Ensuring Safe Usage of Buffers in Programming Language C

Milena Vujošević–Janičić
Faculty of Mathematics, University of Belgrade
Studentski trg 16, Belgrade, Serbia
www.matf.bg.ac.yu/~milena

ICSOFT 2008
Porto, Portugal, July 5-8, 2008.

Roadmap

- Buffer Overflows
- Proposed Approach
- The FADO Tool
- Conclusions and Future Work

Roadmap

- Buffer Overflows
- Proposed Approach
- The FADO Tool
- Conclusions and Future Work

Buffer Overflows

- A *buffer overflow* (or *buffer overrun*) is a programming flaw which enables storing more data in a data storage area (i.e. buffer) than it was intended to hold.

- Buffer overflows are the most frequent and the most critical flaws in programs written in C.

- Buffer overflows are suitable targets for security attacks and source of serious programs' misbehavior. Buffer overflows account for around 50% of all software vulnerabilities.

- In handling and avoiding possible buffer overflows, standard testing of software is not sufficient.

Buffer Overflows — Static and Dynamic Analysis

- The problem of automated detection of buffer overflows has attracted a lot of attention over the last ten years.

- There are two approaches for detecting buffer overflows:
  - Tools based on dynamic analysis examine the program while it is being executed (dynamic testing, specialized compilers, library of functions, operating systems).
  - Tools based on static analysis examine the source code of the program and aim at detecting buffer overflows before the execution.

Buffer Overflows — Static Analysis Tools

- Lexical analysis (ITS4 (2000), RATS (2001), Flawfinder (2001))
- Semantical analysis
  - BOON (Univ. of California, Berkeley, USA, 2000)
  - Splint (Univ. of Virginia, USA, 2001)
  - CSSV (Univ. of Tel-Aviv, Israel, 2003)
  - ARCHER (Stanford University, USA, 2003)
  - UNO (Bell Laboratories, 2001)
  - Caduceus (Univ. Paris-Sud, Orsay, France, 2007)
  - Polyspace C Verifier, AsTree, Parfait, Coverty, CodeSonar

Roadmap

- Buffer Overflows
- Proposed Approach
- The FADO Tool
- Conclusions and Future Work

Proposed Approach

- The proposed approach belongs to the group of static analysis methods based on semantical analysis of source code.

- The main motivation is to make a system with a flexible architecture that enables easily changing of components of the system and simple communication with different external systems.

- Correctness conditions are expressed in terms of first order logic, linear arithmetic and verified by a SMT theorem prover.

Parser and intermediate code generator
– parsing
– intermediate code generating

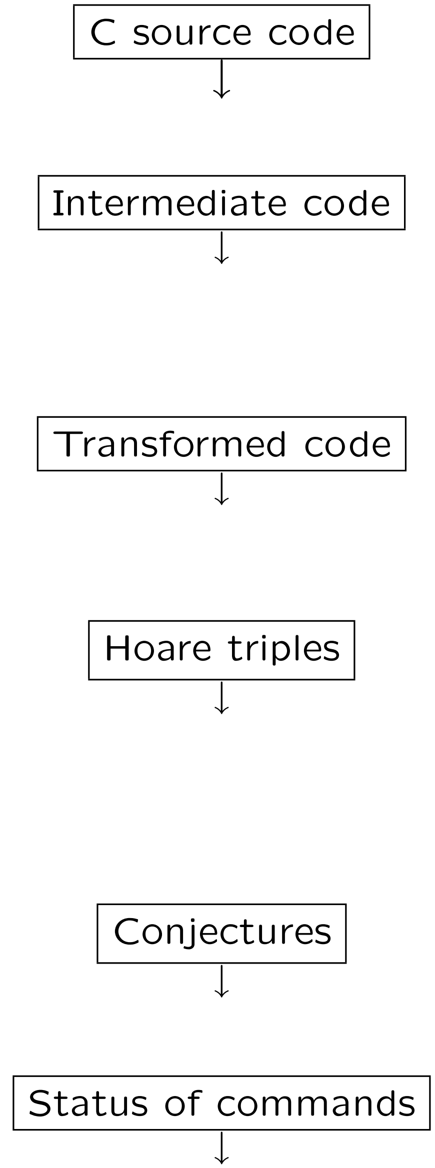
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– updating states for sequences of commands

Generator and optimizer for correctness and incorrectness conjectures
– resolving preconditions and postconditions of functions
– eliminating redundant conjuncts
– evaluation
– abstraction

Automated theorem prover for LA
– processing input formulae in smt-lib format
– returning results

Results
– providing explanations for status of the commands



Parser and intermediate code generator
– parsing
– intermediate code generating

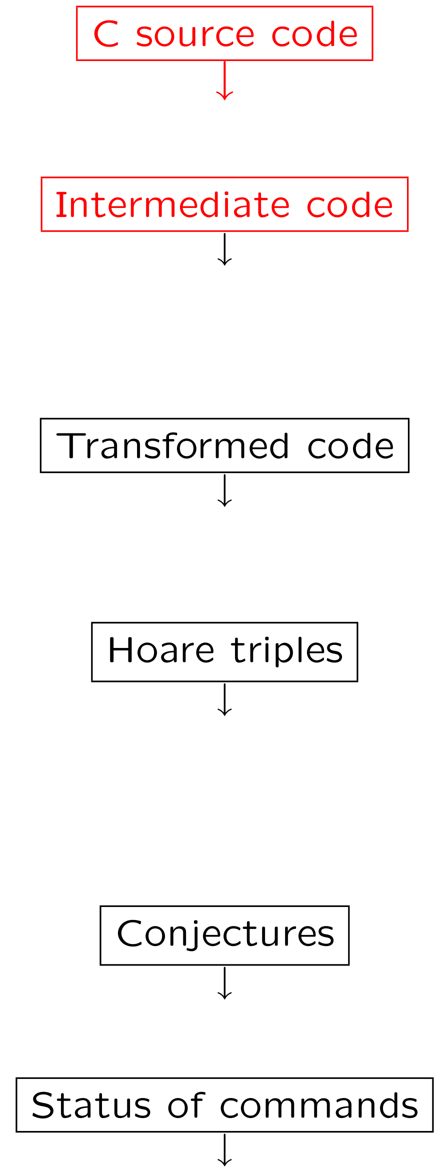
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– updating states for sequences of commands

Generator and optimizer for correctness and incorrectness conjectures
– resolving preconditions and postconditions of functions
– eliminating redundant conjuncts
– evaluation
– abstraction

Automated theorem prover for LA
– processing input formulae in smt-lib format
– returning results

Results
– providing explanations for status of the commands



Parser and intermediate code generator
– parsing
– intermediate code generating

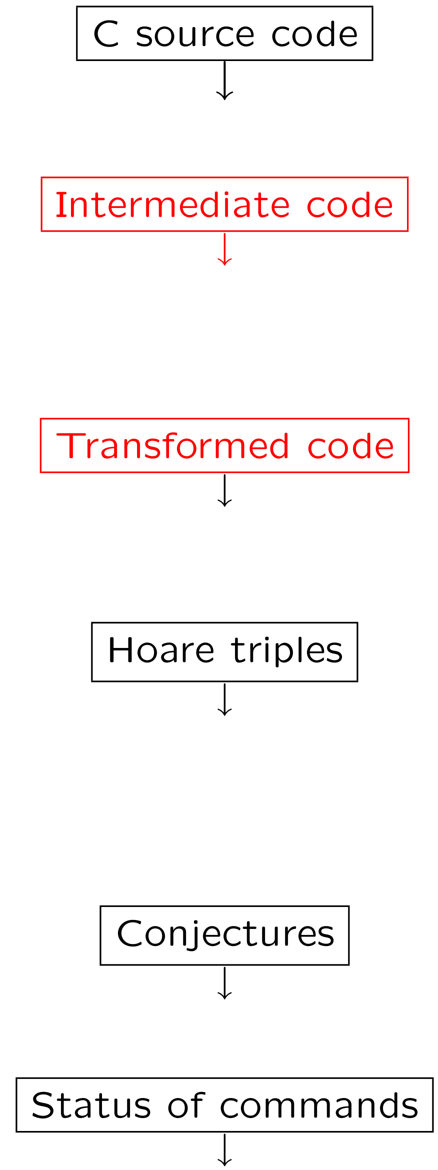
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– updating states for sequences of commands

Generator and optimizer for correctness and incorrectness conjectures
– resolving preconditions and postconditions of functions
– eliminating redundant conjuncts
– evaluation
– abstraction

Automated theorem prover for LA
– processing input formulae in smt-lib format
– returning results

Results
– providing explanations for status of the commands



Parser and intermediate code generator
– parsing
– intermediate code generating

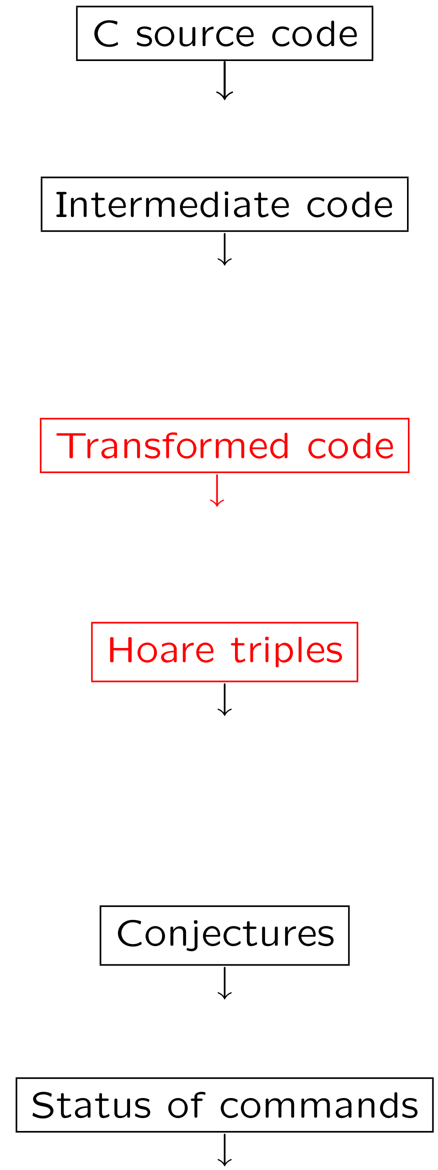
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– updating states for sequences of commands

Generator and optimizer for correctness and incorrectness conjectures
– resolving preconditions and postconditions of functions
– eliminating redundant conjuncts
– evaluation
– abstraction

Automated theorem prover for LA
– processing input formulae in smt-lib format
– returning results

Results
– providing explanations for status of the commands



Proposed Approach — Database of Conditions

- The database of conditions is used for generating correctness conditions for individual commands.

- The database stores triples (*precondition*, *command*, *post-condition*). The semantics of a database entry (ϕ, E, ψ) is:

  - in order E to be safe, the condition ϕ must hold;
  - in order E to be flawed, the condition $\neg\phi$ must hold;
  - after E, the condition ψ holds.

- The database is external and open. Initially, it stores reasoning rules about operators and functions from the standard C library. Also, the user can add or remove entries.

Proposed Approach — Modelling Semantics of Programs

- For defining correctness conditions we use meta-level functions:
  - $value$, returns a value of a given variable;
  - $size$, returns a number of elements allocated for a buffer;
  - $used$, relevant only for string buffers, returns a number of elements used by the given buffer (including `'\0'`).

- These functions have an additional argument called state or timestamp, which provides basis for flow-sensitive analysis and a form of pointer analysis (similar to SSA).

Proposed Approach — Generating Correctness Conditions

- Examples of database entries:

precondition	command	postcondition
–	`char x[N]`	$size(x,1) = value(N,0)$
–	`x = y`	$value(x,1) = value(y,0)$

- For an individual command C, if there is a database entry (ϕ, E, ψ) such that there is a substitution σ such that $C = E\sigma$, then $precond(C) = \phi\sigma$ and $postcond(C) = \psi\sigma$.
- States are updated in order to take into account the wider context of the command. For example:

code	postcondition
`int a,b;`	—
`a = 1;`	$value(a,1) = value(1,0)$
`b = 2;`	$value(b,1) = value(2,0)$
`a = b;`	$value(a,2) = value(b,1)$

Proposed Approach — Generating Correctness Conditions

- Postcondition for an `if` command are constructed as follows:

precondition	command	postcondition
–	`if(p)`	
–	`{`	p
$precond(C1)$	`C1;`	$postcond(C1)$
$precond(C2)$	`C2;`	$postcond(C2)$
	`...;`	$\ldots$
–	`}`	$(p \wedge postcond(C1) \wedge postcond(C2) ...)$
		$\vee(\neg p \wedge update\_states)$

- Currently, loops are processed in a limited manner — only the first iteration is considered (which is often sufficient).

Parser and intermediate code generator
– parsing
– intermediate code generating

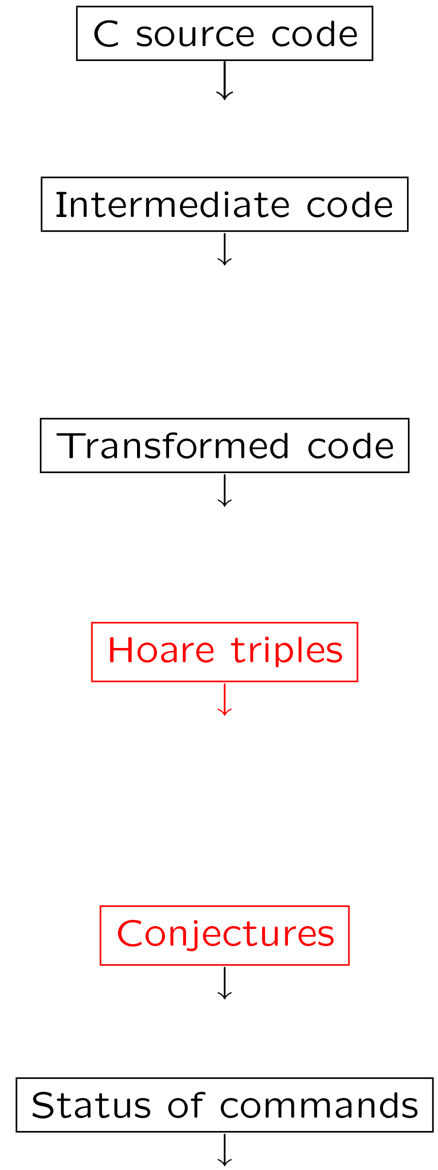
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– updating states for sequences of commands

Generator and optimizer for correctness and incorrectness conjectures
– resolving preconditions and postconditions of functions
– eliminating redundant conjuncts
– evaluation
– abstraction

Automated theorem prover for LA
– processing input formulae in smt-lib format
– returning results

Results
– providing explanations for status of the commands



Proposed Approach — Correctness Conjectures

- For a command C, let Φ be conjunction of postconditions for all commands that precede C. The command C is:
  - **safe**, if $(\forall *)(\Phi \Rightarrow precond(C))$ is valid;
  - **flawed**, if $(\forall *)(\Phi \Rightarrow \neg precond(C))$ is valid;
  - **unsafe**, if neither of above;
  - **unreachable**, if it is both safe and flawed.
- Before sending conjectures to the prover, evaluation, elimination of irrelevant conjuncts and abstraction are applied.

Parser and intermediate code generator
– parsing
– intermediate code generating

Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– updating states for sequences of commands

Generator and optimizer for correctness and incorrectness conjectures
– resolving preconditions and postconditions of functions
– eliminating redundant conjuncts
– evaluation
– abstraction

Automated theorem prover for LA
– processing input formulae in smt-lib format
– returning results

Results
– providing explanations for status of the commands

C source code
↓
Intermediate code
↓
Transformed code
↓
Hoare triples
↓
Conjectures
↓
Status of commands
↓

Parser and intermediate code generator
– parsing
– intermediate code generating

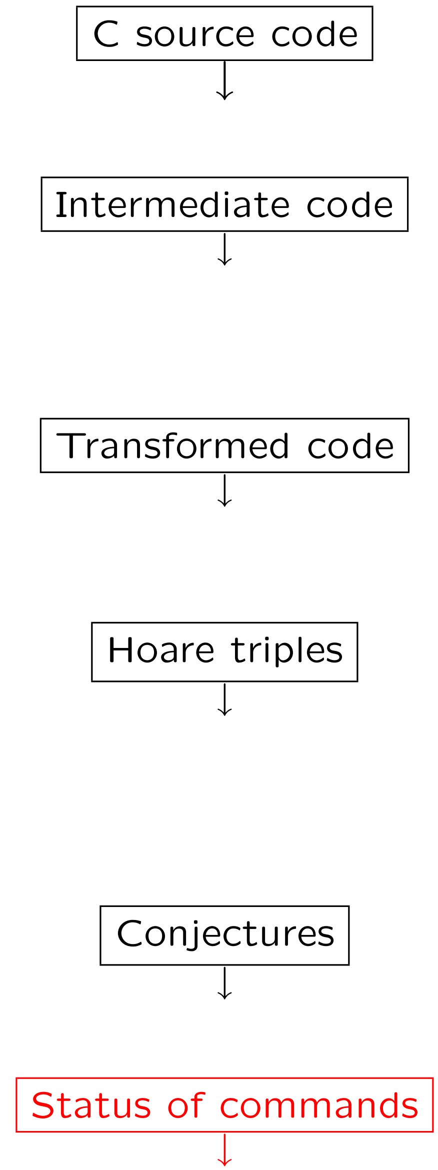
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– updating states for sequences of commands

Generator and optimizer for correctness and incorrectness conjectures
– resolving preconditions and postconditions of functions
– eliminating redundant conjuncts
– evaluation
– abstraction

Automated theorem prover for LA
– processing input formulae in smt-lib format
– returning results

Results
– providing explanations for status of the commands



Proposed Approach — Example

For the following fragment of code:

```
char src[200];
fgets(src,200,stdin);
```

if the database of conditions contains the following entries:

precondition	command	postcondition
–	`char x[N]`	$size(x,1) = value(N,0)$ $\wedge used(x,1) > 0$
$size(x,0) \geq value(y,0)$	`fgets(x,y,z)`	$used(x,1) \leq value(y,0)$ $\wedge used(x,1) > 0$

then the following conditions are generated:

precondition	command	postcondition
–	`char src[200]`	$size(src,1) = value(200,0)$ $\wedge used(src,1) > 0$
$size(src,0) \geq value(200,0)$	`fgets(src,200,stdin)`	$used(src,1) \leq value(200,0)$ $\wedge used(src,1) > 0$

Proposed Approach — Example

Using the generated conditions, the correctness conjecture for the command `fgets(src,200,stdin)` is

$$(0 < used(src, 1)) \wedge (size(src, 1) = value(200, 0)) \Rightarrow (size(src, 1) \geq value(200, 0))$$

After evaluation, the conjecture becomes:

$$(0 < used(src, 1)) \wedge (size(src, 1) = 200) \Rightarrow (size(src, 1) \geq 200)$$

After abstraction, the conjecture becomes:

$$(0 < used\_src\_1) \wedge (size\_src\_1 = 200) \Rightarrow (size\_src\_1 \geq 200)$$

This formula is transformed to SMT-lib format and sent to an automated theorem prover which can confirm its validity. Therefore, the usage of the command `fgets(src,200,stdin)` is safe.

Roadmap

- Buffer Overflows
- Proposed Approach
- The FADO Tool
- Conclusions and Future Work

The FADO Tool

- *FADO* — *F*lexible *A*utomated *D*etection of Buffer *O*verflows
- The tool is implemented in programming language C++, it consists of $\approx$ 13000 lines of code organized in 35 classes.
- The architecture of the tool follows the described phases.
- It uses two external systems: JSCPP parser and ArgoLib theorem prover.
- Modularity makes the tool very flexible: different components can be easily updated or replaced by alternatives.

The FADO Tool — Experimental Results

- Evaluation results are obtained on the set of benchmarks (291 programs in four versions) that is freely available at `http://www.ll.mit.edu/IST/corpora.html`.
- On these benchmarks, FADO detected 57% of buffer overflows, with 6.5% false alarm rate. With additional, specific database entries, the false alarm rate was 3%.
- From the remaining flaws:
  - 35% are due to the loops that cannot currently be processed;
  - 3% cannot be detected because the current implementation still does not cover some programming constructs.
  - 5% are substantially beyond the reach of our system.

The FADO Tool — Experimental Results

- For processing these 291 test programs, FADO spends 46.8s on a PC computer with 2.4GHz and 768MB RAM memory. Average time for a single test program is 0.16s.

- The times spent by different phases were:

Phase	Percent of time
Parsing	1.2%
Transforming	0.5%
Generating conditions	51.8%
Exporting and testing conjectures	46.4%
Processing and formatting results	0.1%

FADO Tool — Experimental results

The results of experimental comparison based on the mentioned corpus:

Tool	Detection rate	False alarm rate	Confusion rate	Average CPU time spent
PolySpace	99.7	0.0	2.4	172.53s
ARCHER	90.7	0.0	0.0	0.25s
FADO	57.0	6.5	12.5	0.16s
Splint	56.4	12	21.3	0.02s
UNO	51.9	0.0	0.0	0.02s
BOON	0.7	0.0	0.0	0.06s

Roadmap

- Buffer Overflows
- Proposed Approach
- The FADO Tool
- Conclusions and Future Work

Conclusions and Future Work

- A new, static, modular system for automated detection of buffer overflows in programs written in C is presented.

- The system analyzes the code, generates correctness and incorrectness conjectures for individual commands, and invokes an external automated theorem prover for linear arithmetic to test the generated conjectures.

- The FADO tool is a prototype implementation of the presented system, and it gives promising results.

Conclusions and Future Work

- Some of the novelties that our system introduce are:
  - its modular and flexible architecture (so its building blocks can be easily changed and updated),
  - an external and open database of conditions (so the underlying reasoning rules are not hard-coded into the system),
  - buffer overflow correctness conjectures given explicitly in logical terms,
  - usage of external theorem provers and related standards.

Conclusions and Future Work

- Future work:
  - Extend the system to preform a deeper analysis of loops and of user defined functions, so the system will be sound and its inter-procedural analysis will be fully automatic.
  - Use theorem provers with more expressive background theories.
  - Extend the system for other sorts of program analysis (e.g., detecting memory leaks).

Thank You for Your Attention