Using SMT Solver in Detection of Buffer Overflow Bugs

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

Second Workshop on Formal and Automated Theorem Proving and Applications
Belgrade, Serbia, January 30-31, 2009.

Context

- SAT and SMT solvers have many applications in software and hardware verification tasks.
- One application of SMT solvers in detection of buffer overflows will be presented.
- This work was a main part of my MSc thesis (advisor: prof. Dušan Tošić).
- The work was presented at 3rd International Conference on Software and Data Technologies (ICSOFT, Porto, 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.

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

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 goal 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 and checked by an SMT solver for linear arithmetic.

- Due to the nature of the pointer arithmetic, the theory of linear arithmetic is suitable for this purpose.

Parser and intermediate code generator
– parsing
– intermediate code generating

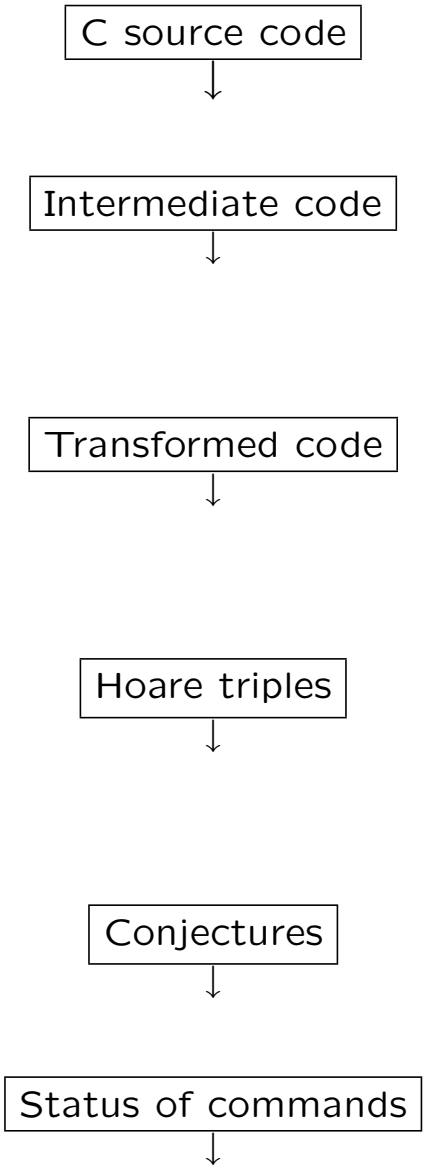
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

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

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

SMT solver 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

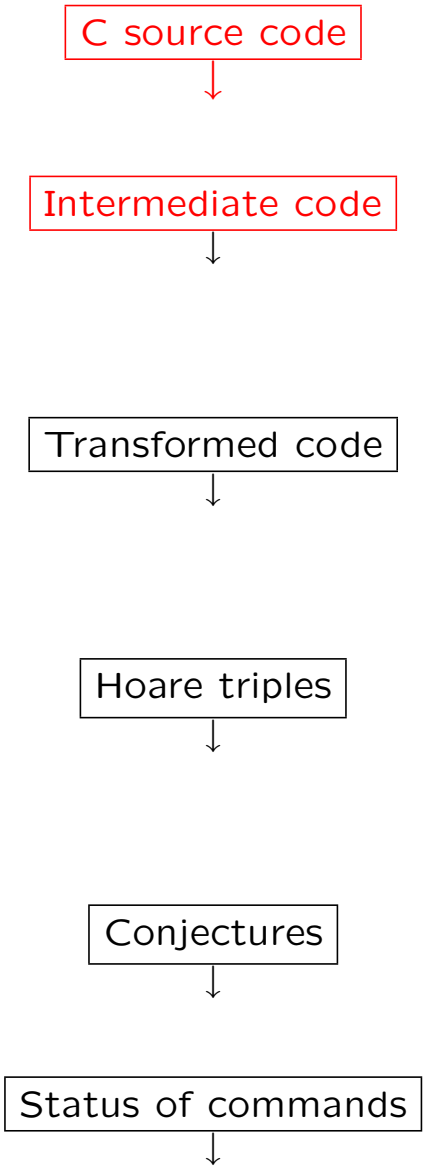
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

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

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

SMT solver 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

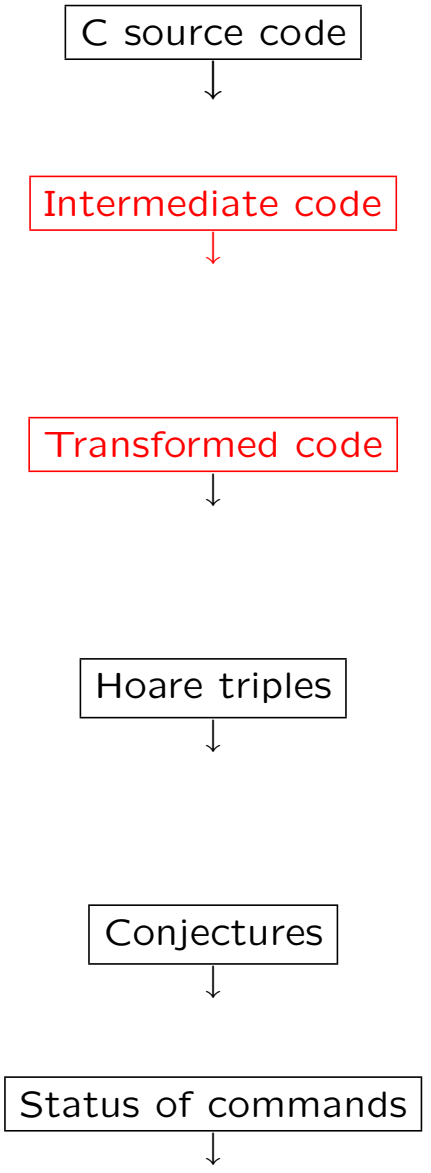
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

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

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

SMT solver 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

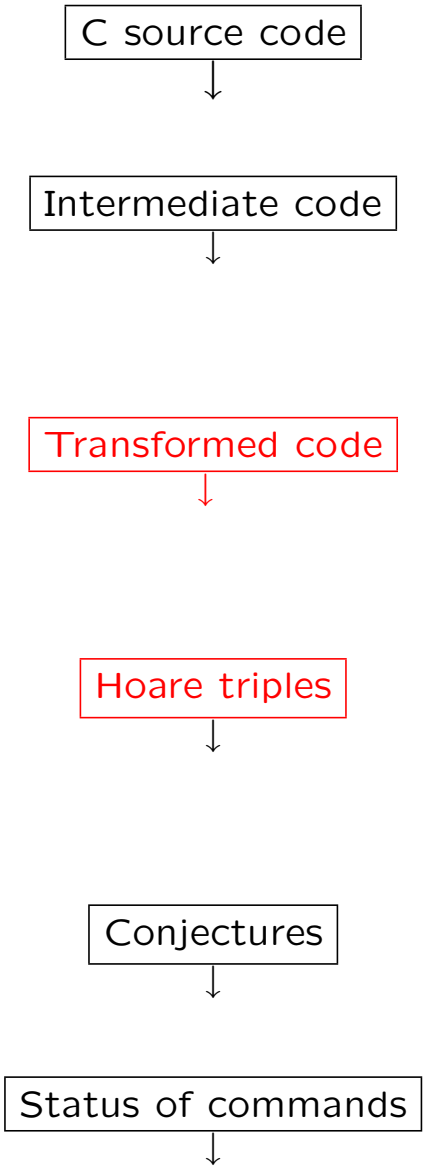
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

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

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

SMT solver 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, the user can add or remove entries. Initially, it stores reasoning rules about operators and functions from the standard C library.

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.

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

- Ground expressions are evaluated (for example, $value(10, 0)$ evaluates to 10).

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

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

- Currently, loops are processed in a limited manner — only the first iteration is considered.

Parser and intermediate code generator
– parsing
– intermediate code generating

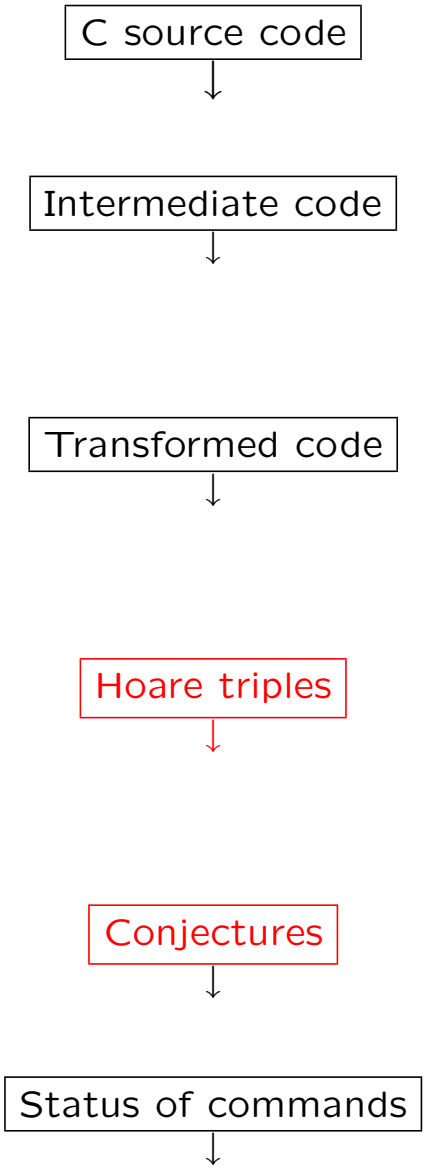
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

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

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

SMT solver 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, elimination of irrelevant conjuncts and abstraction are applied.
- Conjectures are transformed to SMT-LIB format.

Parser and intermediate code generator
– parsing
– intermediate code generating

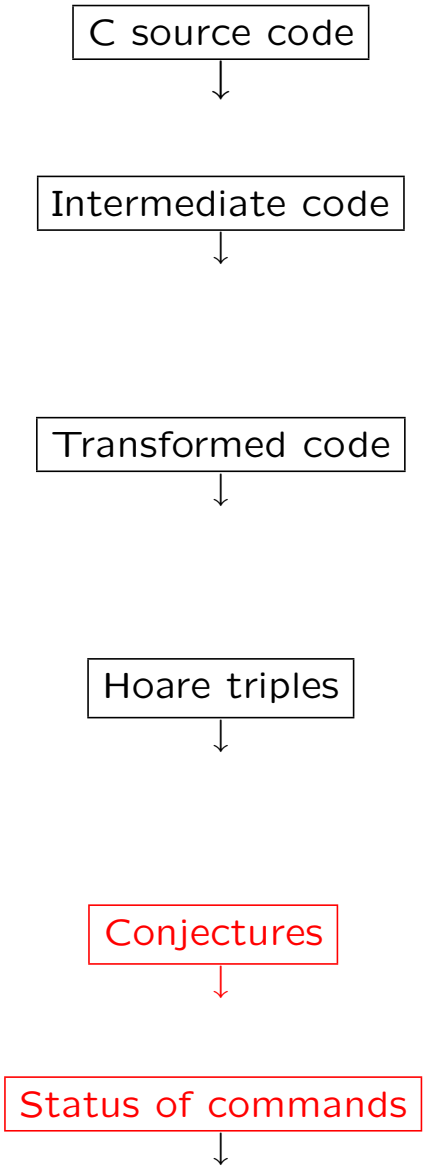
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

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

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

SMT solver 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

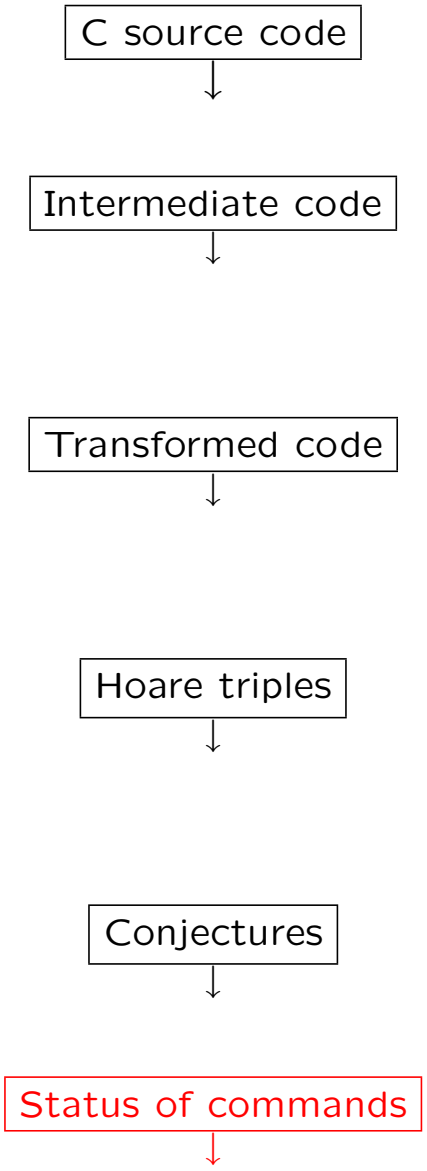
Code transformer
– eliminating multiple declarations
– reducing all loops to `do-while` loops
– eliminating all compound conditions
– etc.

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

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

SMT solver 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, after the evaluation, the correctness conjecture for the command `fgets(src,200,stdin)` is

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

There are no irrelevant conjuncts, so 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 the SMT solver which can confirm its validity. 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) is implemented in C++, it consists of $\approx$ 13000 lines of code organized in 35 classes.
- It uses two external systems:
  - JSCPP parser (developed by Jörg Schön)
  - ArgoLib (developed by Filip Marić) — SMT solver for linear arithmetic, based on the simplex method, meets SMT-LIB standards
- Modularity makes the tool very flexible: different components can be easily updated or replaced by alternatives.

FADO Tool — Experimental results

The results of experimental comparison based on 291 benchmarks used in one MIT study:

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

- Static, modular system for automated detection of buffer overflows in programs written in C is presented.
- The underlying reasoning rules are not hard-coded into the system.
- Correctness conditions are given explicitly in logical terms and checked by an external SMT solver for linear arithmetic.
- The FADO tool is a prototype implementation of the presented system, and it gives promising results.

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 SMT solvers with more expressive background theories.
  - Extend the system for other sorts of program analysis (e.g., detecting memory leaks).

Thank You for Your Attention