ENSURING SAFE USAGE OF BUFFERS IN PROGRAMMING LANGUAGE C

Milena Vujosevic-Janicic
Department for Computer Science, Faculty of Mathematics, University of Belgrade, Studentski trg 16, Belgrade, Serbia
milena@matf.bg.ac.yu

Keywords: C programming language, buffer overflow, static analysis, automated bug detection

Abstract: We consider the problem of buffer overflows in C programs. This problem is very important because buffer overflows are suitable targets for security attacks and sources of serious programs' misbehavior. Buffer overflow bugs can be detected at run-time by dynamic analysis, and before run-time by static analysis. In this paper we present a new static, modular approach for automated detection of buffer overflows. Our approach is flow-sensitive and inter-procedural, and it deals with both statically and dynamically allocated buffers. Its architecture is flexible and pluggable — for instance, for checking generated correctness and incorrectness conditions, it can use any external automated theorem prover that follows SMT-LIB standards. The system uses an external and easily extendable knowledge database that stores all the reasoning rules so they are not hard-coded within the system. We also report on our prototype implementation, the FADO tool, and on its experimental results.

1 INTRODUCTION

A *buffer overflow* (or *buffer overrun*) is a programming flaw which enables storing more data in a data storage area (buffer) than it was intended to hold. This problem is important because buffer overflows are suitable targets for security attacks and source of serious programs' misbehavior.

Buffer overflows are very frequent because programming language C is inherently unsafe. Namely, array and pointer references are not automatically bounds-checked. In addition, many of the string functions from the standard C library (such as `strcpy()`, `strcat()`, `sprintf()`, `gets()`) are unsafe. Even functions dealing with bounds (such as `strncpy()`) can cause vulnerabilities when used incorrectly. Programmers often assume that calls to these functions are safe, or do the wrong checks. The consequence is that there are many applications that use the string functions unsafely. Even experienced programmers often use unsafe operations, sometimes with some checks, or relying only on the hand audits of the code being developed.

Buffer overflows are suitable targets for security attacks and they are probably the best known form of software security vulnerability. According to CERT, buffer overflows account for up to 50% of vulnerabilities, and this percentage seems to be increasing over time (Wagner et al., 2000). Attackers have managed to identify buffer overflows in a large number of products and components (Viega and McGraw, 2002). In a classic scenario for buffer overflow exploit, the attacker sends data to a program, which it stores in an undersized stack buffer. The result is that information on the call stack is overwritten, including the function's return pointer. The data sets the value of the return pointer, so when the function returns, rather than to the original caller, it transfers control to malicious code contained in the attacker's data. Unsafe operations over buffers allocated on the heap or the data segment can also be maliciously exploited but usually harder. For a survey of security attacks based on buffer overflow see, for instance, (Cowan et al., 2000).

Malicious exploiting of buffer overflows is not the only potential problem. Buffer overflow can lead to different sorts of bugs, some leading to program misbehavior, corrupted data, program crashes, and some to quite involved bugs (e.g., contents of memory reserved for one variable being rewritten only partially).

In handling and avoiding possible buffer overflows, standard testing is not sufficient, and more involved techniques are required. Because of its importance, the problem of automated detection of buffer overflows attracted a lot of attention and several techniques for handling this problem were proposed, most of them over the last ten years. Modern techniques can help in detecting bugs that were missed by hand audits. The approaches for detecting buffer overruns are divided into dynamic and static techniques. A formal account of analyzing buffer overflow problem is given in (Simon and King, 2002).

Dynamic analysis examines the program while it is being executed. These techniques can be very useful as addition to standard debugging tools. Systems based on dynamic analysis can be very successful in detecting all buffer overflows that occur during a particular execution. However, that still does not mean that there are no other bugs in other branches of the program. Some of the tools based on dynamic analysis are intended to be used during program development and testing, while some aim at preventing buffer overflows during program execution, often with serious performance overhead. For a survey and comparison of dynamic analysis tools see, for instance, (Zhivich et al., 2005; Wilander and Kamkar, 2003).

Methods based on static program analysis aim at detecting potential buffer overflows before run-time. They analyze source code and check critical commands in different ways: by simple pattern matching, or by generating and checking constraints over integer variables. A major advantage of static analysis is that bugs can be eliminated before the code is deployed. More details about tools based on static analysis are given in Section 2.

In this paper we present a new static, flow-sensitive and inter-procedural system for detecting buffer overflows, with modular architecture (and also our prototype implementation, called *Fado*). The system analyzes the code, generates correctness conditions for commands, and invokes external automated theorem prover (for linear arithmetic[1]) to test the generated conditions. The system is modular and very flexible — it is built from building blocks that can be easily changed or updated. For instance, the conditions are generated based on the external database, an external prover can be chosen among available ones (as long as it has the SMT-LIB interface[2]), etc.

2 TOOLS BASED ON STATIC ANALYSIS

There is a number of tools for detection of buffer overflows based on static analysis. Here we briefly discuss some of them, mostly those related to our approach. For an empirical comparison between different tools see (Wilander and Kamkar, 2002; Zitser et al., 2004; Kratkiewicz and Lippmann, 2005).

ARCHER (Xie et al., 2003) symbolically executes the code and performs path-sensitive, inter-procedural symbolic analysis to bound the values of both variables and memory sizes. It requires no annotations (although it can use them). For checking generated correctness conditions, ARCHER uses a custom built constraint solver (which is neither sound nor complete).

BOON (Wagner et al., 2000) generates integer range constraints for critical commands (by using a model extracted from the source code) and then checks them by a custom built range solver. The approach uses flow-insensitive and context-insensitive analysis and considers only unsafe functions from the standard C library.

CSSV (Dor et al., 2003) parses the code and transforms it to a simplified version. It requires annotating the code by preconditions and postconditions of functions. Correctness assertions are included in the output program. The tool uses a conservative static analysis algorithm to detect faulty integer manipulations. The approach is not very efficient, but is sound (with a small number of false alarms). iCSSV (Ellenbogen, 2004) is a new, more efficient, version of CSSV that is inter-procedural and does not require manual annotations.

Caduceus (Fillitre and March, 2007) is a tool for deductive verification of C code. The system requires annotations in the source code. The tool allows formally proving that a function implementation satisfies its specification and that it does not involve null pointer dereferencing or buffer overflows. It can use different theorem provers.

ITS4 (Viega et al., 2000) works on syntactical level only — it scans the source code and tries to match its fragments with critical calls stored in a special-purpose library (Viega et al., 2000). Although this approach is rather simple, it can detect many bugs

[1] *Linear arithmetic* (over reals) is a decidable fragment of arithmetic (over reals) that involves addition, but not multiplication, except multiplication by constants. Linear arithmetic is widely used in software verification.

[2] *Satisfiability modulo theory (SMT)* is a problem of deciding if a given first-order formula is satisfiable with respect to a background theory. The SMT-LIB initiative (`http://www.smt-lib.org/`, (Ranise and Tinelli, 2003)) provides a library of SMT benchmarks and all required standards and notational conventions. Most state-of-the-art SMT solvers have support for linear arithmetic and can deal with extremely complex conjectures coming from industry.

(but with many false alarms).

PolySpace (PolySpace Technologies, 2003) is a commercial tool using algorithms that are not publicly available in full detail.[3] It uses symbolic analysis, or abstract interpretation (Cousot and Cousot, 2004), escape analysis for determining inter-procedural side effects, and inter procedural alias analysis for pointers.

Splint (Larochelle and Evans, 2001) is annotation-driven tool. It uses a flow-sensitive, intra-procedural program analysis. Splint uses a lightweight static analysis and generates correctness conditions that depend on postconditions of the commands that precede a certain command. It is very efficient, but it is not sound and can produce a large number of false alarms.

UNO (Holzmann, 2002) is a tool designed to find three sorts of bugs in C programs: *u*ninitialized variables, *n*ull-pointer dereferencing, and *o*ut-of-bounds array indexing. UNO deals only with simple indexes of buffers (e.g., single constants or scalars), so it cannot check if an index which is compound expression is within bounds.

3 PROPOSED APPROACH

In this section we describe our new, static, flow-sensitive and inter-procedural, modular system for detecting buffer overflows. The code is parsed, transformed to a simpler (but equivalent) code, and analyzed line by line. For generating *preconditions* and *postconditions* (denoted *precond* and *postcond*) for individual commands, a knowledge database (or a database of conditions) is used. The database stores reasoning rules, in the form of preconditions and postconditions, for critical programming constructs and for the unsafe functions from the standard C library. Preconditions and postconditions for the user-defined functions are generated automatically in some simpler cases, while in remaining cases, the user can add them to the database (if the user fails to do that, the system can still detect bugs, but with decreased power). Preconditions and postconditions are then used for generating correctness conditions for individual commands. These conditions are then checked by an external theorem prover.

The system is built from the building blocks that can be easily changed or updated. The overall system architecture is given in Figure 1. Descriptions of the building blocks of the system in more details are given in the following subsections.

3.1 Modelling Semantics of Programs

For modelling the data-flow and semantics of programs, in formulation of the constraints we use the function *value* and two functions with pointers (i.e., buffers, allocated either statically or dynamically) as arguments, *size* and *used*:

- *value* gives a value of a given variable,
- *size* gives a number of elements allocated for the given buffer, and
- *used*, relevant only for string buffers, gives a number of elements used by the given buffer (i.e., the number of used bytes including the terminating zero).

All these functions have an additional (integer) argument called *state* or *timestamp*, capturing data-flow, i.e., the temporal nature of variables and memory space.[4] So, $value(k,0)$ gives a value of the variable k in state 0, $used(s,1)$ gives a number of elements used by the string buffer s in state 1, etc. These timestamps provide the basis for a flow sensitive analysis and a form of pointer aliasing.

3.2 Parser, Intermediate Code Generator, and Code Transformer

The parser reads code from the source files, parses it, and builds a parse tree. The parse tree is then exported to a specific intermediate code that is simpler for processing and enables easily changing the used parser.

The code transformer reads the intermediate code and performs a range of steps (e.g., eliminating multiple declarations, eliminating all compound conjunctions and disjunctions, etc.). The output of the code transformer is a program, represented via intermediate code, in a subset of C, that is equivalent to the original program, i.e., it preserves its semantics. This transformation significantly simplifies and speeds-up further processing stages. Our motivation, transformation and the target language are similar to the ones described in (Yorsh and Dor, 2003).

3.3 Database and Conditions Generator

The role of the database of conditions is in generating preconditions and postconditions of individual commands in the transformed code. Taking into account the context of the command, these preconditions and postconditions are processed and used for generating correctness conditions.

[3] There are also other related commercial tools, such as AsTree (*www.astree.ens.fr*), Parfait (*http://research.sun.com/projects/*), Coverty (*www.coverity.com/*), and CodeSonar (*www.grammatech.com/products/codesonar/*).

[4] There are no absolute values for states, instead there are relative values for each variable.

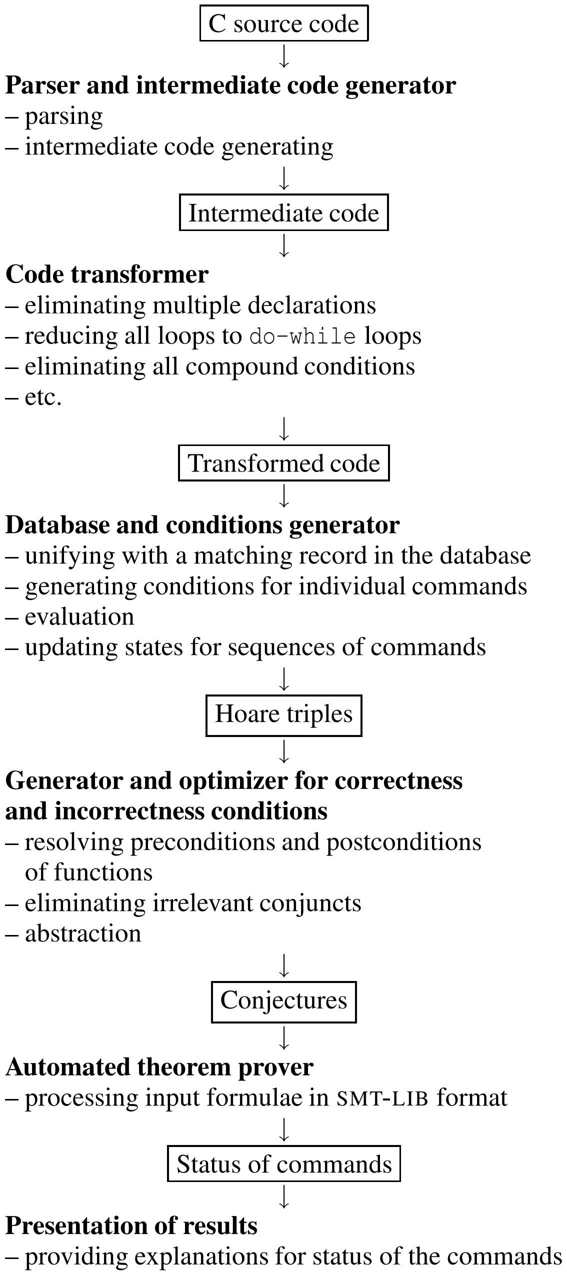
C source code
Parser and intermediate code generator
– parsing
– intermediate code generating
Intermediate code
Code transformer
– eliminating multiple declarations
– reducing all loops to do-while loops
– eliminating all compound conditions
– etc.
Transformed code
Database and conditions generator
– unifying with a matching record in the database
– generating conditions for individual commands
– evaluation
– updating states for sequences of commands
Hoare triples
Generator and optimizer for correctness and incorrectness conditions
– resolving preconditions and postconditions of functions
– eliminating irrelevant conjuncts
– abstraction
Conjectures
Automated theorem prover
– processing input formulae in SMT-LIB format
Status of commands
Presentation of results
– providing explanations for status of the commands

Figure 1: Overall system architecture

The database stores triples *(precondition, command, postcondition)*. The semantics of a database entry (ϕ, F, ψ) is as follows: in order F to be safe, the condition ϕ must hold; in order F to be flawed, the condition $\neg\phi$ must hold; after F, the condition ψ holds. These Hoare-style triples are similar to *contracts* used in CSSV system. However, these triples describe not only functions, but can describe any specific (potentially unsafe) language construct.

For example, the database may contain the following entry: $(\mathit{size}(x,0) \geq \mathit{used}(y,0)$, `strcpy(x,y)`, $\mathit{used}(x,1) = \mathit{used}(y,0))$. Note the modelling of changing of buffers via states in this example: $\mathit{size}(x,0) \geq \mathit{used}(y,0)$ says something about the state *before* executing the command (in the state 0), while $\mathit{used}(x,1) = \mathit{used}(y,0)$ says that *after* executing the command (in the state 1), x will use the same number of bytes as y before executing the command. In subsequent phases, these relative states will be transformed to absolute states within the relevant function.

The database is external and open, so can be easily changed by the user. Initially, the database stores entries related to some specific reasoning rules (e.g., `a[k]='\0'` has a precondition `value(k,0)<size(a,0)` and a postcondition `used(a,1)<=value(k,0)`). Also, the database stores information about operators and functions from the standard C library. Therefore, the underlying reasoning is not hard-coded into the system (unlike the of most other related systems). The user can add or remove specific rules, obtaining a wider scope (with higher detection rate) or a narrower scope (with less false alarms). While processing an input C program, the database may temporarily expand with entries that correspond to the user-defined functions of the program being processed.

The database is used for generating preconditions and postconditions for single commands. The precondition for a single command Φ (obtained by code transformation) is constructed as follows: if there is a database entry (ϕ, Ψ, ψ) such that Ψ matches Φ, i.e., there is a substitution σ such that[5] $\Phi = \Psi\sigma$, then $\mathit{precond}(\Phi) = \phi\sigma$. Postconditions are constructed by analogy.

When preconditions and postconditions are constructed, ground expressions are evaluated (e.g., $\mathit{size}(\text{"}\mathit{test}\text{"},.)$ is replaced by 5) and expressions involving pointer arithmetic are simplified (e.g., $\mathit{size}(a+k,.)$ is replaced by $\mathit{size}(a,.) - \mathit{value}(k,.)$, where a is a pointer, and k is an integer). This simplifies and speeds-up further processing.

After computing preconditions and postconditions for individual commands, states in functions *value*, *size*, and *used* are updated (in order to take into account the wider context). The updating is performed command by command:

- for each variable v, the initial state for *value* is 0; for each pointer variable v, the initial state for *size* is 0; for each variable v of type `char*`, the initial state for *used* is 0; since size and number of used

[5] If Ψ matches Φ, the substitution σ can be computed by *one-way matching*, a restricted form of unification in which all variables in Ψ are considered to be constants.

bytes can be changed independently, and independently of changing the value of the pointer, current states for these functions are kept separately;

- if the current state for v for *value* is S, then in the subsequent command, $value(v,0)$ is replaced by $value(v,S)$, $value(v,1)$ is replaced by $value(v,S+1)$; if $value(v,1)$ occurs in the postconditions of the command, then the current state S is incremented; the same rules are applied for *size* and for *used*;
- whenever the value of the pointer p is changed, current states for $size(p,.)$ and $used(p,.)$ are incremented (since p points to another location).

Preconditions and postconditions for commands within `if` command (`else` parts are eliminated within transformation) are constructed as follows:

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

The formula $update\_states$ stands for conjunction of conditions obtained from updating states. States within `if` commands are updated as follows: let S be the state of v for $value(v,.)$ before command `if(p)`; conditions for the commands within `if` are updated as described above, leading to the current state S'; finally, $value(v,S) = value(v,S')$ is added to $update\_states$. Thanks to this, the current state of v after the `if` construction is always S', no matter if p was true or not. This is done for all variables with states changed within `if`. States for *size* and *used* are updated by analogy. Therefore, after `if` command (like after individual commands), each variable has uniquely determined states (for *value*, *size*, and *used*), which enables further processing. In dealing with commands that follow an `if` command, the `if` command is treated as a single command and $(p \wedge postcond(C1) \wedge postcond(C2)...) \vee (\neg p \wedge update\_states)$ is used as its postcondition.

Like in some other tools, loops are processed only in a limited manner. Currently, following ideas and motivation from (Larochelle and Evans, 2001), our system tests only the first iteration of a loop (which is reasonable and sufficient in some cases), within loops covers function calls with constant arguments, and applies several other simple heuristics for dealing with commands within loops. This restriction can also limit, after a loop, detection of buffer overflows in the rest of the function.

Preconditions and postconditions for user-defined functions can be constructed automatically only in some simpler cases. For other functions, the user can add their preconditions and postconditions to the database. If the user fails to add the conditions for some user-defined function `f`, the system can still go on and potential references for $precond(f)$ and $precond(f)$ in all constraints are just abstracted and replaced by new variables (see *abstracting* step in subsection 3.4). So, our system can work (although with decreased power) even without user's annotations. Many tools based on static analysis also require annotating user-defined functions in some form, and this limitation is not critical — many (or even most) of buffer overflows do not depend on complex control flow and on inter-procedural communication, but are due to the unsafe usage of standard C functions (Larochelle and Evans, 2001).

3.4 Generator and Optimizer for Correctness and Incorrectness Conditions

For each command, correctness and incorrectness constraints are generated as follows. For a command C, let Φ be conjunction of postconditions for all commands that precede C (within its function), the command C is:

- **safe**, i.e. *it never causes an error during execution*, if $\Phi \Rightarrow precond(C)$ (universal closure is assumed) is valid;
- **flawed**, i.e. *when encountered, it always causes an error during execution*, if $\Phi \Rightarrow \neg precond(C)$ (universal closure is assumed) is valid;
- **unsafe**, i.e. *when encountered, it can cause an error during execution*, if neither of above;
- **unreachable**, i.e. *the command is not reachable*, if proved to be both safe and flawed (this happens when the preconditions that precede the command are inconsistent).

Before sending conditions to the prover, conjectures are preprocessed via the following phases:

resolving: all references to preconditions and postconditions of functions are resolved; this is done in iterations, while there are no such conditions; in the case of mutually recursive functions and in case of unresolved preconditions and postconditions for functions (both standard or user-defined), such conditions cannot be eliminated, so they are abstracted to new variables (see *abstracting* step).

eliminating irrelevant conjuncts: from a conjecture $\Phi \Rightarrow C$, all *irrelevant conjuncts* are deleted;

a conjunct from Φ is *irrelevant* for C if it is not *relevant* for C; whether a conjunct from Φ is *relevant* for C is defined recursively: a conjunct is relevant for C if it involves some variable occurring in C; also, a conjunct is relevant if it involves a variable occurring in some relevant conjunct or a function call occurring in some relevant conjunct with same arguments;

abstracting: terms that do not belong to linear arithmetic are abstracted; for instance, $size(t,2)$ is abstracted to $size\_t\_2$ (for the sake of brevity, $value(x,N)$ is abstracted to x_N); this transformation is not complete, but it is sound: if abstracted formula is valid, then the original formula is valid too, but the opposite does not hold (this means that the system is not complete for commands that involve abstracted terms from, say, abstracted unresolved function calls).

Notice that system can prove that some commands are not safe, but can also prove that some commands are safe. This feature can limit the number of false alarms — one of the main concerns for most approaches.

3.5 Automated Theorem Prover

The generated correctness conditions are checked for validity by an automated theorem prover. A theorem prover for linear fragment of arithmetic is suitable for this task as many (or most) of conditions belong to linear arithmetic (namely, pointer arithmetic is based on addition and subtraction only, so it can be well modeled by linear arithmetic).

Formulae produced by conditions generator are translated to SMT-LIB format and passed to the prover. These conjectures can be tested by any SMT solver/prover that covers linear arithmetic. The prover can check whether or not the given formula is valid (unless the time limit was exceeded), yielding an information whether a corresponding command is safe/flawed. If a command was proved to be flawed or unsafe (i.e., it was not proved to be safe), the theorem prover can generate a counterexample for the corresponding correctness conjecture, used for building a concrete example of a buffer overflow, which can be very helpful to the user.

Communication with a theorem prover is performed through external files containing conjectures in SMT-LIB format, so any prover supporting this standard can be simply plugged-in and used.

3.6 Presentation of Results

Each command carries a line number in the original source file and the prover's results are associated to these line numbers and reported to the user. The commands that are marked *flawed* cause errors in any run of the program and they must be changed. The commands that are marked *unsafe* are possible causes of errors and they also must be checked by human programmers. Explanations given by the system (in a form of a model, i.e., a set of variable values that lead to buffer overflow) can help the programmer to fix the error.

3.7 Scope

It is impossible to build a complete and sound static system for detecting buffer overflow errors (a system that detects all possible buffer overflows and has no false alarms). One of the reasons for this is undecidability of the halting problem. Our system has the following restrictions: it deals with loops in a limited manner; for computing preconditions and postconditions of user-defined functions, our system may require human's assistance (although it can also work without it); the generated conjectures belong to linear arithmetic, so the other involved theories are not considered. The system uses the prover for linear arithmetic over rational numbers, which is sound but not complete for integers (i. e., some valid conditions may not be proved). Despite the above restrictions, our system can detect many buffer overflow errors.

The power of our system is also determined by the contents of the database. The database is external and open so the user can extend it by additional reasoning rules, extending that way the scope of the tool.

3.8 Worked Example

We will illustrate the proposed approach on one simple fragment of C code (extracted from one benchmark given in (Kratkiewicz and Lippmann, 2005)). The fragment involves pointers and pointer aliasing and has one buffer overflow bug. In this simple case, the transformed code is same as the original. The code, and the preconditions and postconditions of the individual commands (based on the default contents of the database) are given in Figure 2.

After evaluation and adjustments of the states, the correctness condition for the last command is:

$$\begin{aligned} & size(buf,1) = 10 \wedge value(buf\_alias,1) = value(buf,1) \wedge \\ & size(buf\_alias,1) = size(buf,1) \wedge used(buf\_alias,1) = used(buf,1) \\ & \qquad \Rightarrow size(buf\_alias,1) > 10 \end{aligned} \tag{1}$$

This formula is not valid, so the corresponding command is not safe. Moreover, it can be similarly proved that the command is flawed — namely, the above formula with `size(buf_alias,1)>10` replaced by `size(buf_alias,1)<=10` is valid.

commands	conditions
`char *buf_alias;`	
`char buf[10];`	postcondition: `size(buf,1)=value(10,0)`

`buf_alias=buf;`	postcondition:
	`value(buf_alias,1)=value(buf,0)`
	`and size(buf_alias,1)=size(buf,0)`
	`and used(buf_alias,1)=used(buf,0)`

`buf_alias[10]='A';`	precondition: `size(buf_alias,0)>value(10,0)`

Figure 2: Example code with conditions for the individual commands

4 PROTOTYPE IMPLEMENTATION AND RESULTS

We have made a prototype implementation of the approach presented in Section 3. The implemented system is called *Fado* (from *F*lexible *A*utomated *D*etection of buffer *O*verflows).[6]

The Fado tool is implemented in standard C++. It consists of around 13000 lines of code organized in 35 classes. Fado uses the parser `JSCPP`[7] and the simplex-based solver for linear arithmetic (Dutertre and De Moura, 2006) from ARGO-LIB.[8]

Architecture of the implementation of the system is very flexible: the mentioned parser can be replaced by some other parser, the implementation of the simplex method can also be easily replaced by some other implementation. Moreover, the simplex method can be replaced by some other proving method with a wider domain (thus leading to a more powerful overall system).

Formulae produced by conditions generator are translated to SMT-LIB format and passed to the ArgoLib prover. Since external files are used for communication, it is possible to use any theorem prover that can parse SMT-LIB format. The system could be more efficient if the ArgoLib API was used for communication instead of using external SMT-LIB files, but this would reduce the flexibility of the system because theorem prover could not be changed.

The implementation has been successfully tested on a range of examples. Here we present evaluation results obtained on the benchmarks from (Kratkiewicz and Lippmann, 2005) (one of the very few benchmark sets used for evaluation of more static analysis tools). On these benchmarks, our system detected (without any human intervention or annotations) 57% of the flaws. From the remaining flaws, around 35% are due to the loops that cannot be currently processed, but could be handled with several heuristics. Around 3% flaws were not detected because the current implementation still does not cover some programming constructs. The remaining 5% flaws are substantially beyond the reach of our system. Over this set of benchmarks, the false alarm rate was 6.5% and the confusion rate was 12.5%. With human-annotations used, the false alarm rate was 3% and the confusion rate was 6%. The remaining false alarms were due to the incomplete coverage of the C language in the prototype implementation — 1.5%, and to inherent limitations of the approach — 2%.

Tool	Detection Rate	False Alarm Rate	Confusion Rate	avg.CPU time spent
PolySpace	99.7	2.4	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	12.0	21.3	0.02s
UNO	51.9	0.0	0.0	0.02s
BOON	0.7	0.0	0.0	0.06s

Figure 3: Experimental results of tools ARCHER, BOON, Splint, UNO i PolySpace C Verifier according to the corpus from (Kratkiewicz and Lippmann, 2005), with results of FADO added

According to (Kratkiewicz and Lippmann, 2005), these results position the current version of our system behind Archer and PolySpace, and before UNO and BOON. Our system's results are similar to the results of Splint, but with better false alarm rate and confusion rate.

Concerning the execution time, our tool processed 291 examples in 46.8s (experiments were run on PC 2.4GHz). The times spent by different phases were: 1.2% for parsing the code, 0.5% for code transformation, 51.8% for generating conditions, 46.4% for exporting to SMT files and for checking conditions, and 0.1% for processing and formatting results. The execution time of our tool is comparable to the results of other tools given in (Kratkiewicz and Lippmann, 2005) (although this source does not specify the computer used for testing). A summary of results is given in Figure 3.

[6] Fado is available upon request from the first author.

[7] Developed by Jörg Schön, available from `http://www.die-schoens.de/prg`

[8] ARGO-LIB, developed by Filip Marić, is a generic platform for using decision procedures. It is available at `http://argo.matf.bg.ac.yu`. Concerning the background logic, underlying theories, description of theories, and format, ARGO-LIB follows the SMT-LIB initiative (Ranise and Tinelli, 2003).

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented a new, modular system for automated detection of buffer overflows in programs written in C. Our system performs flow-sensitive and inter-procedural static analysis. The system generates correctness and incorrectness conditions for individual commands, which are then tested for validity by an automated theorem prover for linear arithmetic. Some of the main novelties and advantages of our system 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 so the user can vary them), buffer overflow correctness conditions given explicitly in logical terms, using external theorem provers. We presented the current prototype implementation of the proposed system, the Fado tool, that gives promising results.

The presented system is a subject of further, more detailed evaluation, improvements, and development. For instance, although heuristics for dealing with loops are efficient and can have a wide range, for the next stage of development, we are planning to extend our system to preform full analysis of loops (in a similar manner as proposed in some modern systems (Dor et al., 2003)) and of user-defined functions, so the system will be sound and its inter-procedural analysis will be fully automatic. In addition, we are planning to use theorem provers with more expressive background theories. Our goal is to make Fado efficiently applicable to long, real-world critical programs. Thanks to the tool's flexible architecture, we are also planning to extend it for other sorts of program analysis (e.g., testing for memory leaks).

REFERENCES

P. Cousot and R. Cousot. (2004) Basic Concepts of Abstract Interpretation. In *Building the Information Society*. Kluwer, 2004.

Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J. (2000). Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conf. and Expo*.

Dor, N., Rodeh, M., and Sagiv, M. (2003). Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press.

Dutertre, B. and De Moura, L. (2006). A fast linear-arithmetic solver for dpll(t). In *CAV 2006*, vol. 4144 of *LNCS*. Springer.

Ellenbogen, R. (2004). Fully automatic verification of absence of errors via interprocedural integer analysis. Master's thesis, University of Tel-Aviv, Israel.

Fillitre, J.-C. and March, C. (2007). The why/-krakatoa/caduceus platform for deductive program verification. In *CAV*, vol. 4590 of *LNCS*. Springer.

Holzmann, G. (2002). Static source code checking for user-defined properties. In *Proceedings of 6th World Conference on Integrated Design and Process Technology*.

Kratkiewicz, K. and Lippmann, R. (2005). Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. In *Workshop on the Evaluation of Software Defect Detection Tools*.

Larochelle, D. and Evans, D. (2001). Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*.

PolySpace Technologies (2003). Polyspace c verifier. Paris, France. http://www.polyspace.com.

Ranise, S. and Tinelli, C. (2003). The SMT-LIB Format: An Initial Proposal. on-line at: http://goedel.cs.uiowa.edu/smt-lib/.

Simon, A. and King, A. (2002). Analyzing String Buffers in C. In *International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *LNCS*. Springer.

Viega, J., Bloch, J., Kohno, Y., and McGraw, G. (2000). Its4: A static vulnerability scanner for c and c++ code. In *16th Annual Computer Security Applications Conference (ACSAC'00)*.

Viega, J. and McGraw, G. (2002). *Building Secure Software*. Addison-Wesley.

Wagner, D., Foster, J., Brewer, E., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed System Security*.

Wilander, J. and Kamkar, M. (2002). A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (Nordsec 2002)*.

Wilander, J. and Kamkar, M. (2003). A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*.

Xie, Y., Chou, A., and Engler, D. (2003). Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference*. ACM Press.

Yorsh, G. and Dor, N. (2003). The Design of CoreC. on-line at: http://www.cs.tau.ac.il/ gretay/GFC.htm.

Zhivich, M., Leek, T., and Lippmann, R. (2005). Dynamic buffer overflow detection. In *Workshop on the Evaluation of Software Defect Detection Tools*.

Zitser, M., Lippmann, R., and Leek, T. (2004). Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM.