

# Improving Software Security with a C Pointer Analysis

Dzintars Avots

Michael Dalton

V. Benjamin Livshits

Monica S. Lam

Computer Science Department  
Stanford University  
Stanford, CA 94305

{dzin, mwdalton, livshits, lam}@cs.stanford.edu

## ABSTRACT

This paper presents a context-sensitive, inclusion-based, field-sensitive points-to analysis for C and uses the analysis to detect and prevent security vulnerabilities in programs. In addition to a conservative analysis, we propose an optimistic analysis that assumes a more restricted C semantics that reflects common C usage to increase the precision of the analysis.

This paper uses the proposed pointer alias analyses to infer the types of variables in C programs and shows that most C variables are used in a manner consistent with their declared types. We show that pointer analysis can be used to reduce the overhead of a dynamic string-buffer overflow detector by 30% to 100% among applications with significant overheads. Finally, using pointer analysis, we statically found six format string vulnerabilities in two of the 12 programs we analyzed.

## Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging;  
D.3.4 [Programming Languages]: Processors—*Compilers*;  
D.2.3 [Operating Systems]: Security and Protection

## General Terms

Algorithms, Programming languages, Software security, Vulnerabilities, Software errors.

## Keywords

Program analysis, context-sensitive, pointer analysis, type safety, error detection, software security, buffer overflows, dynamic analysis, security flaws, format string violations.

---

This material is based upon work supported by the National Science Foundation under Grant No. 0326227 and an NSF Graduate Student Fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.  
Copyright 2005 ACM 1-58113-963-2/05/0002 ...\$5.00.

## 1. INTRODUCTION

Software vulnerabilities in C programs have led to tremendous losses of productivity and have caused damages estimated in billions of dollars. These vulnerabilities often stem from the lack of type safety in the C language, with buffer overflows and format string exploits being two well-known security vulnerabilities.

Tools exist which audit programs for vulnerabilities; however, the unsafe nature of C makes it difficult to create precise software analysis tools. In particular, pointer alias analysis is an important component in any program auditing tool. The semantics of C can cause a sound pointer analysis to generate many false positive warnings by using unsound pointer analysis instead. The Metal compiler and the Intrinsa system, known to find numerous serious vulnerabilities in widely used operating systems, assume that pointers are unaliased until proven otherwise with a local analysis [4, 9]. The same assumption was used in a buffer overflow detection tool by Wagner et al. [18]. However, by making such unsound assumptions, these tools may not find all possible vulnerabilities and cannot be used to guarantee that a program is safe.

This paper explores the use of more advanced pointer alias analysis to create practical tools for improving software security. We present a *context-sensitive*, *inclusion-based*, and *field-sensitive* points-to analysis for C. We apply the analysis to (1) infer the type of variables in C based on their usage, (2) reduce the overhead of a dynamic bounds checker, and (3) find format string vulnerabilities.

### 1.1 Pointer Alias Analysis

Our analysis is based on a points-to algorithm developed originally for Java [20] using a binary-decision-diagram (BDD) representation. Our analysis is *context-sensitive*, meaning that calling contexts along different acyclic call paths have distinct points-to relations. It is *inclusion-based*, meaning that two pointers may point to overlapping but different sets of objects. It is also *field-sensitive*, meaning that separate fields in an object have distinct points-to relations. The BDD representation efficiently handles the exponential number of contexts by exploiting their similarities. It also accelerates inclusion-based analysis through efficient transitive closure computations [3, 20, 24]. Field-sensitive analyses are known to be much more precise than field-insensitive approaches. Our analysis, however, is flow-insensitive: the order of execution of program statements is not modeled.

Because of the semantics of the C language, pointer anal-

ysis for C is much more complex than that for Java. While Java enforces type safety by limiting user access to memory, C affords the user complete, direct control. A C programmer can take the address of any field within an object, treat any portion of memory as a contiguous region of bytes, perform arbitrary address arithmetic, and perform arbitrary object casts, whether between variant types of a union or in complete violation of declared type. Although many of these actions may violate the letter or spirit of the C99 standard, in practice most C compilers allow them. None of the proposed C pointer alias analyses are strictly sound because such an analysis would conclude that most locations in memory may point to any location in memory. In particular, they all assume that the relative placement of objects in memory is undefined. Most programs satisfy this assumption, which is part of the C99 standard, for the sake of portability. We propose in this paper a conservative pointer analysis, referred to as CONS, that is sound for all programs satisfying the above assumption.

Our conservative pointer analysis is likely to be imprecise because programs frequently conform to the C99 standard. By imposing additional assumptions on the C programs that reflect common usage, we can improve upon the pointer analysis results. We thus also propose an optimistic analysis, referred to as PCP (practical C pointers), that is not only sound for programs adhering to the C99 standard, but also for programs that may violate its type model. For example, we use structural equivalence to determine type compatibility, whereas C99 resorts to name equivalence for aggregates such as structures and unions.

Like the analysis for Java, we express our points-to analysis in Datalog, a logic programming language used in deductive databases [17]. We use the `bddbdb` tool to translate the Datalog rules into efficient BDD operations [20]. The resulting points-to relations can be used in subsequent Datalog programs, making it easy to develop further algorithms based on the results.

## 1.2 Type Inference

The declared types of C variables are hints indicating how the variables are likely to be used. Type inference determines the actual types of objects by analyzing the usage of those objects. This is of great interest, since it can show that a program is type safe and portable, or identify unsafe program operations that should be audited statically or checked dynamically.

We perform a type inference using the results of our pointer analysis. Using both the PCP and CONS models, we found very few instances where type declarations were violated. However, the CONS model consistently found more type declaration violations, as well as heap objects with multiple inferred types.

To check the results of our type inference, we instrumented our programs with a dynamic type checker, and with only two exceptions, found that there were no actual type conflicts. In the case that there were type conflicts, those operations did not propagate any pointer values. These results suggest that the PCP analysis is effective in finding more precise pointer alias results for these programs.

## 1.3 Optimized Dynamic Buffer Overflow Detection

Buffer overflows are the most common form of security

threat in software systems today, consistently dominating CERT advisories [5]. Despite years of effort on this topic, there is no practical static checker that can guarantee to find all buffer overflows automatically. The C semantics make it difficult to create a dynamic bounds checker that can ensure no buffer overflows occur without breaking existing programs.

CRED (C Range Error Detector) is a recently developed dynamic bounds checker that has been shown to work properly on over one million lines of C code [15]. Since buffer overflow exploits tend to be transported as user input strings, CRED can be run with less overhead by checking only string buffer overflows. Such an approach is shown to be effective against a testbed of 20 different buffer overflow attacks [21].

CRED is based on Jones and Kelly's concept of *referent objects* [11]. A pointer's *referent object* is the object that it is intended to reference. A pointer arithmetic operation produces a new address, but the result has the same referent object as the original pointer. When a pointer is dereferenced, it must point within the bounds of its referent object. CRED uses an *object table* to dynamically track the base and extent of each object. It instruments pointer arithmetic, dereferences, and string-manipulation functions such as `memcpy` to associate referent objects with pointer addresses and check that out-of-bounds pointers are not dereferenced.

Normally, CRED enters all referent objects into the object table. To reduce the overhead of the strings-only CRED system, we use our points-to results to inform CRED about objects which do not contain string-typed values. This optimization reduces the overhead of some of the benchmarks by 30% to 100%.

## 1.4 Format String Vulnerabilities

Another common security threat is the format string vulnerability exploit. A program may be exploited if it passes a user-supplied string as the format string argument to a system function of the `printf` family. The user-supplied string may contain format specifiers that can cause the program to write to unintended memory locations. The static detection of user-supplied format strings is an example of a *taint analysis*, and requires the results of a pointer analysis. We use our pointer analysis to identify strings containing data that may be supplied by the user, and to report instances of format string arguments which point to these tainted strings. We find actual format string vulnerabilities in two of our benchmarks.

## 1.5 Paper Organization

The rest of the paper is organized as follows: we first describe our C pointer alias analysis based on our PCP model in Section 2. Section 3 presents the CONS model. Section 4 discusses our type inference analysis. Section 5 presents our optimizations for the CRED dynamic bounds checker. Section 6 presents our algorithm to find format string vulnerabilities. We report our experimental results in Section 7. Finally, Sections 8 and 9 describe related work and conclude.

## 2. THE PCP POINTS-TO ANALYSIS

As discussed in Section 1, our CONS analysis assumes disjoint object spaces, as stated formally in Assumption 1. To increase precision, our PCP (practical C pointers) model makes several additional assumptions that reflect typical C

usage. This section justifies the assumptions in PCP and describes the points-to analysis algorithm.

**Assumption 1 (Disjoint object spaces)** *Each object is allocated in a separate memory space; a pointer to an object can only be derived from a pointer to the same object.*

## 2.1 Types and Fields

An object may be cast to more than one type during its lifetime, which requires us to consider whether a pointer deposited in a field under one type may be read from a field under another type. The PCP model represents fields as distinct and separate locations, unless it can be proven that they have the same physical location and the same physical layout. We will first discuss references to equivalent fields under identical physical layouts, followed by the use of casting to access physically overlapping union variants with different physical layouts.

### 2.1.1 Structural Equivalence

The C99 standard treats structure and union types as compatible if they are name equivalent. We use a more liberal model of type compatibility, *structural equivalence*. Under structural equivalence, any two types, regardless of naming, are considered type compatible as long as their physical layout is identical. This is similar to the notion of common initial sequences as specified in the C99 standard. Types that are structurally equivalent are considered to be interchangeable in our model. We also describe fields of two structures (or variants of a union type) as structurally equivalent if they have the same offset and have structurally equivalent field types.

We model an object as having separate locations for each structurally distinct field. Data stored in one field of an object is accessible through any structurally equivalent field in that object. A formal definition of structural equivalence is provided through structural induction:

- *Scalar types.* Primitive types, such as integers, pointers, `chars`, `floats`, `doubles`, are structurally equivalent if and only if they have the same sizes. A scalar object can be thought of as a field, since it is structurally equivalent to a leading field of a structure with a structurally equivalent field type.
- *Structures and union types.* The  $i$ th fields of two structures (or variants of a union type) are considered to be structurally equivalent if the types of the first  $i - 1$  fields are structurally equivalent and the declared types of the  $i$ th fields are also structurally equivalent. Fields that are structural equivalent are referred to having a *common initial sequence* [23]. Two structures are structurally equivalent if they have the same number of fields and all their fields are structurally equivalent.

We normalize C programs by renaming the types and fields such that they have the same names if and only if they are structurally equivalent. This enables us to determine if two fields are structurally equivalent by a simple inspection of their field paths.

### 2.1.2 Dynamic Variant Types

Objects, most commonly unions, may be cast between multiple types throughout the execution of a C program. If

data stored into a field under one variant type is retrieved through a field under another variant type, the exact values read depend on the platform-specific physical layout. This behavior is not defined under the ANSI C standard, which says that data stored into an object under one variant type is not expected to be retrieved when the object is cast to another variant type.

If an object is cast to another structurally nonequivalent type, the PCP model assumes that reads from a field in one variant type will not be used to access data stored to a structurally nonequivalent field in the other variant type.

**Assumption 2 (Variant Types)** *Pointers written to one field can only be retrieved when accessed as fields that are structurally equivalent.*

#### Example 1

```
union U {int x; char c;} u;
u.c = 'c';
int i = u.x;
```

Our optimistic model will not recognize that the character 'c' in `u.c` will be accessed through `u.x`. □

## 2.2 Handling of Pointers

Unlike Java references, C pointers can point into the middle of an object. A pointer is modeled as a pair  $\langle o, p \rangle$ , where  $o$  is the object name, and  $p$  is a field path. Stack-allocated objects are given names of the variable; heap objects are named by the allocation sites. A pointer pointing to the base address of an object is known as a *base pointer*, and has an empty field path denoted as  $\epsilon$ .

The PCP model allows an object to be cast to any type, so a base pointer can be used as a pointer to any type. However, a pointer to a field in the middle of an object cannot be used as if it pointed to a type incompatible with the field type.

#### Example 2

```
struct A { struct B A1; int A2; int A3; } a;
struct B { void *B1; void *B2; } *ptr;

ptr = (B*)&a;           (1)
ptr->B2 = ptr;          (2)
ptr = (B*)&a.A2;        (3)
ptr->B2 = ptr;          (4)
```

The use of `ptr` in lines (1) and (2) illustrate a common use of type casts. The base address of `a` is also the base address of its first field `A1`, so it can be interpreted either as pointing to an object of type `A` or `B`. So, it is safe to perform the field offset `B2` to the pointer. On the other hand, when `ptr` points to `a.A2`, the only possible type `int` does not have any fields. The PCP model would simply ignore the statement to add the field offset `B2` to `ptr`. The conservative model would write the pointer into both fields `A2` and `A3`. □

**Assumption 3 (Field type safety)** *If a dereference or a field-access operation is applied to a non-base pointer, the referent field type must be structurally equivalent to the type assumed by the operation.*

Using type information to eliminate certain points-to relations is not new. Such type-filtering is commonly used in

type-safe languages [3, 20]. It helps in removing side effects due to spurious pointers which result from the imprecision of static analysis.

To model this restriction properly, we define the concept of a *legal field path*. Let  $t \in T$  be the set of types in the program. Each **struct**  $t$  has a set of canonical fields, denoted  $F(t)$ . The first field of **struct**  $t$  is known as the leading field. The type of field  $f$  is given by  $type(f)$ .

We define a *legal field path* to be a sequence of field names,  $f_1.f_2.\dots.f_n$ , such that  $f_1$  is a field of some structure  $t_1$  and  $\forall i > 1 : f_i \in F(type(f_{i-1}))$ . Note that adding a leading field to a path does not advance the path. We drop all the leading fields from the head and tail of a legal field path to derive a *canonical field path*.

**Example 3** In Example 2, paths  $\epsilon$ , **A1** and **A1.B1** all point to the base address of an object, whereas **B2** and **A1.B2** both point to the second field in structure **B**. The canonical path of the former is  $\epsilon$  and the latter is **B2**. The  $\epsilon$  path can be used as a pointer to any kind of structure, however, the latter field path can only be used as a pointer to a field of type **void\***.  $\square$

We say that  $cpath(p, l)$  is true if  $p$  is the canonical version of the legal path  $l$ . When a field access  $f$  is applied to a pointer with a canonical field path  $p$ , we check if  $p$  can point to a structure of which  $f$  is a field. Formally, we ask if there is a legal path  $l$  such that  $cpath(p, l)$  and  $cpath(p', l.f)$ .  $p'$  is the new canonical path representing the location. If  $l$  does not exist, then the field access cannot be performed in the PCP model.

In the PCP model, a pointer pointing to a non-union field in the middle of an object can point to only one primitive type. However, if a pointer points to a start address of a union type, it can correspond to the start of multiple variants with different types. We say that  $safecast(p, t)$  is true if a canonical path  $p$  points to a location of primitive type  $t$ . Formally, it is true if  $p$  is an empty path, or if  $p$  can be extended to form a legal path whose last field is declared to be of primitive type  $t$ .

### 2.3 Primitive Assignment Statements

We now describe the program representation which our analysis uses. Since our algorithm is flow-insensitive, we ignore all control flow statements. All remaining C statements are broken down into simple operations on primitive type variables. For example, an assignment to a field of a structure is decomposed into a field address calculation, followed by a store. Temporary variables are introduced as needed.

All program information relevant to the analysis is extracted and stored as relations in a program database. The database can be queried using Datalog, a logic query language [17]. We write Datalog rules to capture how we infer the points-to relations from each simple program operation. Relations are expressed as predicates in Datalog; for example, we say that  $A(w, x)$  is true if the tuple  $(w, x)$  is in relation  $A$ . A Datalog rule defines a predicate as a conjunction of other predicates. For example, the Datalog rule

$$D(w, z) :- A(w, x), B(x, y), C(y, z).$$

says that “ $D(w, z)$  is true if  $A(w, x)$ ,  $B(x, y)$ , and  $C(y, z)$  are all true.”

[alloc]	$points\text{-}to(\langle x, \epsilon, type(x) \rangle, \langle malloc_c, \epsilon \rangle) :-$ “ $c : x = malloc(y)$ ”
[address]	$points\text{-}to(\langle x, \epsilon, type(x) \rangle, \langle y, \epsilon \rangle) :-$ “ $x = \&y$ ”
[assign]	$assign(\langle x, \epsilon, type(x) \rangle, \langle y, \epsilon, type(y) \rangle) :-$ “ $x = y$ ”
[prop]	$points\text{-}to(\langle x, \epsilon, type(x) \rangle, \langle o, p \rangle) :-$ $assign(\langle x, \epsilon, type(x) \rangle, \langle y, \epsilon, type(y) \rangle),$ $points\text{-}to(\langle y, \epsilon, type(y) \rangle, \langle o, p \rangle)$
[load]	$assign(\langle x, \epsilon, type(x) \rangle, \langle o, p, t \rangle) :-$ “ $x = *(t*)y$ ”, $points\text{-}to(\langle y, \epsilon, type(y) \rangle, \langle o, p \rangle), safecast(p, t)$
[store]	$assign(\langle o, p, t \rangle, \langle y, \epsilon, type(y) \rangle) :-$ “ $*(t*)x = y$ ”, $points\text{-}to(\langle x, \epsilon, type(x) \rangle, \langle o, p \rangle), safecast(p, t)$
[field]	$points\text{-}to(\langle x, \epsilon, type(x) \rangle, \langle o, p_2 \rangle) :-$ “ $x = \&(((t*)y)\text{-}>f)$ ”, $points\text{-}to(\langle y, \epsilon, type(y) \rangle, \langle o, p_1 \rangle),$ $cpath(p_1, p'_1), cpath(p_2, p'_1.f)$

**Figure 1: Inference Rules for Simple Statements.**

Due to space constraints, we show only the inference rules, written in Datalog, for simple assignment statements (Figure 1). For the sake of clarity, all the input relations extracted from the program are simply presented as code in quotes. We say that  $points\text{-}to(\langle o_1, p_1, t \rangle, \langle o_2, p_2 \rangle)$  is true if the pointer  $\langle o_2, p_2 \rangle$  is stored as type  $t$  in the pointer location  $\langle o_1, p_1 \rangle$ . We say that  $assign(\langle o_1, p_1, t_1 \rangle, \langle o_2, p_2, t_2 \rangle)$  is true if the contents at the pointer address of  $\langle o_2, p_2 \rangle$  is retrieved as type  $t_2$  and stored as type  $t_1$  in the pointer location  $\langle o_1, p_1 \rangle$ .

[alloc]	$c : x = malloc(y);$ $x$ points to the object allocated, which is named by its allocation site $c$ .
[address]	$x = \&y;$ $x$ points to the starting address of $y$ .
[assign]	$x = y;$ Note that here we are dealing with primitive types, since aggregate assignments have been broken down to primitive assignments. This creates an <i>assign</i> relation, which says that the right-hand-side points to what the left-hand-side points to. Rule [prop] propagates the points-to targets from the source of the <i>assign</i> relation to the destination.
[load]	$x = *(t*)y;$ Check if $*y$ is allowed to be cast to type $t$ ; if so, then assign $x$ to what is pointed to by $y$ .
[store]	$*(t*) x = y;$ Check if $*x$ is allowed to be cast to type $t$ ; if so, then assign $y$ to the locations pointed to by $x$ .
[field]	$x = \&(((t*)y)\text{-}>f);$ Check if it is legal to extend the paths pointed to by $y$ with $f$ , if so assign the resulting canonical path to $x$ .

## 2.4 Arrays and Pointer Arithmetic

**Assumption 4 (Pointer arithmetic)** *Arithmetic is applied only to pointers pointing to an element of an array to compute another element in the same array.*

This assumption allows precise modeling of the common usage of pointer arithmetic. Likely potential violations include unintended buffer overflows and underflows, and casting of data structures into character arrays to get at the data’s low-level byte representation.

We model an array type as having two fields: the leading field  $f_1$  represents the first element and the second field  $f_2$  represents all remaining elements. Both fields are structurally equivalent to type  $t$ , the type of the array element. Since  $f_1$  is a leading field, any pointer to an array may also be used as a pointer to its element type. Furthermore, if two array types have the same element type, their respective  $f_2$  fields are considered structurally equivalent, regardless of the size of the array. However, the array types themselves are considered structurally equivalent only if they have the same sizes.

Arithmetic operations are allowed only on pointers to array elements. For each arithmetic operation, we determine if it may leave the pointer value the same, increment the pointer, or decrement the pointer. The former is modeled as a simple pointer assignment, the latter two are modeled as follows:

1. Incrementing a pointer to field  $f_1$  or  $f_2$  in an array returns a pointer to  $f_2$ .
2. Decrementing a pointer to field  $f_2$  in an array results in a pointer to both field  $f_2$  and  $f_1$ .

In both cases, the element type of the referent array must be structurally equivalent to the expected element type.

Some programs cast pointers to integers for the purposes of performing arithmetic. Expression of the form  $addr = (\text{int})baseptr + i * n$ , where  $n$  is a constant, are often used to find element  $i$  of an array with elements of bytesize  $n$ . We substitute equivalent pointer arithmetic expressions for every possible element type of that bytesize. If  $i$  does not have a constant multiplier we use `char` as the element type.

## 2.5 Memory Copy Routines

Routines such as `memcpy` and `strcpy` are often used to copy values from one memory region to another, by treating the source destination locations as arrays of bytes and copying a specified number of bytes from the source to the destination. It is difficult to model a copy routine accurately because no expected type information is provided and it is often not possible to determine the range of fields copied.

**Assumption 5 (Memcpy)** *We assume that the length of a memory copy operation only extends to the end of the largest field type located at the source address. For each type at the source address, we match a common initial sequence of fields with each type at the destination address. We precisely model the operation as an assignment between corresponding fields.*

## 2.6 Context-Sensitive Analysis

Our treatment of procedure invocations and context sensitivity is the same as the one used by Whaley and Lam [20].

Readers are referred to that paper for details. Here we shall only provide an overview of the algorithm for completeness.

To handle indirect function calls, the algorithm first computes the context-insensitive points-to relations while discovering call targets on the fly. The algorithm then uses the call graph computed to find the context-sensitive points-to relations.

The analysis starts by extracting all the input relations from the source program and store them as relations in the database. In the context-insensitive phase, assignment relations linking the actual with the formal parameters are generated on-the-fly as the function targets are discovered. The inference rules are applied until no more points-to relations are discovered.

Context sensitivity is handled with *cloning*. Cloning conceptually generates multiple instances of a function such that every distinct calling context invokes a different instance, thus preventing information from one context from flowing to another. Cloning makes generating context-sensitive results algorithmically trivial: the *context-insensitive* algorithm can be applied to the cloned program to obtain *context-sensitive* results. Note that the analysis does not clone the code per se; it simply produces a separate answer for each clone. We use entire call paths to distinguish between contexts in programs without recursion. To handle recursion, call paths are reduced by eliminating all invocations whose callers and callees belong to the same strongly connected component in the call graph. These reduced call paths are used to identify contexts.

A location in a stack-allocated object in the context-sensitive analysis is represented as  $\langle c, o, p \rangle$ , where  $c$  is the context number identifying the clone in the call graph,  $o$  is the local variable, and  $p$  represents the path taken to reach the location. Context-sensitive relations and inference rules are analogous to the context-insensitive ones, except that stack allocated objects are qualified with a context number.

The call graph computed in the first pass is used to create the clones for each function. Assignment statements are generated before the analysis begins, linking the actual arguments of each instance to the formal parameter of the matching instance of the callee function. We then compute the context-sensitive points-to relations by applying inference rules to the expanded call graph until no more new relations are generated.

## 3. CONSERVATIVE POINTS-TO MODEL

Our conservative model, `CONS`, includes the inference rules in the `PCP` model, as well as additional rules to model program behavior which falls outside assumptions of the `PCP` model. The points-to relations found by the `PCP` model are a subset of the relations found by the `CONS` model. `CONS`, assumes only that displacements between objects are undefined (Assumption 1):

1. Contrary to Assumption 2, any read from a field also retrieves values from any possibly overlapping field.
2. Contrary to Assumption 3, we might dereference a pointer of type  $*t$  pointing to path  $p$ , where there is no type  $t$  field. All fields at  $p$  are accessed by the dereference, as well as all fields following  $p$ , since the range of the dereference may span into those fields as well.
3. Contrary to Assumption 3, we may lookup field  $f_1$  in type  $t_1$  relative to path  $p$ , where there is no type

$t_1$  field. CONS will return the address of every field following  $p$  which can overlap with a hypothetical  $f_1$  field.

4. In the PCP model, arithmetic is applied only to pointers to an array element to derive pointers pointing to another element in the same array (Assumption 4). Unless it is proven otherwise, the CONS model assumes that a computed pointer may point to anywhere in the object pointed to by the source pointer.
5. PCP assumes that `memcpy` will only copy data up to the end of the largest field type pointed to by the source pointer, and only to structurally equivalent fields in the destination. (Assumption 5). CONS copies data beyond the end of the largest field, and the destination location does not need to be structurally equivalent.

## 4. TYPE INFERENCE

The declared type of a C variable may not reflect its actual type, since the address can be arbitrarily cast and dereferenced. However, we can infer the actual type of objects from their uses. Such information helps programmers understand the code, as conflicts between declared and inferred types flag dubious coding practice that can lead to obscure bugs. Also knowing the types of objects can be useful for code optimizations.

From the results of our pointer analysis, we can trivially determine which fields of an object are accessed. We can infer the type of an object by looking up the type or set of types which contain the accessed fields. Practically all objects that violate their declared types have multiple types. An object has multiple types unless all its accessed fields belong to the same type. This information can be easily gathered with a small number of Datalog rules.

## 5. OPTIMIZED CRED

The CRED bounds checker imposes two kinds of runtime overheads. The first occurs from maintaining the object table as stack and heap objects are allocated and deallocated. The second occurs as address computations and string manipulations are checked for OOB addresses. We optimize the first kind of overhead by only entering objects which can possibly be referenced by string operations.

The overhead of inserting objects into the table can be large. CRED inserts all dynamically created objects into the table, as well as all stack-allocated variables whose addresses have been taken in the code. Even though the strings-only version of CRED only checks for overflows in strings, CRED enters all addressed objects into the object table, for fear that non-string objects may be dynamically cast as strings.

We identify objects containing strings by finding the referent objects of operations that operate on strings with the results of our points-to analysis. Any object addressed by pointer  $p$  must be entered into the table if:

1.  $p$  is the base address in a `char*` pointer arithmetic expression (including those converted from `int` arithmetic expressions).
2.  $p$  is used as an argument to a string-manipulating external library function such as `strcat` or the `memcpy` family of routines.

Having identified all possible string-containing objects, we inform CRED so that it can omit the rest from the object table.

## 6. FORMAT STRING VULNERABILITIES

The detection of format string vulnerabilities requires a taint analysis to determine which object fields may contain data originally derived from user input. Perl provides a dynamic taint analysis, whereby user inputs and derived data are considered to be tainted. A dynamic analysis can only detect a problem given the right test inputs, and will incur a runtime overheads if used as a preventative measure. Static analysis, on the other hand, has the benefit of finding all potential vulnerabilities before the program is run.

We define *source objects* to be all objects directly under user control which originate taint. These include elements of the `argv` array, the return results of IO functions, and functions interacting with the environment such as `getenv`. We find source objects using parameter and return value references from selected system functions.

Every field in a source object is considered to be tainted. Taint is propagated to other object fields through direct and indirect assignments found by our pointer analysis. Objects referenced by the format string argument of a function such as `printf` are considered *sink objects*. A tainted sink object indicates a potential format string vulnerability.

## 7. EXPERIMENTAL RESULTS

We have implemented and run the described analyses on a suite of 12 benchmarks. In this section we discuss the results of our pointer analysis, static type inference query, dynamic type verification, dynamic string-buffer bounds checking, and static format string vulnerability query.

### 7.1 Experimental Setup

We have chosen a suite of 12 benchmarks, including commonly used Unix systems software, security-critical daemon programs, and standardized SPEC 2000 benchmarks. `ffingerd` is a finger daemon, `polymorph` is a filename conversion program, `bzip2` is a compression utility, `pcrc` is a regular expression library, `bftpd` is an FTP server, `gzip` is a compression utility, `mcf` is a scheduling program, `muh` is a network game, `monkey` is a Web server, `enscript` is a program for converting text to PostScript, `crafty` is a chess playing program, and `hypermail` is a program for converting mailbox files into HTML. Table 1 shows the versions of the programs analyzed, the number of lines of code in the program source as well as the number of lines after preprocessing with blank lines ignored. Runtime measurements were taken on an AMD Opteron 150 machine with 1GB of memory running Linux. We used the test cases provided; when test cases were not available, we created appropriately large test files composed of common inputs.

Our analysis begins by transforming a program into a collection of simplified operations and building a set of BDD relations. After compiling and linking using the SUIF 2 compiler framework [1], we number objects (distinguishing between SSA versions), normalize types and field names, and map legal field paths to canonical paths. We use the BuDDy BDD library to build BDD relations which represent program operations and path relationships. The time spent creating the initial program representation is reported in the fifth column of Table 1.

### 7.2 Pointer Analysis

Implementing the basic context-insensitive PCP analysis required 32 Datalog inference rules. In addition, we de-

Benchmark	Version	Line count	Preproc. lines	BDDs creation time	Solver time (seconds)			
					Context-Insensitive		Context-Sensitive	
					PCP	CONS	PCP	CONS
ffingerd	1.28	328	1,706	5.6	2.0	2.2	2.8	3.0
polymorph	0.4.0	582	3,984	7.1	3.4	2.3	3.9	3.5
bzip2	SPEC'00	3,923	4,609	9.0	0.6	1.3	3.3	4.2
pcre	3.9	6,875	8,441	14.8	2.8	10.4	10.0	37.0
bftpd	1.0.12	901	8,887	8.6	2.4	3.9	4.4	5.8
gzip	1.2.4	6,571	14,070	17.9	5.3	11.3	7.4	15.5
mcf	SPEC'00	1,511	19,993	8.9	1.2	4.1	3.9	9.3
muh	2.05d	4,264	30,427	8.9	2.8	5.0	9.2	10.0
monkey	0.8.4-2	3,982	34,027	20.1	7.2	11.6	11.1	28.4
enscript	1.6.1	27,724	58,003	53.1	9.9	26.4	26.6	136.3
crafty	SPEC'00	19,189	73,442	301.2	14.1	29.5	45.5	146.4
hypermail	2.1.5	29,912	93,606	48.2	48.1	170.7	185.0	735.7

**Table 1: Analysis times of the benchmark suite. In this and other tables in the paper, benchmarks are sorted by the preprocessed line count. Blank lines are not included in the line counts.**

finer 200 external system calls through the use of additional rules which model each system call as inlined program operations. Implementing a context-sensitive analysis required converting the existing rules to augment each occurrence of an object variable with an accompanying context variable, as well as augmenting the assignments of actual arguments to formal parameters with a proper context numbering. Implementing CONS analysis required adding an additional 26 rules to the base PCP versions. The PCP model requires fewer rules thanks to its simplistic assumptions. The CONS analysis must define additional rules for each case that the PCP model ignores. These rules create new relations which can propagate further, and increase the runtime of the CONS analysis. The choice of BDD variable domain ordering can significantly affect the performance of a Datalog program. We used the ordering automatically discovered by the `bddbdb` tool [20].

In context-insensitive PCP mode, most of the benchmarks take under 6 seconds to run, except for our largest 4 benchmarks, of which `hypermail` solves in 48.1 seconds. CONS modes, on average, takes about 2.5 times as long as PCP modes, with the worst case of `enscript` taking 5 times as long in context-sensitive CONS mode. As noted, CONS analyses may take longer due to an increased ruleset and additional iterations of rule applications as non-PCP relation tuples propagate. The context-sensitive analyses take, on average, almost 3 times as long as the context-insensitive analyses. This is a small slowdown in return for points-to results for an exponential number of calling contexts.

### 7.3 Static Type Checking

In Table 2 we present the results of our static type inference Datalog query. We have separately counted the number of stack and heap-allocated objects with multiple types, and have provided the total number of stack and heap objects for comparison. The results using PCP and CONS modes are listed in separate columns, and context-insensitive results are listed in parentheses where different from context-sensitive results.

In every benchmark, additional objects are found with multiple types in CONS mode than those found in PCP mode. However, the vast majority of stack objects have single types. This is also true for heap objects in PCP mode, but

Benchmark	Stack			Heap		
	Multi-type		Total	Multi-type		Total
	PCP	CONS		PCP	CONS	
ffingerd	0	7	226	0	0	0
polymorph	0	4	699	0	0	4
bzip2	1	7	1,035	2	2	10
pcre	0	6	995	5	9	19
bftpd	0	14(15)	941	0	1	5
gzip	2	22	2,813	1	3	7
mcf	1	1	2,214	0	4	4
muh	0	10	838	3	40	42
monkey	8	15	3,447	14(15)	21	25
enscript	26	69	2,320	11	23	27
crafty	0	77	10,121	0	11	12
hypermail	53(102)	105(136)	5,935	23	25	128

**Table 2: Number of multi-type objects found with context-sensitive pointer analysis. Context-insensitive results are shown in parenthesis where different.**

a large proportion of heap objects have multiple types in CONS mode. These results show that we can determine the actual type of most program objects. Only a small number require closer attention, which is more often in regards to user-allocated objects.

### 7.4 Dynamic Type Checking

We perform dynamic type checking to identify program behavior which violates the actual type of program objects. We have instrumented our benchmarks to verify that every object is treated as having a single type over its lifetime, and that every dereference, array element- and field- address calculation is consistent with its type. We also test that source and destination types match in routines such as `memcpy`. Dynamic checking can only show that a program is type safe for the given input, not for all possible runs.

We have implemented a dynamic type checker using the CRED buffer overflow detection framework. We augment the CRED object table to track each object’s type, which we represent as a linear sequence of bytes. Types are determined statically for local variables. The type of a global or dynamically allocated object is taken to be the type of its first dereference. This imprecision can result in false positive type violations, which we remove by hand.

Benchmark	Heap allocs	Stack allocs	Bounds checks	Non-strings	
				% Heap	% Stack
ffingerd	0	313,605	88,201	0.0	87.5
polymorph	0	1,464	96,904	0.0	0.0
bzip2	12	63,050	0	41.7	99.6
pcre	1,052,003	145,405,004	20,218,005	2.3	93.4
gzip	31,292	22,802	14,386	100.0 (0.0)	84.8
mcf	3	36	1	100.0	88.9
monkey	382	146,327	83,694	1.3 (0.0)	100.0
enscript	279,282	977,239	44,130,077	6.2 (0.0)	31.3
crafty	37	2,458,970	7,688,896	13.5	63.3
hypermail	36,065	19,620,748	145,968,123	0.0	56.3

**Table 3: Overhead operations introduced by strings-only CRED and percentage of non-string objects in programs found with PCP analysis. Numbers in parentheses are for CONS results, when different. Choice of context sensitivity does not affect the results.**

Benchmark	Base runtime (seconds)	Unopt. overhead (%)	Optimized overhead(%)	
			PCP	CONS
ffingerd	14.4	2	0	0
polymorph	28.2	-1	-2	-1
bzip2	13.6	-2	-1	-2
pcre	12.5	144	-1	-1
gzip	10.9	3	2	1
mcf	11.8	39	39	39
monkey	33.8	-1	0	0
enscript	9.1	19	19	20
crafty	1.7	27	12	12
hypermail	1.8	387	270	260

**Table 4: Reducing the runtime overhead of strings-only CRED by entering only strings in the object table.**

If an object’s type contains union-typed fields, we keep a list of possible variant types. We remove variant types from this list as we find them to be incompatible with subsequent operations. If the list of candidate union fields is exhausted, we report a type violation.

Out of 10 benchmarks (we were unable to test `bftpd` or `muh` using the CRED system), only two were found to contain objects treated with more than one type. This is an unusual event in portable programs, as it requires making assumptions about the size and alignments of C’s types. `hypermail` was found to cast a double to an array of characters in order to access it byte by byte. `bzip2` casts an array of 32-bit integers to an array of 16-bit shorts. In both of these cases, assumptions are violated to get at the low-level representation of numeric data. Fortunately, though these casts would violate the PCP model, code inspection showed that they were not involved in the propagation of pointer values.

## 7.5 Optimization of CRED

We have timed program runs using CRED, counted the number of instances of stack and heap objects entered and removed from the object table, and counted the number of checked operations performed. We compiled each program normally, then with CRED, and then with an optimized version of CRED for each variation of our pointer analysis. Optimized versions of CRED are provided with lists

of variable identifiers and allocation sites corresponding to non-string objects.

The first two columns of Table 3 display the number of object table insertions and deletions performed by CRED for stack and heap variables CRED does not insert unaddressed stack variables into the object table. The third column lists the number of bounds checks performed, for comparison with object table modifications. The next two columns list the percentage of CRED object table modifications pertaining to non-string objects under PCP and CONS.

For the most part, our PCP and CONS analyses identified the same objects as non-strings. However, in `gzip`, 100% of inserted heap objects were determined to be non-strings by our PCP analysis, while 0% were found to be non-strings by our CONS analysis. `enscript` showed a smaller disagreement in the number of non-string heap objects. Most stack objects were determined to be non-string objects, with proportions higher than 87% in 8 out of 10 benchmarks.

We were able to optimize the performance of CRED by preventing it from inserting non-string objects. Table 4 displays the normal runtime for each program, the percentage overhead due to CRED, and the percentage overhead due to each optimized version of CRED. We find that `ffingerd`, `bzip2`, `polymorph`, `gzip`, and `monkey` have a negligible overhead. At the other extreme, `hypermail`’s overhead is over 3 times the base runtime, and the overhead for `pcre` is about 1.5 times the base runtime. The other benchmarks have around a 30% overhead. Our optimizations caused a complete reduction in overhead for `pcre`. `crafty`’s overhead is reduced by more than half, and `hypermail`’s overhead is reduced by a third.

We also investigated how well these optimizations combined with alternative optimization techniques. After examining the most frequently fired checks in `hypermail` and `enscript`, we manually optimized the heavily executed portions of the programs using two strategies. We first eliminated redundant bounds checks dominated by earlier checks. Often, a pointer would be dereferenced twice or more in an inner loop, latter dereferences were removed if we ascertained that they were redundant. Comparisons in test conditions were also a common source of redundant checks.

For loops with monotonically increasing or decreasing index variables, we performed code motion to move checks outside the loop. This way, a check fired multiple times was replaced with a single check for an entire range of indices.

Benchmark	Source calls	Tainted objects		Sink calls	Sink objects	Vulnerabilities	
		PCP	CONS			PCP	CONS
<code>ffingerd</code>	3	23	23	15	0	0	0
<code>polymorph</code>	2	28	28	25	0	0	0
<code>bzip2</code>	1	19	19	71	0	0	0
<code>pcre</code>	4	96	165	130	0	0	0
<code>bftpd</code>	13	25 (26)	26	67	1	1	1
<code>gzip</code>	6	230	294	76	0	0	0
<code>mcf</code>	3	134	206	26	0	0	0
<code>muh</code>	4	93	97	15	5	5	5
<code>monkey</code>	14	298 (300)	310	46	5	12	21
<code>enscript</code>	24	510 (538)	578 (583)	652	0	0	0
<code>crafty</code>	10	832 (837)	949	842	0	0	0
<code>hypermail</code>	27	891 (971)	1,034 (1,069)	0	0	0	0

**Table 5: Results of the format string vulnerability detector. Numbers in parentheses indicated context-insensitive answers, when different.**

`enscript` relied heavily on simple inner loops which could be optimized in this manner. String processing loops that iterated through each character until the null terminator was found were frequently optimized using this strategy.

Using these techniques, we were able to decrease the number of string checks fired in `hypermail` by 52%, and in `enscript` by 25%. `enscript`’s CRED overhead was reduced from 19% to 16%, with no further improvement due to our string-object optimizations. `hypermail`’s CRED overhead was reduced from 387% to 246%, and optimized CRED overhead was reduced from about 270% to only 170%.

## 7.6 Format String Vulnerabilities

Table 5 shows the results of our format string vulnerability analysis. Column 1 lists the number of calls to functions which provide source pointers to tainted user data. The next two columns display the number of objects found to contain tainted data after our taint analysis. Column 4 lists the number of calls to functions with format strings, and column 5 lists the number of sink objects passed as format string arguments to these calls. We do not count constant strings, the majority of format string arguments, which is reflected by the low number of sink objects in these benchmarks. The last two columns report the number of vulnerabilities. We count a tainted format string once for each callsite which uses it, which explains why `monkey` can have more than 5 vulnerabilities.

The six vulnerabilities found in `bftpd` and `muh` are real. Those found in `monkey` are false positives, and the analysis based on CONS results generates additional false positives. The remaining programs do not have non-constant format string objects to be considered vulnerable.

Given the low number of sink objects, we can learn more about the accuracy of this analysis by looking at the number of tainted objects. The CONS model clearly produces more tainted objects than the PCP model. Context-insensitive pointer relations also produce additional tainted objects, which is most dramatic for `hypermail`. If there were more sink objects, it is possible that these modes would generate additional false positives.

## 8. RELATED WORK

Space limitations preclude us from reviewing the large volume of research on pointer alias analysis in detail. Since

this paper presents a field-sensitive, context- and flow-insensitive, inclusion-based analysis, we shall limit our discussion to similar projects.

Inclusion-based analyses [2] generate significantly more precise results than equivalence-based ones [16]. Zhu [24] and Berndt et al. [3] showed that BDDs can be used effectively to implement inclusion-based points-to analysis. Their algorithm can be formulated as and implemented by simply applying a set of inference rules until they converge. This simplicity makes it relatively easy for us to experiment with the intricate semantics of C. Prior to the use of BDDs, scalable inclusion-based algorithms had to include optimizations to compute closures of points-to relations [8, 10]. Context-sensitive versions of BDD-based points-to analysis for Java and C have been proposed recently [20, 25].

A number of field-sensitive inclusion-based analyses have been proposed for Java [3, 13, 14, 19]. Because of type-safety, however, Java is much easier to handle. Field sensitivity is more complicated with C because of casting and pointer arithmetic. A number of proposed algorithms distinguish between fields of structures in C, but do not handle casting [2, 6, 7]. Steensgaard’s algorithm handles casting; however, it is an equivalence-based rather than an inclusion-based analysis and is therefore less precise [16]. Yong et al. [23] propose the use of common initial sequences; Ryder et al. [12] and Wilson and Lam [22] propose using an offset-based approach.

## 9. CONCLUSIONS

In this paper, we presented a context-sensitive, inclusion-based, and field-sensitive points-to analysis for C. We used the analysis to find objects that potentially have multiple types, to improve a dynamic bounds checker, and to statically detect format string vulnerabilities.

To gain precise C pointer analysis results, we proposed a PCP model which captures the common usage of C. We compared PCP with a CONS analysis which finds more points-to relations than the PCP model, many of which are likely to be spurious. The PCP model found fewer multi-type objects than the CONS model, a result which was more consistent with results we obtained from a dynamic type checker. We also found that the PCP model improved the accuracy of the format string vulnerability detector.

We showed that C pointer alias analysis can be used

to reduce the overhead of a dynamic string-buffer bounds checker. It significantly reduced the overhead for programs dominated by the insertion of non-string objects into the bounds checker's object table. Finally, our analysis found six actual format string vulnerabilities in the programs we analyzed.

## 10. REFERENCES

- [1] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Stanford University, 2000.
- [2] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.
- [3] M. Berndt, O. Lhotk, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114, 2003.
- [4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. In *Proceedings of Software Practice and Experience*, pages 775–802, 2000.
- [5] CERT/CC. Advisories 2002. <http://www.cert.org/advisories>.
- [6] J.-D. Choi, M. Burke, and P. 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*, pages 232–245, 1993.
- [7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
- [8] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.
- [9] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, June 2002.
- [10] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 146–161, June 2001.
- [11] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26, May 1997.
- [12] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.
- [13] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program analysis for Software Tools and Engineering*, pages 73–79, June 2001.
- [14] A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java using annotated inclusion constraints. Technical Report DCS-TR-417, Department of Computer Science, Rutgers University, July 2000.
- [15] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.
- [16] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th Annual ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.
- [17] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, MD, volume II edition, 1989.
- [18] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, February 2000.
- [19] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Symposium on Static Analysis*, pages 180–195. Springer-Verlag, September 2002.
- [20] J. Whaley and M. 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*, pages 131–144, June 2004.
- [21] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium*, pages 149–162, February 2003.
- [22] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.
- [23] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 91–103, June 1999.
- [24] J. Zhu. Symbolic pointer analysis. In *Proceedings of the International Conference in Computer-Aided Design*, pages 150–157, November 2002.
- [25] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157, June 2004.