Improving Software Security with a C Pointer Analysis

Dzintars Avots, Michael Dalton, V. Benjamin Livshits, Monica S. Lam

Standford University

Computer Science Department

Staford, CA 94305

International Conference on Software Engineering 2005

Presented by Bennilyn Quek

Overview

- Software Vulnerabilities in C
- Pointer Alias Analysis
- Practical C Pointers (PCP)
- Conservative Pointer (CONS)
- Type Inference
- C Range Error Detector (CRED)
- Format String Vulnerabilities
- Experiment

Software Vulnerabilities in C

- Lack of type safety
  - Buffer overflows
  - Format string exploits
- Difficult to create precise analysis tools that minimize false warnings
  - Assume pointers are unaliased until proven otherwise with a local analysis
    - May not find all vulnerabilities
    - Cannot guarantee program is safe

Pointer Alias Analysis

Pointer Alias Analysis - Proposal

- **Context-sensitive**
  - Calling contexts along different acyclic call paths have distinct points-to relations
- **Inclusion-based**
  - 2 pointers may point to overlapping but different sets of objects
- **Field-sensitive**
  - Separate fields in object have distinct points-to relations
  - More precise than field-insensitive

Pointer Alias Analysis - Proposal

- Flow-insensitive
  - Program statements execution order not modeled
- Based on **points-to algorithm**
  - Originally for Java
  - Expressed in **Datalog**, use **bddbddb** tool to translate into BDD
- Using **binary-decision-diagram** (**BDD**) representation
  - Handles exponential number of contexts by exploiting their similarities

Pointer Alias Analysis - Application

- Infer variables type based on their usage
- Reduce overhead of dynamic bounds checker
- Find format string vulnerabilities

Practical C Pointers (PCP)

Practical C Pointers (PCP)

- Sound for programs adhering or violating C99 standard in commonly accepted ways

- Example:
  - Use structural equivalence to determine type compatibility
  - C99 use name equivalence for aggregates such as structures & unions

Practical C Pointers (PCP) – Assumption 1

- **Disjoint Object Spaces**
  - Each object allocated in separate memory space
  - A pointer to an object can only be derived from a pointer to the same object
- And additional assumption reflecting typical C usage

Practical C Pointers (PCP) – Assumption 2

- **Variant Types**
  - Pointers written to one field can only be retrieved when accessed as fields that are structurally equivalent

Structural Equivalence

- Any 2 types are type compatible if **physical layout is identical**, regardless of naming
  - Normalize by making types & fields same name
  - Simple inspection of field paths

- Fields of 2 structures (or variants of a union type) are structurally equivalent if they have the **same offset** & have **structurally equivalent field types**

- Structurally equivalent types are interchangeable

Structural Equivalence – Structural Induction

- **Scalar types**
  - Primitive types – if and only if have same sizes
  - Object (field) – structurally equivalent to a leading field of a structure with a structurally equivalent field type
- **Structures & union types**
  - If have same number of fields & all their fields are structurally equivalent
  - Structurally equivalent fields referred to as having *common initial sequence*

Dynamic Variant Types

- Objects may be cast between multiple types
- If data stored under a variant type is retrieved through another type
  - Exact values read depend on platform-specific physical layout
- In ANSI C standard
  - Data store under a variant type is not expected to be retrieved when object is cast to another type

Dynamic Variant Types

- If object is cast to another structurally nonequivalent type
  - Assumes reads from a field in one variant type will not be used to access data stores to a structurally nonequivalent field in another variant type

Practical C Pointers (PCP) – Assumption 2

- Example:

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

  - Optimistic model will not recognize character ‘c’ in u.c will be accessed through u.x

Practical C Pointers (PCP) – 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

Practical C Pointers (PCP) – Assumption 3

- Example:

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

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

  - (1)(2) common use of type casts
  - Base address of **a** is same as 1st field **A1**, pointing to either type **A** or **B**. Can calculate field offset for **B2**.

Practical C Pointers (PCP) – Assumption 3

- Example:

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

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

  - When points to **a.A2**, addressed structure must be treated as type **int**
  - Assignment in (4) cannot be modelled given address from (3)

Practical C Pointers (PCP) – Assumption 4

- **Pointer Arithmetic**
  - Arithmetic is applied only to pointers pointing to an array element to compute another element in the same array

- Allow precise modeling of common usage of in-bounds pointer arithmetic by leaving uncommon behavior unmodeled

Practical C Pointers (PCP) – Assumption 4

- Model array as 2 fields:
  - Leading field **f1** containing 1st element
  - 2nd field **f2** represents all remaining elements
  - Both fields are structurally equivalent to type **t**, (array element type)

- If 2 array types have same element type, their **f2** are considered structurally equivalent, regardless of array size
  - Array types are only structurally equivalent only if they are same size

Practical C Pointers (PCP) – Assumption 5

- **Memcopy**
  - Assume 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, match a common initial sequence of fields with each type at the destination address
  - Precisely model the operation as an assignment between corresponding fields

Conservative Pointer (CONS)

Assumptions

Conservative Pointer (CONS)

- Includes inference rules in PCP model
- Additional rules to model program behavior that falls outside of PCP assumptions

- Points-to relations found by PCP model are a subset of relations found by CONS model

Conservative Pointer (CONS) – Assumption 1

- Assumes only that displacements between objects are undefined

- **Disjoint Object Spaces**
  - Each object allocated in separate memory space
  - A pointer to an object can only be derived from a pointer to the same object

Conservative Pointer (CONS) – Assumption 2

- Any read from a field also retrieves values from any possibly overlapping field

- **Variant Types**
  - Pointers written to one field can only be retrieved when accessed as fields that are structurally equivalent

Conservative Pointer (CONS) – Assumption 3

- 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 dereference, as well as all fields following **p**, since the range of dereference may span into those fields as well

- **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

Conservative Pointer (CONS) – Assumption 3

- May lookup field **f1** in type **t1** relative to path **p**, where there is no type **t1** field
- CONS return the address of every field following **p** which can overlap with a hypothetical **f1** field

- **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

Conservative Pointer (CONS) – Assumption 4

- Assumes that a computed pointer may point to anywhere in the object pointed to by the source pointer, unless it is proven otherwise

- **Pointer Arithmetic**
  - Arithmetic applied only to pointers to an array element to derive pointers pointing to another element in the same array

Conservative Pointer (CONS) – Assumption 5

- Copy data beyond the end of the largest field, and the destination location does not need to be structurally equivalent

- **Memcopy**
  - Assume **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

Type Inference

Type Inference

- Declared variable type may not reflect actual type
- But can infer from their uses
- Conflicts between declared & inferred types flag dubious coding practice

- Can show if program is safe & portable
- Identify unsafe operations to be audited statically or checked dynamically

Type Inference – Finding Type

- By identifying accessed field paths in the object
- For each accessed path
  - Know object was cast to a structure type containing that field path
- Single type if all accessed field paths belong to a common structure type
- Query results to find objects cast as multiple types

C Range Error Detector (CRED)

C Range Error Detector (CRED)

- A dynamic bound checker
- Run with less overhead by checking only string buffer overflows

- Has 2 kinds of runtime overheads:
  - From maintaining object table as stack & heap objects are allocated & de-allocated
  - As address computations & string manipulations are checked for out-of-bounds (OOB) addresses

C Range Error Detector (CRED)

- Based on Jones & Kelly's **referent objects concept**

- Pointer should only be used to reference an associated referent object
- Any derived pointer will have same referent object as original pointer
- When dereferenced, pointer point within bounds of its referent object

C Range Error Detector (CRED)

- CRED uses object table to dynamically track base & extent of each object
- CRED enters all referent objects into object table

- To reduce overhead of strings-only CRED
  - Use points-to results to inform CRED which objects do not contain string-typed values
  - Reduce some benchmarks by 30% to 100%

Format String Vulnerabilities

Format String Vulnerabilities

- When user-supplied string contain format specifiers that can cause program to write to unintended memory locations

- Taint analysis is used to detect which object fields may contain data originally derived from user input

  - Static analysis – find all potential vulnerabilities before running program
  - Dynamic analysis – detect vulnerability only if given appropriate test inputs & will incur runtime overhead if used as preventative measure

Format String Vulnerabilities - Taint

- Source objects
  - All objects directly under user control
  - Example:
    - **Argv** array, return results of IO function, functions interacting with environment **gatenv**
  - Find source objects using parameter & return value reference from selected system functions

- Sink objects
  - Objects referenced by format string argument
  - Tainted sink object indicates potential vulnerability

Experiment

Experiment – Benchmarks

- **ffingerd** – finger deamon
- **polymorph** – filename conversion
- **bzip2** – compression utility
- **pcre** – regular expression library
- **bftpd** – FTP server
- **gzip** – compression utility
- **mcf** – scheduling program
- **muh** – network game
- **monkey** – web server
- **enscript** – convert text to PostScript
- **crafty** – chess playing
- **hypermail** – convert mailbox files to HTML

Experiment – Setup

- Runtime measurements taken on AMD
  - Opteron 150 machine
  - 1GB memory
  - Linux
- Use test cases provided
  - If unavailable, create large test files with common inputs

Experiment – Steps

- Transform program into collection of simplified operations
- Build a set of BDD relations
- Compile & link with SUIF 2 compiler framework
- Number objects (distinguishing between SSA versions)
- Normalize types & fields names
- Map legal field paths to canonical paths

Experiment – Resources

- Context-insensitive PCP analysis
  - 32 Datalog inference rules
  - Defined 200 external system calls through use of additional rules

- Context-sensitive analysis
  - Need to convert existing rules to augment each occurrence of object variable with accompanying context variable & assignments of actual arguments to formal parameters with proper context numbering

Experiment – Resources

- CONS analysis
  - Additional 26 rules on top of PCP
  - Additional rules create new relations that may propagate further & increase runtime
  - Choice of BDD variable domain ordering can significantly affect performance of Datalog program
    - Use ordering discovered by **bddbddb**

Experiment – Pointer Analysis

- Context-insensitive PCP
  - Most take <6 seconds to run
- CONS
  - 2.5 times as long as PCP
- Context-sensitive
  - 3 times as long as context-insensitive

Experiment – Pointer Analysis

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

- 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.

Experiment – Static Type Checking

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

Experiment – Dynamic Type Checking

- Unable to test **bftpd** or **muh**
- Out of 10, found 2 containing objects treated with more than 1 type
  - Unusual in portable program, requires making assumptions about size & alignment of C's types

Experiment – Dynamic Type Checking

- **hypermail** cast double to array of chars to access it byte by byte
- **bzip2** casts array of 32-bit int to 16-bit shorts
- Violated assumptions to get at low-level representation of numeric data for PCP
  - Code inspection showed they are not involved in propagation of pointer values

Experiment – Optimization of CRED

Benchmark	Heap allocs	Stack allocs	Bounds checks	Non-strings % Heap	Non-strings % 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

- 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.

Experiment – Optimization of CRED

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

- Reducing the runtime overhead of strings only CRED by entering only strings in the object table.

Experiment – Format String Vulnerabilities

Benchmark	Source calls	Tainted objects PCP	Tainted objects CONS	Sink calls	Sink objects	Vulnerabilities actual	Vulnerabilities false
ffingerd	3	23	23	15	0	0	0
polymorph	2	28	28	25	0	0	0
bzip2	1	19	19	71	0	0	0
pcre	4	96	165	130	0	0	0
bftpd	13	25 (26)	26	67	1	1	0
gzip	6	230	294	76	0	0	0
mcf	3	134	206	26	0	0	0
muh	4	93	97	15	5	2	0
monkey	14	298 (300)	310	46	5	9	15
enscript	24	510 (538)	578 (583)	652	0	0	0
crafty	10	832 (837)	949	842	0	0	0
hypermail	27	891 (971)	1,034 (1,069)	0	0	0	0

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

Summary

- Context-sensitive, inclusion-based, field-sensitive points-to analysis for C

- Used the analysis to
  - Find objects that potentially have multiple types
  - Improve dynamic bounds checker
  - Statically detect format string vulnerabilities

Summary

- PCP
  - Found fewer multi-type objects
  - Result more consistent with results from dynamic type checker
  - Improved accuracy of format string vulnerability detector

Summary

- Reduce overhead of a dynamic string-buffer bound checker
- Reduce overhead for programs dominated by insertion of non-string objects into bound checker's object table
- Found 12 format string vulnerabilities with only 1 false positive

Questions

- What do you find significant about the work presented and why?
- Would the CONS model be better than PCP in certain cases?
- What would the next step be? (Future development/improvement)