregate objects in question are being made, and (2) the type of the new field object is assigned by looking up the aggregate type of the object having its field taken. The `[TBSU]` case occurring on pointees of q has a similar effect to the filtering that `cclyzer-ss` performs where no field object is created for a spurious aggregate object [6, 7].

```
1 int s;
2 int *i = &s;
3 *i = 1;
4 float *f = new(i) float{2.0};
5 *f = 2.0;
6 // It is now undefined behaviour to load i.
```

Figure 9 An example of object reuse.

Though not represented in the rules (for simplicity), rather than further deriving a field object from a field object, we add the field index to the existing field object. For example, rather than deriving field object $o.f_k.f_j$, we would derive $o.f_{k+j}$. Only dealing with field objects derived from aggregate objects makes reasoning about the analysis easier.

Struct objects in C and standard-layout objects in C++ share the same memory address as that of their first field. Our analysis must ensure there is an equivalence between the points-to sets of such objects and their first field. A non-standard-layout object in C++ does not have to alias its first programmer-defined field in C++. This is often the case in practice, as in Clang and GCC, due to the implementation placing a virtual table pointer at the start of some objects for example. However, for the purpose of a pointer analysis, the first field of an object is tracked, regardless of whether it is programmer-defined or not, so this "first-field aliasing" needs to be applied to all struct and class objects.

We follow [6] where changes involving an object or the first field of an object trigger changes in the other through rules `[FF-NOT-IN-PT]` and `[FF-EQ-PT]`. They both use the first-field alias relation, defined as follows, similar to [6],

▶ **Definition 1.** *First-field alias relation. The first-field alias relation is defined as the equivalence relation* $\dot{\sim} : \mathcal{A} \times \mathcal{A}$ *such that* $o, o' \in \mathcal{A}, o \dot{\sim} o'$ *if and only if:*

$$o' = o.f_0$$

The `[FF-NOT-IN-PT]` rule ensures that when an object o belongs to some points-to set, then so does its first field $o.f_0$, if it exists, and vice versa. The `[FF-EQ-PT]` rule ensures that when an object o is in the points-to set of an object o', then it is also in the points-to set of its first-field $o'.f_0$, and vice versa. This keeps both points-to sets equivalent.

4.2 Object reuse

The base analysis presents a simple overview of our approach. This section extends our base analysis by considering object reuse, a special language feature in C and C++, to make the analysis handle programs making use of this feature. In C, writing to a heap object through a pointer of type $t*$ changes that object's type to type t; reads through a pointer of type $t*$ are now legal, and reads through pointers of type $t'*$ where $t' \not\prec t$ result in undefined behaviour. This would be only permitted if t fits in the space allocated for the object. Furthermore, in C, the type of an object (referred to as the "effective type" in the C standard) can be changed through functions `memcpy` and `memmove` or by being copied as a character array, thus not requiring a store through a pointer to the new type. Reuse is also possible in C++ with the addition that this may be achieved through placement `new` and that placement `new` can be used on stack and global objects. Though necessary to achieve better soundness, we have excluded reuse from the base analysis as it introduces a

```
1  char *pool;
2  void *palloc(size_t s) {
3    // Find index appropriate for s...
4    return pool + n;
5  }
6  void *pfree(char *m) {
7    // Return m to pool...
8  }
9  int main(void) {
10   pool = malloc(512);
11   int *i = palloc(sizeof(int));
12   *i = 1;
13   pfree(i);
14   float *f = palloc(sizeof(float));
15   *f = 2.0;
16 }
```

Figure 10 An example of pool allocator.

performance penalty, slight imprecision, is not often used in many C/C++ programs, and can be error-prone outside common patterns like pool allocators (which our analysis can handle when the pool is from the heap). An example of reuse is shown in Figure 9.

The form of reuse shown in Figure 9 cannot generally be soundly handled by our base analysis. A pool allocator, like in Figure 10, on the other hand, can be handled soundly by our base analysis, despite it relying on reuse. In Figure 10 the memory given to `f` can be the same as that given to `i`. The difference from Figure 9 is that the memory object is assigned from the main pool, similar to how an object may flow from any allocation wrapper (around `malloc`). The untyped object from the pool would then be initialised. The base analysis cannot handle pool allocators where the pool is a stack or global object since such an object would not be untyped (and thus cannot be initialised).

The incompatibility between reuse and the base analysis stems from the **init** function. Typically, reuse would not be possible through a Load, nor a Field not being stored to. However, if we limit reuse to stores, the analysis may be unsound since the case of changing the type of an object through `memcpy`/`memmove` or copying as an `char` array would need to be specially handled. Thus, we allow for reuse at Load, Store, and Field instructions. The **init** functions is changed to that in Figure 11.

The new case in the **init** function, `[REUSE]`, checks for an incompatibility between the object's type and the pointer's element type. If the types are incompatible, a clone is returned with the new type, that is, the object is being reused with the new type. This is a semi-strong update similar to the `[TBSSU]` and `[INITIALISATION]` cases since the previous object is no longer propagated and another is instead. We also remove the $h(o_{t'})$ condition from the `[TBSSU]` case to conservatively implement C++'s allowance of reuse of stack and global objects, as a stack or global object may be accessed through a pointer to a derived type (i.e., through downcasting) with placement `new`. It is not necessary to specially handle placement `new` since the object will eventually be written to as the new type, or read from as such. The `[TBSU]` will no longer be triggered as all cases are now covered with the introduction of the `[REUSE]` case. Fortunately, the `[REUSE]` case is a TBSSU rather than a TBWU.

$$\mathbf{init}(t, o_{t'}) = \begin{cases} \boxed{o_{\tilde{t}}} & \text{if } t' \equiv \bullet \texttt{[INITIALISE]} \\ o_{t'} & \text{if } \tilde{t} \prec t' \texttt{[TBWU]} \\ \boxed{o_{\tilde{t}}} & \text{if } t' \prec \tilde{t} \wedge t' \neq \tilde{t} \texttt{[TBSSU]} \\ \boxed{o_{\tilde{t}}} & \text{if } t' \not\prec \tilde{t} \wedge \tilde{t} \not\prec t' \texttt{[REUSE]} \\ \varnothing & \text{otherwise} \texttt{[TBSU]} \end{cases}$$

Figure 11 Modifications to the **init** function to account for reuse.

$$\texttt{[BACK-PROPAGATE-SG]}\quad \frac{\ell : p = \&o \quad o_t \text{ newly cloned}}{o_t \in pt(\ell, p)}$$

$$\texttt{[BACK-PROPAGATE-FIELD]}\quad \frac{\ell : p = \&q \rightarrow f_k \quad \ell' \xrightarrow{q} \ell \quad o \in pt(\ell', q) \quad o' = \mathbf{init}(T(q), o) \quad (o'.f_k)_t \text{ newly cloned}}{(o'.f_k)_t \in pt(\ell, p)}$$

Figure 12 Extensions to the analysis to implement back-propagation for stack, global, and field objects.

In the base analysis, back-propagation was only done for heap objects because stack and global objects had one unchanging type (hence they are never cloned). Since C++ allows for reuse of stack and global objects, we need to back-propagate them when they are cloned. `[BACK-PROPAGATE-SG]` implements this in Figure 12 like the `[BACK-PROPAGATE]` rule.

Furthermore, we need to account for reuse of field objects. Back-propagation for field objects is less obvious since field objects can be generated at multiple locations depending on the solver's worklist order. Since fields do not have an allocation site to back-propagate to, field clones are retrieved at any Field instruction which had retrieved the original field object, as in the `[BACK-PROPAGATE-FIELD]` rule. `[BACK-PROPAGATE-FIELD]` implements this as a second `[FIELD]` rule operating solely on the clones. Figure 12 also shows this rule.

4.3 Soundness and the heap cloning upper bound

For a C or C++ program conforming to our type model, our analysis is as sound as Sparse. To soundly analyse programs conforming to our type model, object reuse must be enforced even though doing so incurs a performance and precision penalty. The typical allocation-site-based model bounds the number of objects of a program by the number of allocation sites. Context-sensitive analyses bound the number of context-sensitive heap objects by the number of calling contexts [37]. For real-world scenarios, this is too large, so the context depth is often limited by a small number to make analyses scalable. When the maximum calling context depth is capped at 3 (or more), context-sensitive analysis is usually unscalable for larger programs [37]. The number of heap objects in our analysis is bounded by the number of allocation sites, the number of types on the generated type graph, and the number of fields in the largest structure type. Thus, in the worst case, the number of objects in our analysis would be the product of those three values, which would usually be far fewer than the number of objects created when cloning according to calling contexts.

Table 2 Statistics about the benchmarks. The first column of data represents the lines of code, the second column represents the size of the compiled program's bitcode, the fourth, fifth, and sixth columns represent the number of different instructions in the bitcode with the number of those instructions which are annotated by `ctir` in parentheses, the seventh column represents the number of canonical types with the number of those which are structs in parentheses, and the final column shows the number of fields in the largest struct in the program.

Bench.	LOC	Size	Instructions (`ctir` annotated)			# Canon. types (structs)	Largest struct
			Loads	Stores	GEPs		
du	22212	1372 KiB	14742 (2879)	5781 (907)	5384 (4928)	565 (79)	37
date	10002	1132 KiB	12185 (4430)	2860 (912)	7395 (6925)	182 (21)	30
touch	9820	1056 KiB	11416 (4392)	2502 (907)	7304 (6878)	178 (20)	30
ptx	16247	1056 KiB	11787 (2362)	4395 (714)	4546 (4180)	339 (43)	31
csplit	14565	936 KiB	10609 (2020)	3930 (563)	3887 (3628)	347 (49)	31
expr	14070	912 KiB	10336 (2028)	3807 (544)	4000 (3728)	315 (38)	31
tac	13888	876 KiB	10067 (1908)	3678 (491)	3710 (3457)	295 (34)	31
nl	13420	868 KiB	9960 (1924)	3629 (500)	3678 (3469)	293 (34)	31
mv	15962	844 KiB	7713 (1291)	3500 (544)	2437 (2136)	454 (59)	39
ls	14471	804 KiB	7050 (834)	3576 (334)	1655 (1233)	375 (50)	29
ginstall	14968	772 KiB	6843 (944)	3266 (376)	1800 (1521)	416 (53)	39
sort	12000	744 KiB	7743 (945)	3312 (422)	2262 (1802)	391 (58)	42

5 Evaluation

The aim of our evaluation is to compare the performance and precision (through alias testing) of TypeClone and Sparse. We first describe our implementation of TypeClone, and all required components, in Section 5.1 and then present the results of our experiments and discuss them in Section 5.2.

5.1 Implementation

Our implementation of the analysis is comprised of two major components: a custom Clang frontend to produce annotated LLVM IR with C/C++ type information and the TypeClone implementation built upon LLVM and SVF [40]. We use version 9.0 of both Clang and LLVM.

LLVM's type system is different to that of C/C++. In the LLVM IR produced, Clang does not maintain any type information from C/C++ except through TBAA metadata. Due to the basic nature of TBAA metadata, and that Clang does not annotate all instructions that we are interested in with TBAA metadata (GEP instructions, for example), we implement our own type metadata system called `ctir` which, like EffectiveSan's customised Clang [14], tags instructions of interest with DWARF debug information which can be read and operated upon by SVF.

During code generation, Clang introduces loads, stores, and other instructions which do not directly map to high-level code. This can, for example, be a byproduct of the nature of partial SSA form, or implementation-defined details like using virtual tables to implement virtual calls. Like TBAA, the instructions and declarations which correspond to C/C++ features of interest are annotated. In the absence of type information, our analysis falls back to standard flow-sensitive pointer analysis methods, not updating upon the type, i.e. not using **init**.

The following are annotated by `ctir`:

- Allocations corresponding to stack and global declarations (the allocated object's type).
- Load and store instructions which correspond to pointer dereferences and C++ reference accesses (the dereferenced pointer's element type).
- GEP instructions which correspond to field and array accesses (the base pointer's element type).
- Virtual calls (the base pointers's element type).
- Virtual tables (the owning class).

Since DWARF types correspond exactly to C/C++ types, in SVF, we reduce all types to "canonical types" which are types stripped of their signedness, `const`ness, `typedef`s, and other auxiliary data. The type graph is built from these canonical types and the analysis then only operates on canonical types (converting types obtained from `ctir` annotations as necessary).

A virtual call like $p \rightarrow foo()$ is translated into four LLVM instructions: (1) a Load instruction, $vtptr = *p$, which retrieves virtual table pointer $vtptr$ by dereferencing pointer p, (2) a Field instruction, $vfn = \&vtptr \rightarrow k$, which retrieves the entry (i.e., target function) in the virtual table at offset k, (3) a Load instruction, $fp = *vfn$, which retrieves the address of the target, and finally (4) a Call instruction, $fp(p)$. For calls to external functions where code is unavailable to analyse, a list of commonly used functions is maintained which summarise their side-effects (like `memcpy`, `_Znwm` for C++'s `new`, `mmap`, `strcpy`, and others) following [19, 35].

Within SVF, Andersen's analysis, optimised with *wave propagation* [34, 29] for better performance, is used to build the value-flow graph of the input program. Our analysis is then implemented on top of the built value-flow graph following the rules described in Section 4. We compare TypeClone (with and without reuse taken into consideration) with a sparse flow-sensitive and context-insensitive analysis (Sparse) [20] available in SVF [39]. To the best of our knowledge, this is the only publicly available implementation of a whole-program sparse flow-sensitive and context-insensitive C/C++ pointer analysis for LLVM. We also do not know of a publicly available implementation of a whole-program flow- and context-sensitive (FSCS) C/C++ pointer analysis for LLVM. According to a study using commercial tools [3], existing FSCS algorithms for C "do not scale even for an order of magnitude smaller size programs than those analyzed [with Andersen's analysis]" in their study. As shown in our evaluation, for annotated pointer accesses, TypeClone can achieve more precise results than Sparse, thus leaving limited room for benefit in this case by modelling the heap context-sensitively.

5.2 Experiments

We compare the performance and precision of our analysis, with and without reuse considered, with Sparse. We use the 12 largest programs, per LLVM bitcode size, in GNU Coreutils 8.31 (excluding `dir` and `vdir` since they are almost identical to `ls`). Coreutils was chosen because the included programs use various memory allocation wrappers to perform allocation. Table 2 shows the size (in LOC and of the generated bitcode), number of instructions, number of canonical types and how many of those are structs, and the largest struct by number of fields (after flattening) for each benchmark. All experiments were carried out on a machine running 64-bit Ubuntu 18.04.2 LTS with an Intel Xeon Gold 6132 processor at 2.60GHz and 128GB of memory.

To test the performance, we ran Sparse and TypeClone (without and with reuse) ten times and averaged the total running time of the analyses (constraint solving upon the VFG, excluding the pre-analyses to build the VFG and other auxiliary data structures like

Table 3 Running times and object counts of SPARSE and TYPECLONE (with and without reuse). The first column of data represents the running time of SPARSE and the second column represents the number of objects in the analysis. The third and fourth columns represent the running time of TYPECLONE (without reuse) and its slowdown from SPARSE, and the fifth column represents the total number of objects in the analysis with the number of clones created in parentheses. The same is repeated for TYPECLONE with reuse in the final 3 columns. The final row shows the geometric mean of slowdown.

Bench.	SPARSE		TYPECLONE			TYPECLONE (reuse)		
	Time	Obj.	Time	Diff.	Obj. (clones)	Time	Diff.	Obj. (clones)
du	15.83s	4295	92.81s	5.86×	5283 (991)	2829.37s	178.73×	7436 (3144)
date	0.34s	1924	1.02s	3.00×	2003 (80)	1.24s	3.65×	2051 (128)
touch	0.33s	1730	0.97s	2.94×	1817 (87)	1.20s	3.64×	1866 (136)
ptx	5.19s	3245	282.76s	54.48×	4242 (997)	378.30s	72.89×	4538 (1292)
csplit	3.45s	2885	4.98s	1.44×	3147 (263)	265.83s	77.05×	3919 (1035)
expr	2.17s	2750	50.40s	23.23×	3358 (609)	80.91s	37.29×	3603 (854)
tac	2.59s	2700	58.84s	22.72×	3383 (684)	78.63s	30.36×	3462 (763)
nl	2.93s	2663	101.69s	34.71×	3342 (680)	118.65s	40.49×	3424 (762)
mv	0.75s	3441	43.90s	58.53×	4403 (961)	67.04s	89.39×	4615 (1173)
ls	0.48s	2975	4.49s	9.35×	3278 (307)	4.58s	9.54×	3302 (331)
ginstall	0.30s	3332	1.75s	5.83×	3712 (378)	2.29s	7.63×	3779 (446)
sort	0.77s	2657	11.93s	15.49×	2994 (339)	12.36s	16.40×	3034 (379)
Average				11.14×			25.19×	

the type graph). The results are presented in Table 3. The "Diff." columns represent how many times slower an analysis was compared to SPARSE, and the "Obj." and "Obj. (clones)" columns represent the total number of objects in an analysis, with the number of clones created mentioned separately, where relevant. Generally, we expect running time to increase in TYPECLONE because of the introduction of new objects, which means larger points-to sets and thus extra propagation time, and back-propagation, which means VFG nodes are processed more often. For TYPECLONE without reuse, all slowdown presented is in a general range of acceptability of 1.45×–35× except for when analysing benchmarks `mv` and `ptx`. Both benchmarks created the largest number of clone objects relative to original objects. Overall, the slowdown is usually affordable and the (geometric) mean slowdown is a little over 11× when not considering reuse.

Modelling reuse in TYPECLONE slows down the analysis. Too many opportunities for TBSUs, which would reduce the number of objects created (and prevent some back-propagation) and reduce the size of points-to sets, become TBSSUs with the `[REUSE]` rule. Stack and global objects also become a source of clones. This is seen in the number of clones created. Benchmarks which were many times slower than TYPECLONE without reuse, like `du` and `csplit`, had many more clones created, and those that remained close in running time had a more modest growth in the number of extra clones created. We see a (geometric) mean slowdown of a little over 25× when considering reuse, more than twice as much compared to the base analysis.

To test the precision, we performed an alias query between all top level pointers of interest within a function (those pointers accessed at an instruction annotated with a C/C++ type with `ctir`) against each other. Two pointers are considered aliases if their points-to sets

Table 4 The number of alias queries performed (between `ctir`-annotated instructions), the number of those alias queries returning a no-alias relation for Sparse and TypeClone (with and without reuse considered), and the improvement to the number of alias queries returning a no-alias relation presented by TypeClone (with and without reuse considered) against Sparse.

Bench.	Queries	Sparse	TypeClone		TypeClone (reuse)	
		No-alias results	No-alias results	Improv.	No-alias results	Improv.
du	76291490	55553836	74384866	33.90%	72825202	31.09%
date	151400720	111391680	141377516	26.92%	138450386	24.29%
touch	149194010	109198398	139183292	27.46%	136253192	24.78%
ptx	52845630	43771886	50867888	16.21%	50655276	15.73%
csplit	38719506	30450964	37758260	24.00%	36824196	20.93%
expr	39835032	33654030	38228618	13.59%	38017650	12.97%
tac	34427556	27745100	32782666	18.16%	32654250	17.69%
nl	34863120	27895764	33204888	19.03%	33065830	18.53%
mv	15940056	12655806	14978588	18.35%	14894140	17.69%
ls	6167772	5242862	5869944	11.96%	5817110	10.95%
ginstall	8193906	7755314	7959676	2.64%	7920014	2.12%
sort	10198442	8189034	9583680	17.03%	9499016	16.00%
Average				16.64%		15.36%

intersect, or if one pointer contains a field object generated from an object in the other pointer's points-to set. A no-alias result is always more desirable than a may-alias result as it paves the way for more optimisations, for example. The number of alias queries, the number of queries returning a no-alias relation (the remainder return a may-alias relations), and the improvement TypeClone presents in the number of no-alias relations returned compared to Sparse are presented in Table 4. We find that for Sparse, in all benchmarks except `ginstall`, 72%–85% of alias queries return a no-alias result. `ginstall` is an outlier with almost 95% of alias queries returning a no-alias result. For TypeClone, 93% to over 97% of alias queries result in a no-alias relation (91% to over 96% when reuse is taken into account). Excluding `ginstall`, TypeClone (without considering reuse) sees an increase in almost 12% to almost 34% in the number of no-alias results against Sparse. The improvement for `ginstall` is much less at 2.64%. The results produced by Sparse for `ginstall` were already strong and TypeClone had little room to improve. Overall, the (geometric) mean improvement sits at over 16% when not taking reuse into consideration. When taking reuse into account, results are still strong albeit weaker than when not taking reuse into account. Excluding `ginstall` again, we see an increase in almost 11% to over 31% in the number of no-alias results against Sparse. For `ginstall`, the improvement is 2.12%, and the overall (geometric) mean improvement is over 15%.

Overall, TypeClone is successful at differentiating points-to sets when objects appear from the same allocation site, as is the case with allocation wrappers, which Coreutils makes use of. We also notice that the imprecision introduced by handling reuse is very slight. Even though handling object reuse eliminates many opportunities for TBSUs, instead creating more clones, the `[REUSE]` case in **init** is a TBSSU and thus does not cause as much precision loss as it would have if it was a TBWU, even if it may adversely affect performance.

6 Related work

Whole-program flow-sensitive pointer analysis for C and C++ has been studied extensively in the literature. The approaches in [8] and [15] provide the formulations for an iterative data-flow framework [25]. The work presented in [45] considered both flow- and context-sensitivity by representing procedure summaries with partial transfer functions. To eliminate unnecessary propagation of points-to information during the iterative data-flow analysis, sparse analysis propagates points-to facts sparsely across pre-computed def-use chains [20, 33]. Initially, sparsity was achieved through a Sparse Evaluation Graph [9, 21, 22], a refined CFG with irrelevant nodes removed. Further progress was made through various SSA forms like factored SSA [10], HSSA [11] and partial SSA [27]. The def-use chains of top-level pointers, once put in SSA form, can be explicitly and precisely identified, giving rise to a semi-sparse flow-sensitive analysis [19]. Then, by leveraging the idea of staged analyses [17, 20] where a fast, imprecise analysis bootstraps a more precise analysis, flow-sensitive analysis was made fully sparse, with the first stages identifying def-use chains of both top-level and address-taken pointers [20, 39]. Despite these achievements, most flow-sensitive analyses model the heap with one abstract object per allocation site. Most analyses which provide a more precise heap model do so by employing context-sensitivity.

On the other hand, structure-sensitive analysis (`cclyzer-ss`) [7] improves the allocation-site-based heap model and presents a field-sensitive Andersen's analysis that lazily infers the types of heap objects through the casting of pointers to those objects to eventually filter out redundant field derivations. When a pointer to a heap object is cast, that object is considered to potentially be of the element type of the pointer it is cast to and so a new object is created with the pointer's element type, and back-propagated to the allocation site to ensure soundness. Type-based alias analysis (TBAA) [13] uses Modula-3's type system to (almost) statelessly determine aliasing relations. TBAA is implemented in Clang/LLVM and GCC for C, C++, and Objective-C, and works because of the strict aliasing rules defined by those languages. Inspired by TBAA and `cclyzer-ss`, this paper proposes a new flow-sensitive type-based heap cloning model to improve the precision of sparse points-to analysis for C and C++ programs which conform to the strict aliasing rules.

7 Conclusion

This paper presents a new flow-sensitive points-to analysis with type-based heap cloning and no context-sensitivity. The novelty of our approach lies in its lazy heap cloning. An untyped abstract heap object created at an allocation site is killed and replaced with a new (clone) object uniquely identified by the type information at its use site for flow-sensitive points-to propagation. This yields more precise points-to relations at different program points without incurring the high costs of context-sensitivity. Our approach also explores a new form of strong updates based on types for flow-sensitive modelling. The resulting analysis improves upon state-of-the-art sparse flow-sensitive analysis answering, on average, over 15% more alias queries with a no-alias result.

References

1 ISO/IEC 14882:2017. Programming languages – C++. Standard, International Organization for Standardization, 2017.

2 ISO/IEC 9899:2018. Information technology – programming languages – C. Standard, International Organization for Standardization, 2018.

3 Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 746–755. IEEE, 2011. doi:10.1145/1985793.1985898.

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

5 Dzintars Avots, Michael Dalton, V Benjamin Livshits, and Monica S Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 332–341. ACM, 2005. doi:10.1145/1062455.1062520.

6 George Balatsouras. *Recovering Structural Information for Better Static Analysis*. PhD thesis, National and Kapodistrian University of Athens, 2017.

7 George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium (SAS)*, pages 84–104. Springer, 2016. doi:10.1007/978-3-662-53413-7_5.

8 Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 232–245. ACM, 1993. doi:10.1145/158511.158639.

9 Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 55–66. ACM, 1991. doi:10.1145/99583.99594.

10 Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, 1994. doi:10.1109/32.265631.

11 Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *International Conference on Compiler Construction (CC)*, pages 253–267. Springer, 1996. doi:10.1007/3-540-61053-7_66.

12 Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for Java with strong updates. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 665–687. Springer, 2012. doi:10.1007/978-3-642-31057-7_29.

13 Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, page 106–117. ACM, 1998. doi:10.1145/277650.277670.

14 Gregory J Duck and Roland HC Yap. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 181–195. ACM, 2018. doi:10.1145/3192366.3192388.

15 Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, 1994. doi:10.1145/773473.178264.

16 Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 329–340. ACM, 2017. doi:10.1145/3092703.3092729.

17 Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008. doi:10.1145/1348250.1348255.

18 Coreutils - GNU core utilities. URL: https://www.gnu.org/software/coreutils/.

19 Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, page 226–238. ACM, 2009. doi:10.1145/1594834.1480911.

20 Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO)*, pages 289–298. IEEE, 2011. doi:10.1109/CGO.2011.5764696.

21 Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999. doi:10.1145/325478.325519.

22 Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *International Static Analysis Symposium (SAS)*, pages 57–81. Springer, 1998. doi:10.1007/3-540-49727-7_4.

23 Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. HexType: Efficient detection of type confusion errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2373–2387. ACM, 2017. doi:10.1145/3133956.3134062.

24 Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):100, 2017. doi:10.1145/3133924.

25 John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977. doi:10.1007/BF00290339.

26 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 423–434. ACM, 2013. doi:10.1145/2499370.2462191.

27 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

28 Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *International Conference on Compiler Construction (CC)*, pages 287–304. Springer, 2005. doi:10.1007/11406921_22.

29 Yuxiang Lei and Yulei Sui. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *International Static Analysis Symposium (SAS)*, pages 27–47. Springer, 2019. doi:10.1007/978-3-030-32304-2_3.

30 Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–16. ACM, 2011. doi:10.1145/1926385.1926389.

31 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015. doi:10.1145/2644805.

32 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 (ESEC/FSE)*, pages 317–326. ACM, 2003. doi:10.1145/940071.940114.

33 Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 229–238. ACM, 2012. doi:10.1145/2345156.2254092.

34 Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization (CGO)*, pages 126–135. IEEE, 2009. doi:10.1109/CGO.2009.9.

35 Rajiv Ravindran Rick Hank, Loreena Lee. Implementing next generation points-to in Open64. In *Open64 Developers Forum*, 2010. URL: http://www.affinic.com/documents/open64workshop/2010/.

36 Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 693–706. ACM, 2018. doi:10.1145/3192366.3192418.

37 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 17–30. ACM, 2011. doi:10.1145/1925844.1926390.

38 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.22.

39 Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 460–473. ACM, 2016. doi:10.1145/2950290.2950296.

40 Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 265–266. ACM, 2016. doi:10.1145/2892208.2892235.

41 Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014. doi:10.1109/TSE.2014.2302311.

42 Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica Amsterdam, 1994.

43 Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, page 439–449. IEEE, 1981. doi:10.1109/TSE.1984.5010248.

44 John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144. ACM, 2004. doi:10.1145/996841.996859.

45 Robert P Wilson and Monica S Lam. Efficient context-sensitive pointer analysis for C programs. *ACM Sigplan Notices*, 30(6):1–12, 1995. doi:10.1145/223428.207111.