Using Simplex Method in Verifying Software Safety*

Milena Vujošević-Janičić
milena@matf.bg.ac.yu

Filip Marić
filip@matf.bg.ac.yu

Dušan Tošić
dtosic@matf.bg.ac.yu
Faculty of Mathematics, University of Belgrade,
Studentski trg 16,11000 Belgrade, Serbia

Abstract

In this paper we discuss an application of the Simplex method in checking software safety — the application in automated detection of buffer overflows in C programs. This problem is important because buffer overflows are suitable targets for hackers' security attacks and sources of serious programs' misbehavior. We also describe our implementation, including a system for generating software correctness conditions and a Simplex -based theorem prover that resolves these conditions.

1 Introduction

The Simplex method is considered to be one of the most significant algorithms of the last century.[1] It is a method for solving the linear optimization problem [4] and its worst case complexity is exponential in the number of variables [11]. However, it is very efficient in practice and converges in polynomial time for many input problems, including certain classes of randomly generated problems ([17], [9]). Apart from the basic Simplex method for the optimization problem, there are many other variants, including a decision variant that decides if a set of linear constraints is satisfiable or not.

The Simplex method has a wide range of applications, in different sorts of optimization problems, but also in software and hardware verification. In this paper, we describe how a decision version of the Simplex method can be used in automated detection of buffer overflows in programming language C. *Buffer overflow* (or *buffer overrun*) is a programming flaw which enables storing more

*This work was partially supported by Serbian Ministry of Science grant 144030.

[1]For instance, the journal *Computing in Science and Engineering* listed it as one of the top 10 algorithms of the century.

data in a data storage area (buffer) than it was intended to hold. This shortcoming can produce many problems. Namely, buffer overflows are suitable targets for breaking security of programs and sources of serious programs' misbehavior.

Further in this paper, in Section 2 we give background information, in Section 3 we describe one decision variant of the Simplex method and our implementation, and in Section 4 we present our technique for automated detection of buffer overflows, that uses the mentioned implementation. In Section 5 we briefly discuss related work and in Section 6 we draw final conclusions and discuss the future work.

2 Background

Linear programming. *Linear programming*, sometimes known as linear optimization, is the problem of maximizing or minimizing a linear function over a convex polyhedron specified by linear and non-negativity constraints. A linear programming problem consists of a collection of linear inequalities on a number of real variables and a given linear function (on these real variables) to be maximized or minimized. A linear programming problem, in its standard form, is to maximize function given by $c^t x$ with regards to constraints of the type $Ax \leq b$ where $b \geq 0$, $x \geq 0$, x, b and c are vectors from $\mathbb{R}^n$, and A is a real $m \times n$ matrix.

Linear Arithmetic. *Linear arithmetic* (over rationals (LRA) or integers (LIA)) is a fragment of arithmetic (over rationals or integers) involving addition, but not multiplication, except multiplication by constants. A quantifier-free linear arithmetic formula is a first-order formula whose atoms are equalities, disequalities, or inequalities of the form $a_1x_1 + \ldots + a_nx_n \bowtie b$, where $a_1, \ldots, a_n$ and b are rational numbers, $x_1, \ldots, x_n$ are (rational or integer) variables, and $\bowtie$ is one of the operators $=$, $\leq$, $<$, $>$, $\geq$, or $\neq$.

Linear arithmetic (both over rationals and integers) is decidable (i.e., there is a *decision procedure*, returning *true* if and only if an input linear arithmetic sentence Φ is a theorem, and returning *false* otherwise)). Two most popular methods for deciding satisfiability of linear arithmetic formulae are Fourier-Motzkin procedure [14] and the Simplex method [7]. Linear arithmetic is widely used in software verification, especially its quantifier-free fragment, because it can model many types of constraints, and it is decidable. Decision procedures for LRA are much faster then decision procedures for LIA.

Simplex method. The Simplex method is originally constructed to solve linear programming optimization problem, but its variants can be used to solve the decision problem for quantifier-free fragment of linear arithmetic. The method iteratively finds *feasible solutions* satisfying all the given constraints, while greedily tries to maximize the objective function.

In geometric terms, a series of linear inequalities defines a closed convex polytope (called simplex), defined by intersecting a number of half-spaces in

n-dimensional Euclidean space; each half-space is an area which lies on one side of a hyperplane. The Simplex algorithm begins at a starting vertex and moves along the edges of the polytope until it reaches the vertex of the optimum solution. At every iteration an adjacent vertex is chosen such that the value of the objective function does not decrease. If no such vertex exists, a solution to the problem is found. Usually, such an adjacent vertex is not unique, and a *pivot rule* must be specified to determine which vertex to pick. There are various pivot rules used in practice.

The decision problem for linear arithmetic reduces to finding a single feasible solution. The basic Simplex method can be modified to cover some other, different types of constraints than those used in standard linear programming optimization problem (e.g., some variables x_i might be unconstrained, some coefficients b_i might be negative, a minimal solution instead of maximal one might be requested). The dual Simplex algorithm [15] is quite effective when constraints are added incrementally. This algorithm is particularly useful for reoptimizing the problem after a constraint has been added or some parameters have been changed so that the previously optimal solution is no longer feasible.

SMT. *Satisfiability Modulo Theories* (SMT) solvers check satisfiability of Boolean combination of constraints formulated in some first-order theory or combination of several such theories. SMT solving has many industrial applications, especially in software and hardware verification. Some of the interesting background theories for different applications are linear arithmetic, theory of uninterpreted functions, and theories of program structures like arrays and recursive structures. Most state-of-the-art SMT solvers have support for linear arithmetic and can deal with extremely complex conjectures coming from industry. In these cases the decision procedures are usually based on the Simplex method.

The SMT-lib initiative[2] is aimed at producing a library of SMT benchmarks and all required standards and notational conventions [18], linking a range of SMT solvers and research groups. In SMT-lib, the underlying logic is classical first order logic with equality.

Buffer Overflow Bug Buffer overflow, i.e., writing outside the bounds of a block of allocated memory, can lead to different sorts of bugs and can provide possibility to an execution of malicious code. According to some estimates, buffer overflows account for up to 50% of software vulnerabilities, and this percent seems to be increasing over time [22]. In particular, buffer overflow is probably the best known form of software security vulnerability. Attackers have managed to identify and exploit buffer overflows in a large number of products and components [21, 3].

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

[2] http://www.smt-lib.org/

library (such as `strcpy()`, `strcat()`, `sprintf()`, `gets()`) are unsafe. Programmers often assume that calls to these functions are safe, or do the inadequate checks. The consequence is that there are many applications using the string functions unsafely.

In handling and avoiding possible buffer overflows, standard testing is not sufficient, and more involved techniques are required. 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 missed by hand audits. The approaches for detecting buffer overruns are divided into dynamic and static techniques. Dynamic techniques examine the program during its execution. Methods based on static program analysis aim at detecting potential buffer overflows before run-time and their major advantage is that bugs can be found and eliminated before code is deployed.

3 Simplex-based SMT Solving

In this section we will describe basics of a DPLL(T) framework for SMT, and then present a Simplex-based decision procedure for Linear Arithmetic (over rationals) designed to fit within the DPLL(T) framework.

ArgoLib is an SMT solver based on DPLL(T) framework and developed by the Automatic Reasoning GrOup at Faculty of Mathematics in Belgrade.[3] Among several supported theories, ArgoLib contains a solver for the theory of Linear Arithmetic over rationals (LRA), based on the Simplex method implementation described in Section 3.2.

3.1 DPLL(T)

Amongst a plethora of recent research on satisfiability modulo theory, the *DPLL(T)* framework [16] has proven to be very successful. Within this framework, an SMT solver consists of two separated components:

1. $DPLL(X)$ — a Boolean satisfiability solver based on a slightly modified variant of Davis-Putnam-Logeman-Loveland (DPLL) algorithm [5].

2. $Solver_T$ — a solver for the given theory T capable to check the consistency of conjunctions of atomic formulae from T.

These two components have to cooperate during the solving process. $DPLL(X)$ is parameterized with $Solver_T$, giving a $DPLL(T)$ solver. A given formula Φ of the theory T is transformed into a Boolean formula Φ_{bool} by replacing its atoms $\phi_1, \ldots, \phi_k$ with fresh propositional variables $p_1, \ldots, p_k$. The role of the $DPLL(X)$ component is to find and enumerate propositional models of the formula Φ_{bool}. Each propositional model M induces a conjunction of atoms $\Phi_T^M = \bigwedge_{i=1}^{|M|} \psi_i$, such that $\psi_i = \phi_i$ if $p_i \in M$ or $\psi_i = \neg\phi_i$ if $\neg p_i \in M$. The

[3] ArgoLib is being developed by the second author of this paper.

role of the $Solver_T$ component is to check consistency of conjunctions Φ_T^M, with respect to the background theory T. The formula Φ is satisfiable if and only if there is a propositional model M satisfying Φ_{bool} such that its corresponding formula Φ_T^M is consistent with the theory T.

Example 1 *Let us consider the formula* $\Phi \equiv (x + y > 0 \ \wedge \ x < 0) \ \vee \ y < 0$ *(implicitly existentially quantified) with respect to the theory of linear arithmetic over rationals. The atoms* $\phi_1 \equiv x + y > 0$, $\phi_2 \equiv x < 0$ *and* $\phi_3 \equiv y < 0$, *are abstracted with propositional variables* p_1, p_2 *and* p_3 *respectively and the corresponding Boolean formula* Φ_{bool} *is* $(p_1 \wedge p_2) \vee p_3$. *The model* $M_1 = \{p_1, p_2, p_3\}$ *for* Φ_{bool} *induces the formula* $\Phi_{LRA}^{M_1} \equiv x + y > 0 \ \wedge \ x < 0 \ \wedge \ y < 0$, *which is inconsistent in linear arithmetic. On the other hand, the model* $M_2 = \{p_1, p_2, \neg p_3\}$ *for* Φ_{bool} *induces the formula* $\Phi_{LRA}^{M_2} \equiv x + y > 0 \ \wedge \ x < 0 \ \wedge \ y \geq 0$ *which is consistent in linear arithmetic and, therefore, the formula* Φ *is satisfiable.*

The DPLL(X) component based on DPLL search algorithm builds propositional models incrementally, starting from an empty valuation, and *asserting* literals one-by-one until all variables become assigned, or until it shows that formula has no propositional models. In order to obtain better efficiency, propositional models are not only checked against theory T a posteriori i.e., when they are completely constructed, but also, partial propositional models are checked during the Boolean search process. Therefore, $Solver_T$ should be *incremental*, i.e., once it has found a conjunction of atoms consistent, it has to be able to check the consistency of that conjunction extended with additional atom(s), without having to redo all the previous work. In order to achieve this, $Solver_T$ maintains a state consisting of atoms corresponding to propositions asserted so far by $DPLL(X)$. As the search progresses, new literals are asserted and their corresponding atoms are given to $Solver_T$ which then checks the consistency of its state. When inconsistency is detected, the $DPLL(X)$ module is notified about it.Then, it *backtracks* and removes some asserted literals and their corresponding atoms until a consistent state is restored. Literals and their corresponding atoms are asserted and backtracked in LIFO fashion.

When inconsistency of Φ_T^M is detected, it usually comes from a subset of atoms that have been asserted. $Solver_T$ should be able to generate a (preferably small) inconsistent subset of Φ_T^M. This set is called the *explanation* for inconsistency of Φ_T^M and it helps the Boolean search engine $DPLL(X)$ to reject some Boolean models that could induce the same inconsistent core again.

$Solver_T$ should be able also to infer which atoms (and their corresponding propositions) have to hold as a consequence of its current state. This is called the *theory propagation* and it can significantly speed up the search, since the information from the background theory T is used to guide the Boolean search process.

3.2 Simplex-based Solver for LRA

We now describe a $Solver_{LRA}$ based on specific variant of dual Simplex method developed by Duterte and de Moura and used in their SMT solver YICES [8]. This procedure consists of a preprocessing phase and a solving phase.

Preprocessing. The first step of the procedure is to rewrite the formula Φ into an equisatisfiable formula $\Phi_= \ \wedge \ \Phi'$, where $\Phi_=$ is a conjunction of linear equalities and Φ' is an arbitrary Boolean formula in which all atoms occurring in Φ' are *elementary atoms* of the form $x_i \bowtie b$, where x_i is a variable and b is a rational constant. This transformation is straightforward, and it introduces a new variable s_i for every linear term t_i that is not a variable and that occurs as a left-hand side of an atom $t_i \bowtie b$ of Φ.

Example 2 *If* Φ *is* $x \geq 0 \wedge x+y < 0 \wedge 2x+3y > 1$, Φ' *is* $x \geq 0 \wedge s_1 < 0 \wedge s_2 > 1$, *and* $\Phi_=$ *is* $s_1 = x + y \wedge s_2 = 2x + 3y$.

In the next preprocessing step, all disequalities of the form $x \neq b$ are rewritten to $x < b \ \vee \ x > b$. Then, each strict inequality of the form $x < b$ is replaced by $x \leq b-\delta$, where δ has a role of a *sufficiently small* rational number. Similarly, each $x > b$ is replaced with $x \geq b + \delta$. This enables us to assume that there are no strict inequalities in Φ'.

Example 3 *After the second preprocessing step, the formula* Φ' *from Example 2 becomes* $x \geq 0 \ \wedge \ s_1 \leq -\delta \ \wedge \ s_2 \geq 1 + \delta$.

The number δ is not computed in advance, it is treated symbolically, and its effective computation is done only when a concrete, rational model of the formula that is found to be satisfiable over $\mathbb{Q}$ is requested. This means that after the preprocessing phase, all computations are performed in the field $\mathbb{Q}_\delta$, where Q_δ is the set $\{a + b\,\delta \mid a, b \in Q\}$. While addition and multiplication of elements of Q_δ is trivial, comparison of Q_δ elements is defined in the following way: $a_1 + b_1\,\delta \bowtie a_2 + b_2\,\delta$ if and only if $a_1 \bowtie a_2 \vee (a_1 = a_2 \wedge b_1 \bowtie b_2)$, where $\bowtie \in \{\leq, \geq\}$. It can be shown that the original formula is satisfiable over $\mathbb{Q}$ if and only if the transformed formula is satisfiable over $\mathbb{Q}_\delta$. For more details of this subject see [8].

Incremental Simplex Algorithm The formula $\Phi_=$ is a conjunction of equalities and it does not change during the search process, so it can be given to Simplex solver before the model search begins. Let $x_1, \ldots, x_n$ be all variables occurring in $\Phi_= \wedge \Phi'$ (that is, all variables from Φ and m additional variables $s_1, \ldots, s_m$). If all variables are put on the left hand sides, the formula $\Phi_=$ can be represented in matrix form as $Ax = 0$, where A is a matrix $m \times n$, $m \leq n$, and x is a vector of n variables. Instead of that, we will keep this system of equations in a form solved for m variables, i.e., in a tableau derived from the matrix A, written in the form:

$$x_i \quad = \quad \sum_{x_j \in \mathcal{N}} a_{ij} x_j, \qquad x_i \in \mathcal{B}.$$

The variables on the left hand side will be called *basic variables*, and variables on the right hand side will be called *non-basic variables*. We will denote the current set of basic variables by $\mathcal{B}$ and the current set of non-basic variables by $\mathcal{N}$. Basic variables do not occur on the right hand side of the tableau. Initially, only the additional variables will be the basic variables.

On the other hand, formula Φ' is an arbitrary Boolean combination of elementary atoms of the form $x_i \bowtie b$, where $b \in \mathbb{Q}_\delta$. As said in Section 3.1, the Boolean structure is handled by a separate $DPLL(X)$ component, so the Simplex solver needs to be able to check consistency only of conjunctions of elementary atoms of Φ' (where elementary atoms are asserted and backtracked one by one). Because of their special structure ($x \leq u$ or $x \geq l$), the conjunction of asserted elementary atoms determines lower and upper bounds for variables.

Therefore, Φ is consistent if there is $x \in \mathbb{Q}_\delta^n$ satisfying

$$Ax = 0 \quad \text{and} \quad l_j \leq x_j \leq u_j \text{ for } j = 1, \ldots, n,$$

where l_j is an element of $\mathbb{Q}_\delta$ or $-\infty$ and u_j is an element of $\mathbb{Q}_\delta$ or $+\infty$. The solver state includes:

1. A tableau derived from the formula $\Phi_=$, written in the form:

$$x_i \quad = \quad \sum_{x_j \in \mathcal{N}} a_{ij} x_j, \qquad x_i \in \mathcal{B}.$$

2. The known upper and lower bounds l_i and u_i for every variable x_i, derived from asserted atoms of Φ'.

3. The current valuation, i.e., a mapping β assigning a value $\beta(x_i) \in \mathbb{Q}_\delta$ to every variable x_i.

Initially, all lower bounds are set to $-\infty$, all upper bounds are set to $+\infty$, and β assigns zero to each variable x_i.

The main invariant of the algorithm (the property that holds after each step) is that β always satisfies the tableau i.e., $A\beta(x) = 0$ and β always satisfies the bounds i.e., $\forall x_j \in \mathcal{B} \cup \mathcal{N},\ l_j \leq \beta(x_j) \leq u_j$.

When a new elementary atom is asserted, the solver state is updated. Since disequalities and strict inequalities are removed in the preprocessing phase, only equalities and non-strict inequalities are asserted.

Instead of equality $x_i = b$, two inequalities $x_i \leq b$ and $x_i \geq b$ are asserted.

After asserting inequality $x_i \leq b$ (assertion of inequality of $x_i \geq b$ is handled in a similar way), the value b is compared with the current bounds for x_i and bounds are updated:

- If b is greater then u_i, the inequality $x_i \leq b$ does not introduce any new information and state is not changed.

- If b is less then l_i, then the state becomes inconsistent and unsatisfiability is detected.

- In other cases, the upper bound u_i for the variable x_i is decreased and set to b.

If x_i is non-basic variable (i.e., when $x_i \in \mathcal{N}$), and when its value $\beta(x_i)$ does not satisfy the updated bounds l_i or u_i, its value has to be updated. If it holds that $\beta(x_i) > u_i$ (the case $\beta(x_i) < l_i$ is handled in a similar way), the value $\beta(x_i)$ is decreased and set to u_i. With every change of the value of a non-basic variable, the values of basic variables need to be updated in order to keep the tableau satisfied.

The problem arises if x_i is a basic variable (i.e., when $x_i \in \mathcal{B}$), and when its value $\beta(x_i)$ does not satisfy its bounds l_i or u_i. If it holds that $\beta(x_i) > u_i$ (the case $\beta(x_i) < l_i$ is handled in a similar way), the value $\beta(x_i)$ has to be decreased and set to u_i. In order for the tableau equation $x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j$ to remain valid, there must exist a non-basic variable x_j such that its value $\beta(x_j)$ can be decreased (if for its corresponding coefficient a_{ij} it holds that $a_{ij} > 0$) or increased (if for its corresponding coefficient a_{ij} it holds that $a_{ij} < 0$). If there is no non-basic variable x_j allowing this kind of change (because all values are already set to their lower/upper bounds), the state is inconsistent and unsatisfiability is detected. If a non-basic variable x_j that allows this kind of change is found, the *pivoting operation* is performed. The equation $x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j$ is solved for x_j and the variable x_j is then substituted in every other equation of the tableau. Therefore, x_j becomes a basic variable, and x_i becomes a non-basic variable so its value can be set to u_i. Still, this can cause bound violation for some other basic variables, and the process should be iteratively performed until all variables satisfy their bounds, or until inconsistency is detected. A variant of Bland's rule [2] which relies on a fixed variable ordering can be used to ensure termination of this process.

In this variant of the Simplex method, during backtracking, only the bounds have to be changed, while the valuation and tableau can remain the same and no pivoting is requested. This feature is very important.

Explanations for inconsistencies are generated from the bounds of variables occurring in the equation that has become violated. For more details about generating explanations and performing theory propagation see [8].

Implementation of the described algorithm is given in Figure 1. The procedure `assert` is invoked by the $DPLL(X)$ component whenever an atom $x_i \bowtie b$ is asserted. This procedure automatically checks and updates bounds and values for non-basic variables, since this operation is cheap and does not require pivoting. The procedure `check` is used to check bounds and update values for all basic variables. It loops in an infinite loop and iteratively changes the valuation using pivoting until all bounds are satisfied, or an inconsistency is detected. Changing the value of a basic variable can be quite expensive, and the procedure `check` should be invoked only from time to time. This could delay the detection of inconsistency, but usually gives better overall performance. Procedures `update` and `pivotAndUpdate` are auxiliary.

```
procedure assert(x_i ⋈ b)
        if (⋈ is =) then
                assert(x_i ≤ b)
                assert(x_i ≥ b)
        else if (⋈ is ≤) then
                if (b ≥ u_i) then return satisfiable
                if (b < l_i) then return unsatisfiable
                u_i := b
                if (x_i ∈ 𝒩 and β(x_i) > b) then
                        update(x_i, b)
        else if (⋈ is ≥) then
                if (b ≤ l_i) then return satisfiable
                if (b > u_i) then return unsatisfiable
                l_i := b
                if (x_i ∈ 𝒩 and β(x_i) < b)
                        update(x_i, b)

procedure check()
loop
        select the smallest x_i ∈ ℬ such that β(x_i) < l_i or beta(x_i) > u_i
        if there is no such x_i then return satisfiable
        if β(x_i) < l_i then
                select the smallest x_j ∈ 𝒩 such that
                        (a_ij > 0 and β(x_j) < u_j) or (a_ij < 0 and β(x_j) > l_j)
                if there is no such x_j then return unsatisfiable
                pivotAndUpdate(x_i, l_i, x_j)
        if β(x_i) > u_i then
                select the smallest x_j ∈ 𝒩 such that
                        (a_ij < 0 and β(x_j) < u_j) or (a_ij > 0 and β(x_j) > l_j)
                if there is no such x_j then return unsatisfiable
                pivotAndUpdate(x_i, u_i, x_j)
end loop

procedure update(x_i, v)
        for each x_j ∈ ℬ
                β(x_j) := β(x_j) + a_ji(v − β(x_i))
        β(x_i) := v

procedure pivotAndUpdate(x_i, v, x_j)
        θ := (v − β(x_i)) / a_ij
        β(x_i) := v
        β(x_j) := β(x_j) + θ
        for each x_k ∈ ℬ \ {x_i}
                β(x_k) := β(x_k) + a_kj θ
        pivot(x_i, x_j)
```

Figure 1: Implementation of a decision variant of the Simplex method.

Example 4 *Let us check the satisfiability of the conjunction*

$$x \geq 1 \ \wedge \ y \leq 1 \ \wedge \ x + y \leq 0 \ \wedge \ y - x \geq 0.$$

After the initial transformation, the tableau becomes:

$$\begin{aligned} s_1 &= x + y \\ s_2 &= -x + y \end{aligned}$$

and $\mathcal{B} = \{s_1, s_2\}$, $\mathcal{N} = \{x, y\}$. *The formula* Φ' *is* $x \geq 1 \ \wedge \ y \leq 1 \ \wedge \ s_1 \leq 0 \ \wedge \ s_2 \geq 0$.

The initial valuation is $\beta(x) = 0, \beta(y) = 0, \beta(s_1) = 0, \beta(s_2) = 0$, *and the initial bounds are* $-\infty \leq x \leq +\infty$, $-\infty \leq y \leq +\infty$, $-\infty \leq s_1 \leq +\infty$, $-\infty \leq s_2 \leq +\infty$.

When $x \geq 1$ *is asserted, the bounds for* x *become* $1 \leq x \leq +\infty$, *and the valuation becomes* $\beta(x) = 1, \beta(y) = 0, \beta(s_1) = 1, \beta(s_2) = -1$. *No pivoting is performed.*

When $y \leq 1$ *is asserted, the bounds for* y *become* $-\infty \leq y \leq 1$, *and the valuation is not changed since* y *satisfies new bounds. No pivoting is performed.*

When $s_1 \leq 0$ *is asserted, the bounds for* s_1 *become* $-\infty \leq s_1 \leq 0$. *The value* $\beta(s_1) = 1$ *violates this bound, and* $\beta(s_1)$ *has to be decreased to* 0. *Since* s_1 *is a basic variable, pivoting has to be performed. The value of* x *is already on its lower bound so it cannot get decreased. The value of* y *can be decreased, so* y *is chosen to be the pivot variable. After pivoting, the tableau becomes:*

$$\begin{aligned} y &= s_1 - x \\ s_2 &= -2x + s_1 \end{aligned}$$

and y *becomes a basic, and* s_1 *becomes a non-basic variable. The updated valuation becomes* $\beta(x) = 1, \beta(y) = -1, \beta(s_1) = 0, \beta(s_2) = -2$.

Finally, when $s_2 \geq 0$ *is asserted, the bounds for* s_2 *become* $0 \leq s_2 \leq +\infty$. *The current value* $\beta(s_2) = -2$ *violates this bound, and* $\beta(s_2)$ *has to be increased to* 0. *Since* s_2 *is a basic variable, pivoting has to be performed. Consider the equation* $s_2 = -2x + s_1$. *The value of* s_2 *can be increased only if* x *is decreased, or* s_1 *is increased. Since the value of* x_1 *is already set to its lower bound, and the value of* s_1 *is already set to its upper bound, the inconsistency is detected.*

The explanation for the detected inconsistency is the formula $x \geq 1 \wedge x + y \leq 0 \wedge y - x \geq 0$. *It is itself inconsistent, and* minimal *in the sense that its every subset is consistent. It is inferred from the bounds of the violated equation.*

4 New Approach for Automated Detection of Buffer Overflows

In this section we describe our new, static, flow-sensitive and inter-procedural system for detecting buffer overflows, with modular architecture. We also describe our prototype implementation, called *Fado* (from *F*lexible *A*utomated

*D*etection of Buffer *O*verflows).[4] The system is built from the following building blocks that can be easily changed or updated.

Parser, Intermediate Code Generator, and Code Transformer The parser[5] reads code from the source files, parses it, and builds a parse tree. The parse tree is then exported to a specific intermediate code simpler for processing. 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.), yielding a program 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 [26].

Modelling Semantics of Programs, Database and Conditions Generator For modelling the data-flow and semantics of programs, in formulation of the constraints, we use the following functions:

- *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 bytes 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. So, $value(k, 0)$ gives a value of k in state 0, $used(s, 1)$ gives a number of bytes used by s in state 1, etc. When processing a sequence of commands, states for *value*, *size*, and *used*, are updated, with respect to previous commands and states, in order to take into account the wider context. The values $size(s, i)$ and $used(s, i)$ are always non-negative.

The database is used for generating preconditions and postconditions for single commands. 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. The database is external and can be changed by the user. Initially, the database stores information about standard C operators and functions from 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 (but the system can also work if the user fails to do that). So, while processing a C program, the

[4] FADO is being developed by the first author of this paper.

[5] The Fado tool uses the parser `JSCPP`, written by Jörg Schön, available from `http://www.die-schoens.de/prg/index.html`.

database may temporarily expand with entries corresponding to functions from the program being processed.

Like some other tools, our system tests only the first iteration of a loop (which is reasonable and sufficient in some cases), covers function calls with constant arguments, and applies several other simple heuristics for dealing with commands within loops.

Generator and Optimizer for Correctness and Incorrectness Conditions

For a command K, let Φ be conjunction of postconditions for all commands that precede K (within its function). The command K is:

- *safe* (it never causes an error during execution) if $\Phi \Rightarrow precond(K)$ (universal closure is assumed) is valid;
- *flawed* (when encountered, it always causes an error during execution) if $\Phi \Rightarrow \neg precond(K)$ (universal closure is assumed) is valid;
- *unsafe*, if neither of above (when encountered, it can cause an error during execution).

Notice that our system can prove that some commands are unsafe, but can also prove that some commands are safe. This feature limits the number of false alarms — one of the main concerns for most approaches. Additionally, in some cases, a command can be proved to be both safe and flawed (when the preconditions that precede the command are inconsistent), meaning that the command is not reachable. So, our system can be used for detecting non-reachable code, too.

Before sending conditions to the prover, conjectures are preprocessed. All references to preconditions and postconditions of functions are resolved, all irrelevant conjuncts are eliminated, ground expressions are evaluated, certain expressions are simplified, and terms that do not belong to linear arithmetic are abstracted, i.e., replaced by new variables. This transformation is not complete, but it is sound: if abstracted formula is valid, then the original formula is valid too.

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 modelled by linear arithmetic).

Example 5 *For illustration of the described approach, let us consider the following fragment of code:*

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

Let the database have the following entries:

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

For instance, $used(x,1) \leq value(y,0)$ *says that space used by* x after *execution of the command* `fgets(x,y,z)` *is less or equal to the value of* y before *execution of this command.*

After the initial analysis of the code, it is transformed to an intermediate code (the same code in this example) and then preconditions and postconditions are generated based on the database:

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

After updating states in functions size and used, and after evaluation *(in this case,* $value(200,0)$ *is rewritten to* 200*), we get:*

precondition	*command*	*postcondition*
–	`char src[200];`	$size(src,1) = 200$
$size(src,1) \geq 200$	`fgets(src,200,stdin);`	$used(src,1) \leq 200$

The correctness and incorrectness conditions are abstracted *(so they fall in linear arithmetic). For instance, the command* `fgets(src,200,stdin)` *is safe if* $(0 \leq size\_src\_1) \wedge (size\_src\_1 = 200) \Rightarrow (size\_src\_1 \geq 200)$ *is valid. This can be proved by a theorem prover covering linear arithmetic.*

Invoking Automated Theorem Prover 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 made faster 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. Rather than testing the validity of a quantifier-free formula F (implicitly universally quantified) obtained by conditions generator, SMT provers equivalently test the satisfiability of the formula $\neg F$ (implicitly existentially quantified).[6] The prover can check whether or not the given formula is unsatisfiable (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, in some cases, generate a counterexample for the corresponding correctness conjecture. This counterexample can be used for building a concrete illustration of a buffer overflow, which could be very helpful to the user.

[6]The formula $\forall * F$ is valid if and only if $\exists * \neg F$ is unsatisfiable.

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 (these errors are often trivial, and usually trivial to detect by simple program testing). The commands that are marked *unsafe* are possible causes of errors and they also must be checked by human programmers.

It is impossible to build a complete and sound static system (a system that detects all possible buffer overflows and has no false alarms) for detecting buffer overflow errors. 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; the generated conjectures belong to linear arithmetic, so the other involved theories are not considered. The system uses the ArgoLib 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 overflows.

The power of our system is also determined by the contents of the database. We deliberately leave the database to be external and open — so its contents can be extended by the user.

5 Related work

Several state-of-the-art SMT solvers support linear arithmetic. Although several decision procedures for linear arithmetic have been developed (based on both Simplex and Fourier-Motzkin elimination), the variant of the Simplex method used in YICES and described in this paper is adopted by more solvers (e.g., MathSat, Barselogic, Z3, CVC).

Concerning the static techniques for detecting buffer overflows, over the last several years, there have been several tools developed. There cannot be a complete and sound static system (a system that detect all possible buffer overflows and nothing more). Systems that perform static analysis of code try to maximize the number of detected bugs and to minimize the number of false alarms. These systems can be divided into two classes, first that performs only lexical analysis of code and second that takes into account semantics of the code being analyzed. Systems based on lexical analysis of code, like ITS4 [20], RATS [19] and Flawfinder [23], scan the source code and try to match its fragments with critical calls stored in a special-purpose library. Systems that perform deeper analysis of code, like ARCHER [25], BOON [22], UNO [10], CSSV [6] and Splint [13], usually generate different sorts of constraints over integer variables. These constraints correspond to the safety critical commands and represent correctness conditions that have to be satisfied for the commands to be safe. To generate and check constraints different approaches and algorithms are used. For example, ARCHER [25] uses a custom built integer constraint solver (that is not

sound nor complete), BOON [22] uses a complete custom built range solver, etc. For an empirical comparison between different static analysis tools see, for instance, [27, 24, 12].

6 Conclusions and Future Work

We presented an application of the Simplex method for automated detection of buffer overflows in programs written in C. Our system for automated detection of buffer overflows performs flow-sensitive and inter-procedural static analysis. The system generates correctness and incorrectness conditions for individual commands, and then tests them for validity by a variant of the Simplex method. Some of the novelties introduced by our system are: its very flexible architecture (so its building blocks can be easily changed), buffer overflow correctness conditions given in terms of Hoare logic (with a clear logical meaning), using external theorem provers (that can also provide formal correctness proofs), etc.

The presented system is a subject of further improvements and development. For instance, despite the fact that heuristics for dealing with loops are very 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 [6]). We are also planning to improve analysis of user-defined functions so the system would be sound and fully automatic.

In the theorem proving part of our system, we are planning to modify it to use stronger background theories. The current version of our system checks the satisfiability of linear arithmetic constraints over rationals. The Simplex method could be modified to determine the satisfiability of linear arithmetic constraints over integers i.e., to check if there is an integer valuation of the variables satisfying the given constraints. Although this is more natural approach for checking buffer overflows, it could significantly slow down the whole system. The current version of the system simply abstracts all function calls with variables. So, for the following snippet of code `a = b; x = f(a); y = f(b);`[7] it holds that $x = y$, but the system cannot deduce that. This could be improved by using *Ackermans reduction* [1] which statically adds constraints $a = b \implies f(a) = f(b)$, for all function calls, or by replacing the theory LRA with the combination of theories EUF (Equality with Uninterpreted Functions) and LRA.

References

[1] W. Ackerman. Solvable cases of the decision problem. *Studies in Logic and the Foundations of Mathematics*, 1954.

[2] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):104–107, May 1977.

[7] It is assumed that `f` has no side-effects.

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

[4] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[6] N. Dor, M. Rodeh, and M. Sagiv. 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*, pages 155–167. ACM Press, New York, NY, USA, 2003.

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

[8] B. Dutertre and L De Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006.

[9] A. Forsgren, P. E. Gill, and M. H. Wright. Interior methods for nonlinear optimization. *SIAM Rev*, 44:525–597, 2002.

[10] G. Holzmann. Static source code checking for user-defined properties. In *Proceedings of 6th World Conference on Integrated Design and Process Technology*, Pasadena, CA, June 2002.

[11] V. Klee, G. J. Minty, and O. Shisha. How good is the simplex algorithm? *(Eds.) In Inequalities 3*, pages 159 – 175, 1972.

[12] K. Kratkiewicz and R. Lippmann. 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*, Chicago, June 2005.

[13] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington D.C., August 2001.

[14] J.-L. Lassez and M.J. Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning*, 9:373–379, 1992.

[15] C. E. Lemke. The dual method of solving the linear programming problem. *Naval Research Logistics Quarterly*, pages 36–47, 1954.

[16] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[17] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.

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

[19] Secure software solutions. Rats, the rough auditing tool for security. September 2001. on-line at: http://www.securesw.com/rats/.

[20] J. Viega, J.T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000.

[21] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[22] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed System Security*, pages 3–17, San Diego, CA, February 2000.

[23] D. A. Wheeler. Flawfinder. May 2001. on-line at: http://www.dwheeler.com/flawfinder/.

[24] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (Nordsec 2002)*, pages 68–84, Karlstad, Sweden, November 2002.

[25] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327 – 336. ACM Press, 2003.

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

[27] M. Zitser, R. Lippmann, and T. Leek. 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 table of contents*, pages 97–106, Newport Beach, CA, USA, 2004. ACM.