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


- 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 \((\text{precondition}, \text{command}, \text{post-condition})\). The semantics of a database entry \((\phi, E, \psi)\) is:
  - in order \(E\) to be safe, the condition \(\phi\) must hold;
  - in order \(E\) to be flawed, the condition \(\neg\phi\) must hold;
  - after \(E\), the condition \(\psi\) 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:

<table>
<thead>
<tr>
<th>precondition</th>
<th>command</th>
<th>postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>char x[N]</td>
<td>size(x, 1) = value(N, 0)</td>
</tr>
<tr>
<td>-</td>
<td>x = y</td>
<td>value(x, 1) = value(y, 0)</td>
</tr>
</tbody>
</table>

- For an individual command $C$, if there is a database entry $(\phi, E, \psi)$ such that there is a substitution $\sigma$ 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:

<table>
<thead>
<tr>
<th>code</th>
<th>postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a,b;</td>
<td>—</td>
</tr>
<tr>
<td>a = 1;</td>
<td>value(a, 1) = value(1, 0)</td>
</tr>
<tr>
<td>b = 2;</td>
<td>value(b, 1) = value(2, 0)</td>
</tr>
<tr>
<td>a = b;</td>
<td>value(a, 2) = value(b, 1)</td>
</tr>
</tbody>
</table>
Proposed Approach — Generating Correctness Conditions

• Postcondition for an if command are constructed as follows:

<table>
<thead>
<tr>
<th>precondition</th>
<th>command</th>
<th>postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>if(p)</td>
<td>p</td>
</tr>
<tr>
<td>−</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>precond(C1)</td>
<td>c1;</td>
<td>postcond(C1)</td>
</tr>
<tr>
<td>precond(C2)</td>
<td>c2;</td>
<td>postcond(C2)</td>
</tr>
<tr>
<td>−</td>
<td>...;</td>
<td>...</td>
</tr>
<tr>
<td>−</td>
<td>}</td>
<td>(p ∧ postcond(C1) ∧ postcond(C2)...)</td>
</tr>
</tbody>
</table>
<pre><code>                             | \(\lor(p ∧ \neg p ∧ \text{update\_states})\) |
</code></pre>

• 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 $\Phi$ be conjunction of postconditions for all commands that precede $C$. The command $C$ is:
  
  – **safe**, if $(\forall \ast)(\Phi \Rightarrow \text{precond}(C))$ is valid;
  
  – **flawed**, if $(\forall \ast)(\Phi \Rightarrow \neg\text{precond}(C))$ 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
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:
```c
char src[200];
fgets(src, 200, stdin);
```
if the database of conditions contains the following entries:

<table>
<thead>
<tr>
<th>precondition</th>
<th>command</th>
<th>postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>char x[N]</td>
<td>$size(x, 1) = value(N, 0)$ \land used(x, 1) &gt; 0</td>
</tr>
<tr>
<td>$size(x, 0) \geq value(y, 0)$</td>
<td>fgets(x, y, z)</td>
<td>$used(x, 1) \leq value(y, 0)$ \land used(x, 1) &gt; 0</td>
</tr>
</tbody>
</table>

then the following conditions are generated:

<table>
<thead>
<tr>
<th>precondition</th>
<th>command</th>
<th>postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>char src[200]</td>
<td>$size(src, 1) = value(200, 0)$ \land used(src, 1) &gt; 0</td>
</tr>
<tr>
<td>$size(src, 0) \geq value(200, 0)$</td>
<td>fgets(src, 200, stdin)</td>
<td>$used(src, 1) \leq value(200, 0)$ \land used(src, 1) &gt; 0</td>
</tr>
</tbody>
</table>
Proposed Approach — Example

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

\[(0 < used(src, 1)) \land (size(src, 1) = value(200, 0)) \Rightarrow (size(src, 1) \geq value(200, 0))\]

After evaluation, the conjecture becomes:

\[(0 < used(src, 1)) \land (size(src, 1) = 200) \Rightarrow (size(src, 1) \geq 200)\]

After abstraction, the conjecture becomes:

\[(0 < used_{src_1}) \land (size_{src_1} = 200) \Rightarrow (size_{src_1} \geq 200)\]

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** — Flexible Automated Detection of Buffer Overflows

- 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:

<table>
<thead>
<tr>
<th>Phase</th>
<th>Percent of time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing</td>
<td>1.2%</td>
</tr>
<tr>
<td>Transforming</td>
<td>0.5%</td>
</tr>
<tr>
<td>Generating conditions</td>
<td>51.8%</td>
</tr>
<tr>
<td>Exporting and testing conjectures</td>
<td>46.4%</td>
</tr>
<tr>
<td>Processing and formatting results</td>
<td>0.1%</td>
</tr>
</tbody>
</table>
The results of experimental comparison based on the mentioned corpus:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Detection rate</th>
<th>False alarm rate</th>
<th>Confusion rate</th>
<th>Average CPU time spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolySpace</td>
<td>99.7</td>
<td>0.0</td>
<td>2.4</td>
<td>172.53s</td>
</tr>
<tr>
<td>ARCHER</td>
<td>90.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.25s</td>
</tr>
<tr>
<td>FADO</td>
<td>57.0</td>
<td>6.5</td>
<td>12.5</td>
<td>0.16s</td>
</tr>
<tr>
<td>Splint</td>
<td>56.4</td>
<td>12</td>
<td>21.3</td>
<td>0.02s</td>
</tr>
<tr>
<td>UNO</td>
<td>51.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.02s</td>
</tr>
<tr>
<td>BOON</td>
<td>0.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.06s</td>
</tr>
</tbody>
</table>
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