

# CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C

An article by Nurit Dor, Michael Rodeh and Mooly Sagiv  
Presentation by Antoine Amarilli

École normale supérieure

# Table of contents

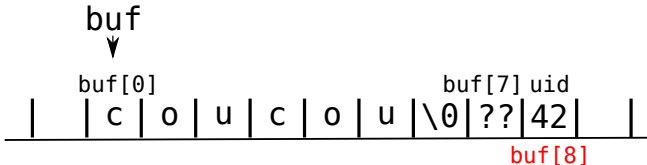
- 1 General Presentation
  - Quick facts
  - The Main Problem
  - Overview of the Solution
  
- 2 In-Depth Analysis
  - Preliminary Steps
  - Pointer Analysis
  - Integer Program
  
- 3 Results and Perspectives
  - Results
  - Perspectives
  - References

## Quick facts

- Who?** Nurit Dor, Michael Rodeh, Mooly Sagiv, from Tel-Aviv University and the IBM Research Lab in Haifa
- Where?** PLDI (Programming Language Design and Implementation)
- When?** 2003
- What?** Static detection of buffer overflows in C
- How?** As a follow-up to a previous study in 2001, with support for more language constructs and better efficiency, and as part of Nurit Dor's ongoing PhD thesis.

# Buffer overflow

- Performing out-of-bound accesses to an array in C can access other values of the program.
- A buffer overflow is an unsafe access of this kind. Such accesses can occur because of bugs in the program.



```
char buf[8] = "coucou";  
char uid = 42;
```

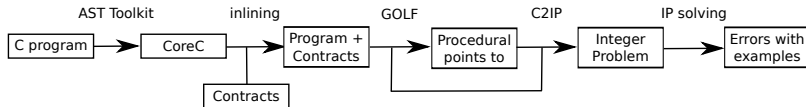
# Buffer overflow problems

- The program can crash or misbehave when such a bug occurs.
- A malicious user can use such bugs to access confidential data or to overwrite data and alter the program's behavior.
- Buffer overflows, and the more specific string manipulation errors, are a common bug in C. The FUZZ study from 1995 is quoted as evidence (60% of Unix failures due to string manipulation errors).

# CSSV's proposed solution

- Perform static analysis to identify string manipulation errors.
- The approach used in the paper is *sound*, meaning that it should identify all errors. However, it raises false alarms.
- Be as precise as possible to minimize the number of false alarms.
- Generate examples when a problem is identified.

# Overview of the solution

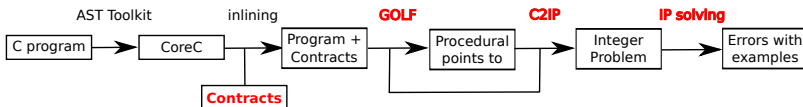


- 1 Translate to CoreC, a simpler subset of C.
- 2 Annotate procedures with contracts (pre- and postconditions) and inline them in the program.
- 3 Perform a static analysis to identify possible pointing targets for pointers.
- 4 Use this information to translate the program in an integer problem.
- 5 Solve this problem.

# False alarms

Possible causes for false alarms:

- 1 Insufficient procedure contracts.
- 2 Abstractions performed when converting to an integer program.
- 3 Imprecision of the pointer or integer analyses.



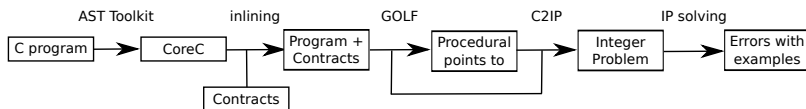


# Table of contents

- 1 General Presentation
  - Quick facts
  - The Main Problem
  - Overview of the Solution
  
- 2 In-Depth Analysis
  - Preliminary Steps
  - Pointer Analysis
  - Integer Program
  
- 3 Results and Perspectives
  - Results
  - Perspectives
  - References

## Translation to CoreC

- C is an expressive language, it is hard to support all of its features.
- For this reason, a first translation pass is performed to translate the program to CoreC.
- CoreC is a complete subset of C with semantics-preserving translation rules.
- The implementation of this transformation uses Microsoft's AST Toolkit (now called PREfast).

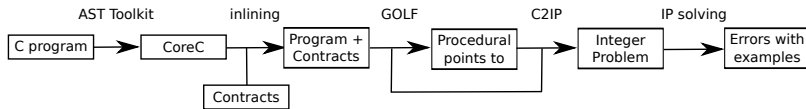


# Contract specification

- Contracts are written for every procedure which specify:
  - ① The assumptions made by the procedure.
  - ② The side effects of the procedure.
  - ③ The guarantees upheld by the procedure.
- They are written in the style of the Larch tool, and are an extension of Hoare triples to C.
- Contracts must be written by hand, though a contract derivation mechanism is sketched (more later).

# Contract inlining

- Contracts are inlined in the program with `assert`'s and `assume`'s.
- An `assume` is added at procedure entry points to check preconditions.
- An `assert` is added at procedure exit points to check postconditions.
- Procedure calls `assert` the preconditions and `assume` the postconditions.



## Concrete program state

**Memory locations** from dynamic and static allocation.

**Base addresses** distinguished from these locations.

**Allocation size** from every base address.

**Assigned memory locations** of each variable (always a base address).

**Actual contents** of memory locations, which can be the address of a memory location, a primitive value, “uninitialized” or “undefined”.

**Size of the value** stored starting at a location.

**Base address mapping** to recover the base address of a location.

# Concrete program state restrictions

**Admissibility.** Require that when a base value isn't "undefined", unaligned accesses up to its contents' size yield "undefined" and there is no overlapping non-"undefined" value before it.

*Intuition:* this is a reasonable structural restriction on concrete program states.

**Reachability.** We aren't concerned with locations which aren't referenced by a visible variable.

*Intuition:* abstract program state will not deal with non-reachable variables.



## Abstract program state

**Base addresses** for reachable base addresses in the concrete.

**Locations** mapping variables to a set of possible abstract locations.

A **pointer relation** indicating, for each abstract location, the set of locations which may point to this location.

A **count** indicating if an abstract location represents exactly one address or a potentially unbounded set of addresses.

These abstractions are defined for each procedure, and are restricted to addresses which are reachable within this procedure.

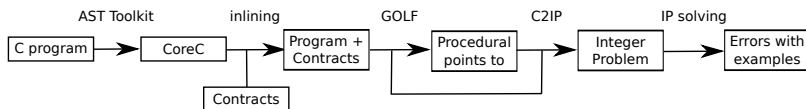
# Sound abstraction

- Base.** All concrete base addresses are mapped to an abstract memory location.
- Stack.** All visible variables are in a concrete location which is mapped to a possible abstract location for this variable.
- Pointer.** If a reachable location points to another location in the concrete, then their base addresses are mapped to two addresses related by the pointer relation.

A procedural abstract points-to-state is a *sound approximation* of a procedure if it is a sound approximation of all the possible concrete states that may arise during this procedure.

# Flow-insensitive pointer analysis

- The aim of this step is to compute a sound abstraction.
- We first apply the GOLF whole-program flow-insensitive analysis to get a sound approximation for all procedures.
- We then restrict this abstraction to the visible variables of a procedure and project the location and pointer relations.
- We refine further by merging the various locations that a node points to, when it is safe to do so.



# Conversion to an integer program (C2IP)

- The constraints over the pointers can be expressed as an integer program (a program which manipulates integer variables and enforces inequalities).
- For every abstract location, we generate several constraint variables:

- **Primitive values** stored in this location.
- **Pointer offset** for pointers stored in this location, relative to their base address.
- **Allocation size** of pointers stored in this location.
- **Null-termination** of the string stored in this location.
- **String length** of the string stored at this location.

# Conversion rules

Here are a few examples to illustrate how the IP is generated:

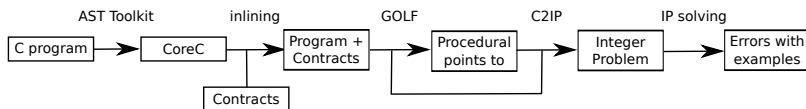
**Dereferencing.** Check that the offset is positive, that we are not going beyond the allocated space, and beyond the string length for strings.

**Pointer arithmetic.** When adding a value to a pointer, check that the result does not go before the base address or beyond the allocated space, and update the offsets.

**Allocation.** Initialize the offset to zero, initialize the size, say that it is not a null-terminated string.

**Writes to a pointer.** When assigning a known zero, we can create a null-terminated string.

**Reads from a pointer.** The read value is unknown unless it is the null-termination of a string.



# Integer analysis

- Any sound integer analysis can be used to study the IP.
- We privilege an integer analysis which is able to identify relationships between variables (instead of tracking each variable's value individually).
- The method used (by Cousot and Halbwachs) is able to infer linear inequalities between program variables.
- The implementation uses the NewPolka library.



# Table of contents

- 1 General Presentation
  - Quick facts
  - The Main Problem
  - Overview of the Solution
- 2 In-Depth Analysis
  - Preliminary Steps
  - Pointer Analysis
  - Integer Program
- 3 Results and Perspectives
  - Results
  - Perspectives
  - References

## Code samples

- The analysis is run on two different examples:
  - A string library from EADS airbus totalling 228 LOCs, on which no errors are found and six false alarms are generated.
  - Part of web2c, totalling 117 LOCs, on which eight errors are found and two false alarms are generated.
- The analysis reports the CPU time and memory usage and the size of the integer problem.

# Experimental results

App.	Function	Source Code			CSSV				Msg		Deriving			
		LOC	SLOC	Contract	IP Vars	IP Size	CPU sec	Space MB	False	Errors	CPU sec	Space MB	Vacuous	Auto
EADS Airbus	RTC_Si_SkipLine	13	260	SBI	39	109	2.6	12	0	0	0.3	3	5	5
	RTC_Se_CopieEtFiltre	66	773	SB	127	812	206	347	6	0	95	433	24	24
	RTC_Si_FiltrerCarNonImp	19	114	S	13	151	0.3	2	0	0	0.2	2	4	4
	RTC_Si_Find	26	820	SI	108	476	2.7	24	0	0	1.4	54	4	1
	RTC_Si_StrNCat	8	299	SBI	54	182	0.9	6	0	0	0.2	3	2	0
	RTC_Si_CalculerStringTime	33	567	SBI	86	529	76	127	0	0	131	173	21	4
	RTC_Si_FormatMcdulo- Formatprinter	18	273	SB	58	323	6.9	28	0	0	6.8	27	9	9
	RTC_Si_StoreIntInBuffer	35	222	SBI	59	346	9.8	43	0	0	3.3	22	15	15
RTC_Se_ComposerEntete	10	550	SI	77	352	3.4	23	0	0	1.3	12	2	0	
fixwrites	remove_newline	12	260	S	35	203	0.1	2	0	0	0.61	1	1	0
	insert_long	14	367	SB	138	571	13	99	0	2	23.4	86	5	0
	join	15	701	SB	95	443	2.1	23	0	2	6.7	15	2	2
	whole	30	423	S	46	352	1.2	20	0	1	0.6	4	9	9
	skip_balanced	20	258	SB	29	215	0.3	5	2	0	0.6	3	6	6
	bare	26	333	S	41	319	0.6	12	0	3	0.4	9	11	11

**Table 5: The experimental results.**

# The burden of contracts

- Though CSSV improves on previous approaches, writing correct contracts for procedures remains an obstacle.
- An algorithm is presented to compute an approximation to the strongest postcondition and weakest precondition to automatically strengthen contracts.
- The algorithm proceeds by forward and backward integer analysis to infer variable inequalities and add them to the contracts.
- Experimental results show a 25% false alarm reduction for automatically derived contracts as opposed to vacuous contracts.
- This needs to be compared to the 93% false alarm reduction achieved with manual contracts.

## Pros and cons





The **good points** of the approach are:

- Support of the full C language (via CoreC translation).
- Soundness.
- Low number of false alarms reported.
- Computational efficiency (compared to the 2001 paper).





The **shortcomings** are:

- False alarms are reported nevertheless.
- Contracts need to be written manually.
- Scalability can be an issue.

# References

-  Nurit Dor, Michael Rodeh, Shmuel Sagiv. “CSSV: towards a realistic tool for statically detecting all buffer overflows in C”. *PLDI 2003*: 155-167.
-  Nurit Dor, Michael Rodeh, Shmuel Sagiv. “Cleanness Checking of String Manipulations in C Programs via Integer Analysis”. *SAS 2001*: 194-212.
-  Nurit Dor. “Automatic Verification of Program Cleanness”. PhD thesis, Tel Aviv University, December 2003.
-  Greta Yorrsh, “The Design of CoreC”.  
<http://www.cs.tau.ac.il/~gretay/gfc/simplifyCC.pdf>

## References (cont'd)

-  B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, J. Steidl. “Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services”. *Computer Sciences Technical Report #1268*, University of Wisconsin-Madison, April 1995.
-  Manuvir Das, Ben Liblit, Manuel Fähndrich, Jakob Rehof. “Estimating the Impact of Scalable Pointer Analysis on Optimization”. *SAS 2001*: 260-278.
-  Bertrand Jeannot. “NewPolka”. <http://pop-art.inrialpes.fr/people/bjeannot/newpolka/>
-  Patrick Cousot, Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. *POPL 1978*: 84-96.

Thanks!

Thanks for your attention!