Analyzing String Buffers in C

Axel Simon and Andy King

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK

Abstract. A buffer overrun occurs in a C program when input is read into a buffer whose length exceeds that of the buffer. Overruns often lead to crashes and are a widespread form of security vulnerability. This paper describes an analysis for detecting overruns before deployment which is conservative in the sense that it locates every possible buffer overrun. The paper details the subtle relationship between overrun analysis and pointer analysis and explains how buffers can be modeled with a linear number of variables. As far as we know, the paper gives the first *formal* account of how this software and security problem can be tackled with abstract interpretation, setting it on a firm, mathematical basis.

1 Introduction

Although C programs are ubiquitous, occurring in many security-critical, safety-critical and enterprise-critical applications, they are particularly prone to inexplicable crashes. Many crashes can be traced to buffer overruns [14]. A buffer overrun occurs when input is read into a buffer and the length of the input exceeds the length of the buffer. Buffer overruns typically arise from unbounded string copies using sprintf(), strcpy() and strcat(), as well as loops that manipulate strings without an explicit length check in the loop invariant [15].

Unchecked input which overflows a buffer can overwrite portions of the stack frame. Hackers have exploited this effect to redirect the instruction pointer in the stack frame to malicious code within the string and thereby gain partial or total control of a host [17]. Buffer overruns are a particularly widespread class of security vulnerability [5] and the National Security Agency predicts that overrun attacks will continue to be a problem for at least the next decade [20]. Finding security faults, such as string handling that may overrun, has been likened to the problem of finding a needle in a hay-stack [11]. This paper describes an abstract interpretation scheme that does not attempt to find the needle, but rather aims to prune the hay-stack down so that it can be searched manually.

1.1 Design space for overrun analysis

One (surprising) tradeoff that is applied in program analysis is that of sacrificing soundness for tractability [24]. In the context of detecting overruns, one potential tradeoff in the design-space is that of achieving simplicity by, say ignoring aliasing, at the cost of possibly missing real errors and possibly reporting spurious errors. Whether this is acceptable depends on whether the analysis aims to detect *most* errors [12, 13, 23] (debugging) or detect *every* error [8] (verification). One point in this design-space is represented by LCLint. LCLint is an annotation-assisted static checking tool founded on the philosophy that it is fine to "accept a solution that is both unsound and incomplete" [12]. This enables, for example, loops to be analyzed straightforwardly using heuristics. Its annotation mechanism achieves both compositionality and scalability. However, as [12, 13] point out, it seems optimistic to believe that a programmer will add annotations to their programs to detect overrun vulnerabilities. Another point in the design space is the constraint based analysis of [23] which can detect potential overruns in unannotated (legacy) C code. This analysis ignores pointer arithmetic and is therefore unsound, but it is fast enough to reason about medium-sized applications without assistance. For speed and simplicity, the analysis is flow insensitive which means that it cannot reason about (non-idempotent) library functions such as strcat() which are a frequent source of overruns [15].

The work of Dor, Rodeh and Sagiv [8] is unique in that it attempts to formulate an analysis that is *conservative* in the sense that it never misses a potential overrun. This is a particularly laudable goal in the context of security where there is a history of elite hackers levering inconspicuous and innocuous features into major security holes. Our work builds on the foundation of Dor *et al* [8] and is a systematic examination of the abstractions and analyses necessary to reason about overruns in a *truly* conservatively way.

1.2 Conservative overrun analysis

The buffer abstraction of Dor *et al* [8] is essentially that of Wagner [23] but specified as a program transformation. Each buffer is abstracted by its size, *alloc*, and the position of the null character, *len*. This two-variable model, however, is not rich enough to accurately track the null position. Suppose, for example, if two pointers p and q point to different positions within the same buffer, then altering the *len* attribute of p (by writing a null character to the buffer) might alter the *len* attribute of q. Dor *et al* [8] therefore introduce special $p\_overlaps\_q$ variables to quantify the pointer offset and thereby model string length interaction. This tactic potentially increases the number of variables quadratically which is unfortunate since relational numeric abstractions such as polyhedra [4] become less tractable as the number of variables increase. This is an efficiency issue. More subtle is the correctness issue that relates to pointers that definitely or possibly point to the same buffer. This is illustrated in the following code:

```
char *p, *q, s[32], t[32];
strcpy(s,"Boat ");        /* s[5] */
strcpy(t,"Aero");         /* s[5] t[4] */
p = t+4;                  /* p[4] s[5] t[4] */
strcat(p, "plane ");      /* p[10] s[5] t[10] */
if (rand()) q=s; else q=t;/* p[10] q[5,10] s[5] t[10] */
strcat(q,"to Réunion");   /* p[10,20] q[15,20] s[5,15] t[10,20]*/
```

The comments indicate the possible null positions at the various lines. The `strcat` in line 7 either updates the `s` buffer or the `t` buffer (but not both) depending on the random value. The transformation described by Dor *et al* [8] does not have a mechanism for dealing with possible changes to a buffer – only definite changes are tracked. In particular the transformation does not correctly reflect that the null is either at position 5 or 15 in the `s` buffer. Our remedy to this is to infer that `q` possibly shares with `s` and `t` so that both 5 and 15 are possible null positions for `s` and symmetrically both 10 and 20 are potential null positions for `t`. Thus *possible* sharing information is used as an aid to correctness. Conversely, *definite* sharing information is used to increase precision. It enables a write to a buffer through one pointer to destructively update the null positions for those pointers that definitely share the same buffer. For example, the `strcat` at line 5 changes the null position of the `t` buffer from 4 to 10. The information that `t` and `p` definitely share is needed to infer that the null position in `t` is no longer 4. Moreover, possible sharing is also required to deduce that no other buffers are affected. To summarize, our work makes the following contributions:

- It gives the first formal account of buffer overrun analysis. A buffer abstraction map is proposed that specifies how to model a buffer. This abstraction is then used to formalize the correctness of the analysis. This means that a programmer can be confident that every possible overrun is detected.
- It shows that any string buffer overrun analysis that is both accurate and correct needs to reason about pointer aliasing. Inter-procedural points-to analysis [1, 21] is required to ascertain which pointers *possibly* point to the same buffer. The paper also shows how to improve precision by employing a *definite* sharing analysis [9].
- It explains how buffers can be modeled with three variables per pointer and shows how the need for $p\_overlaps\_q$ is finessed by sharing analysis, thereby avoiding quadratic blowup. The paper thus describes a practical foundation for analyzing string buffers.

Section 2 introduces the language String C to formalize operations on string buffers. Section 3 explains how polyhedra and sharing domains can be used to describe buffer properties, and then Section 4 explains how these properties can be tracked by analysis. Sections 5 and 6 and present the related and future work and Section 7 concludes. Proofs are omitted because of lack of space.

2 The String C language and its semantics

The analysis is formulated in terms of a C-like mini language called String C that expresses the essential operations on buffers.

2.1 Abstract syntax

Let $(n \in)\ \mathbb{N}$ denote the set of non-negative integers and $(l \in)\ Lab$ denote a set of labels which mark program points. Let $(x, y \in)\ X$ and $(f \in)\ F$ denote the

$$
\begin{array}{lll}
[\text{skip}] & \rho \vdash \langle \text{skip}, \sigma \rangle \to \sigma & \\
[\text{num}] & \rho \vdash \langle x = n, \sigma \rangle \to \sigma.[\rho(x) \mapsto n] & \text{if } n \in \mathbb{N} \\
[\text{str}] & \begin{array}{l} \rho \vdash \langle x = "n_1 \dots n_m", \sigma_1 \rangle \to \\ \quad \sigma_2.[\rho(x) \mapsto \langle a, a, a+m+1 \rangle] \end{array} & \begin{array}{l} \text{if } n_1, \dots, n_m \in \mathbb{N} \\ \wedge\ [a, a+m+1] \cap \text{dom}(\sigma_1) = \emptyset \\ \wedge\ \sigma_2 = \sigma_1.[a+i \mapsto n_{i+1}]_{i=0}^{m-1}. \\ \quad [a+m \mapsto 0] \end{array} \\
[\text{var}] & \rho \vdash \langle x = y, \sigma \rangle \to \sigma.[\rho(x) \mapsto \sigma.\rho(y)] & \\
[\text{add}] & \rho \vdash \langle x = y_1 + y_2, \sigma \rangle \to \sigma.[\rho(x) \mapsto v] & \text{if } \sigma.\rho(y_i) = v_i \wedge v_1 \boxplus v_2 = v \\
[\text{sub}] & \rho \vdash \langle x = y_1 - y_2, \sigma \rangle \to \sigma.[\rho(x) \mapsto v] & \text{if } \sigma.\rho(y_i) = v_i \wedge v_1 \boxminus v_2 = v \\
[\text{arr}_1] & \rho \vdash \langle x = y_1[y_2], \sigma_1 \rangle \to \begin{cases} \sigma_2 & \text{if } a_b \le a < a_e \\ \text{err} & \text{otherwise} \end{cases} & \begin{array}{l} \text{if } \sigma_1.\rho(y_i) = v_i \\ \wedge\ v_1 \boxplus v_2 = \langle a_b, a, a_e \rangle \\ \wedge\ \sigma_2 = \sigma_1.[\rho(x) \mapsto \sigma_1(a)] \end{array} \\
[\text{arr}_2] & \rho \vdash \langle y_1[y_2] = x, \sigma_1 \rangle \to \begin{cases} \sigma_2 & \text{if } a_b \le a < a_e \\ \text{err} & \text{otherwise} \end{cases} & \begin{array}{l} \text{if } \sigma_1.\rho(y_i) = v_i \wedge \sigma_1.\rho(x) \in \mathbb{N} \\ \wedge\ v_1 \boxplus v_2 = \langle a_b, a, a_e \rangle \\ \wedge\ \sigma_2 = \sigma_1.[a \mapsto \sigma_1.\rho(x)] \end{array} \\
[\text{malloc}] & \rho \vdash \langle x = \text{malloc}(y), \sigma_1 \rangle \to \sigma_1.\sigma_2.\sigma_3 & \begin{array}{l} \text{if } \sigma.\rho(y) = v \\ \wedge\ [b, b+v] \cap \text{dom}(\sigma) = \emptyset \\ \wedge\ \sigma_2 = [b+i \mapsto \text{rand}()]_{i=0}^{v} \\ \wedge\ \sigma_3 = [\sigma.\rho(x) \mapsto \langle b, b, b+v \rangle] \end{array}
\end{array}
$$

Fig. 1. Concrete semantics for simple statements. The function rand() returns random values and models malloc's behavior to allocate uninitialized memory

(finite) sets of variables and function names occurring in a program. F includes the symbol `main` to indicate the program entry point and each function $f \in F$ has an associated variable $f_r \in X$ to return values from functions in Pascal-style.

$$
\begin{array}{l}
\mathcal{C} ::= f(x_1, \dots, x_n)\mathcal{L}; \mathcal{C} \ \mid \varepsilon \\
\mathcal{L} ::= [\text{skip}]^l \mid [x = n]^l \mid [x = "n_1 \dots n_m"]^l \mid [x = y]^l \mid [x = y_1 + y_2]^l \\
\quad \mid [x = y_1 - y_2]^l \mid [x = y_1[y_2]]^l \mid [y_1[y_2] = x]^l \mid [x = \text{malloc}(y)]^l \\
\quad \mid [\text{if } x \text{ then } \mathcal{L}_1 \text{ else } \mathcal{L}_2]^l \mid [\text{while } x\ \mathcal{L}]^l \mid [\mathcal{L}_1; \mathcal{L}_2]^l \mid [\text{return}]^l \mid [x = f(y_1, \dots y_n)]^l
\end{array}
$$

The language $\mathfrak{L}(\mathcal{C})$ defines the set of valid String C programs. Given a fixed program, the map $c : F \to (\cup_{i \in \mathbb{N}} X^i) \times \mathfrak{L}(\mathcal{L})$ retrieves the formal arguments $\langle x_1, \dots, x_n \rangle$ and body L for a function f. String C does not distinguish between integers and unsigned chars because reasoning about buffers of different sized objects increases the number of cases in the abstract semantics and obscures the underlying ideas. String C can be enriched following the ideas detailed in [1, 18].

2.2 Instrumented operational semantics

The semantics of String C instruments each pointer with details of its underlying buffer. Specifically, the set of pointer values is defined as $Pnt = \{\langle a_b, a, a_e \rangle \in \mathbb{N}^3 \mid a_b < a_e\}$. The interpretation of a triple $\langle a_b, a, a_e \rangle$ is that the a_b and a_e record the first location within the buffer and the first location past the end of the buffer. The address a records the current position of the pointer within the

$$
\begin{array}{rll}
[\mathrm{if}_1] & \rho \vdash \langle \text{if } x \text{ then } L_1 \text{ else } L_2, \sigma\rangle \to \langle L_1, \sigma\rangle & \text{if } \sigma.\rho(x) \neq 0 \\
[\mathrm{if}_2] & \rho \vdash \langle \text{if } x \text{ then } L_1 \text{ else } L_2, \sigma\rangle \to \langle L_2, \sigma\rangle & \text{if } \sigma.\rho(x) = 0 \\
[\mathrm{while}_1] & \rho \vdash \langle \text{while } x\, L, \sigma\rangle \to \langle L;\ \text{while } x\, L, \sigma\rangle & \text{if } \sigma.\rho(x) \neq 0 \\
[\mathrm{while}_2] & \rho \vdash \langle \text{while } x\, L, \sigma\rangle \to \sigma & \text{if } \sigma.\rho(x) = 0 \\
[\mathrm{seq}_1] & \dfrac{\rho \vdash \langle L_1, \sigma_1\rangle \to \langle L_2, \sigma_2\rangle}{\rho \vdash \langle L_1;\ L_3, \sigma_1\rangle \to \langle L_2;\ L_3, \sigma_2\rangle} & \\
[\mathrm{seq}_2] & \dfrac{\rho \vdash \langle L_1, \sigma_1\rangle \to \sigma_2}{\rho \vdash \langle L_1;\ L_2, \sigma_1\rangle \to \langle L_2, \sigma_2\rangle} & \\
[\mathrm{env}_1] & \dfrac{\rho_2 \vdash \langle L_1, \sigma_1\rangle \to \langle L_2, \sigma_2\rangle}{\rho_1 \vdash \langle \text{env } \rho_2 \text{ in } L_1, \sigma_1\rangle \to \langle \text{env } \rho_2 \text{ in } L_2, \sigma_1\rangle} & \\
[\mathrm{env}_2] & \dfrac{\rho_2 \vdash \langle L, \sigma_1\rangle \to \sigma_2}{\rho_1 \vdash \langle \text{env } \rho_2 \text{ in } L, \sigma_1\rangle \to \sigma_2} & \\
[\mathrm{ret}] & \rho \vdash \langle \text{return};\ L, \sigma\rangle \to \sigma & \\
[\mathrm{call}] & \begin{array}{l} \rho_1 \vdash \langle x = f(y_1, \ldots, y_n), \sigma_1\rangle \to \\ \quad \langle \text{env } \rho_1.\rho_2.[f_r \mapsto \rho(x)] \text{ in } L, \sigma_2.\sigma_1\rangle \end{array} & \begin{array}{l} \text{if } c(f) = \langle z_1, \ldots, z_n, L\rangle \\ \wedge\ \rho_2 = [z_i \mapsto a_i]_{i=1}^n \\ \wedge\ \mathrm{dom}(\sigma_1) \cap \mathrm{rng}(\rho_2) = \emptyset \\ \wedge\ \sigma_2 = [a_i \mapsto \sigma_1.\rho_1(y_i)]_{i=1}^n \end{array}
\end{array}
$$

Fig. 2. Concrete semantics for control statements

buffer. For a valid buffer access $a_b \leq a < a_e$ must hold. The set of values is then defined as $Val = \mathbb{N} \cup Pnt$ whereas the set of environment and store maps are defined $Env = X \to \mathbb{N}$ and $Str = \mathbb{N} \to Val$ respectively. The following (partial) maps formalize address arithmetic.

Definition 1. The functions $\boxplus, \boxminus : Val^2 \to Val$ are defined as follows:

$$
\begin{array}{ccccc@{\qquad}ccccc}
i & \boxplus & j & = & i + j & i & \boxminus & j & = & i - j \\
i & \boxplus & \langle b_b, b, b_e\rangle & = & \langle b_b, b + i, b_e\rangle & i & \boxminus & \langle b_b, b, b_e\rangle & = & \perp \\
\langle a_b, a, a_e\rangle & \boxplus & j & = & \langle a_b, a + j, a_e\rangle & \langle a_b, a, a_e\rangle & \boxminus & j & = & \langle a_b, a - j, a_e\rangle \\
\langle a_b, a, a_e\rangle & \boxplus & \langle b_b, b, b_e\rangle & = & \perp & \langle a_b, a, a_e\rangle & \boxminus & \langle b_b, b, b_e\rangle & = & a - b
\end{array}
$$

Figures 1 and 2 detail the operational semantics for simple statements and the control statements. In Figure 2, $\mathrm{dom}(f)$ and $\mathrm{rng}(f)$ denote the domain and range of a mapping f and . denotes function composition defined such that $g.f(x) = g(f(x))$. The env statement introduced in Figure 2 models scoping.

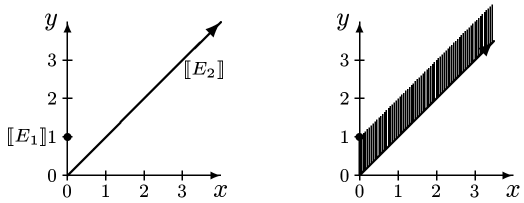
3 Abstract domains

3.1 Linear inequality (polyhedral) domain

Let Y denote the variables $\{y_1, \ldots, y_n\}$, let Lin_Y denote the set of (possibly rearranged) linear equalities $\sum_{i=1}^n m_i y_i = m$ and (possibly rearranged) non-strict inequalities $\sum_{i=1}^n m_i y_i \leq m$ and $\sum_{i=1}^n m_i y_i \geq m$ where $m, m_i \in \mathbb{Z}$. Let Eqn_Y denote all finite subsets of Lin_Y. Note that although each set $E \in Eqn_Y$ is finite, Eqn_Y is not finite. Define $[\![e]\!] = \{\langle x_1, \ldots, x_n\rangle \in \mathbb{R}^n \mid \sum_{i=1}^n m_i x_i \circledcirc m\}$

where $\circledcirc \in \{\leq, =, \geq\}$. Then define $[\![E]\!]$ to be the polyhedron $[\![E]\!] = \cap\{[\![e]\!] \mid e \in E\}$ if $E \in Eqn_Y$. Eqn_Y is ordered by entailment, that is, $E_1 \models E_2$ iff $[\![E_1]\!] \subseteq [\![E_2]\!]$. Equivalence on Eqn_Y is defined $E_1 \equiv E_2$ iff $E_1 \models E_2$ and $E_2 \models E_1$. Let $Poly_Y = Eqn_Y/\equiv$. $Poly_Y$ inherits entailment $\models$ from Eqn_Y. In fact $\langle Poly_Y, \models, \sqcap, \sqcup\rangle$ is a lattice (rather that a complete lattice) with $[E_1]_\equiv \sqcap [E_2]_\equiv = [E_1 \cup E_2]_\equiv$ and $[E_1]_\equiv \sqcup [E_2]_\equiv = [E]_\equiv$ where $[\![E]\!] = cl(conv([\![E_1]\!] \cup [\![E_2]\!]))$ and $cl(S)$ and $conv(S)$ denote the closure and convex hull of a set $S \in \mathbb{R}^n$ respectively [19]. Note that in general $conv([\![E_1]\!] \cup [\![E_2]\!])$ is not closed and therefore cannot be described by a system of non-strict linear inequalities as is illustrated below.

Example 1. Let $Y = \{x, y\}$, $E_1 = \{x = 0, y = 1\}$ and $E_2 = \{0 \leq x, x - y = 0\}$ so that $[\![E_1]\!] = \{\langle 0, 1\rangle\}$ and $[\![E_2]\!] = \{\langle x, y\rangle \mid 0 \leq x \wedge x = y\}$. These polyhedra are illustrated in the left-hand graph. Then $conv([\![E_1]\!] \cup [\![E_2]\!])$ includes the point $\langle 0, 1\rangle$ but not the ray $\{\langle x, y\rangle \mid 0 \leq x \wedge x + 1 = y\}$ and hence is not closed. This convex space is depicted in the right-hand graph.



It is useful to augment $\sqcap$ and $\sqcup$ with three operations: projection, minimization and maximization. The minimization and maximization operations are respectively defined $\min(\langle m_1, \ldots, m_n\rangle, [E]_\equiv) = \min\{\sum_{i=1}^n m_i x_i \mid \langle x_1, \ldots, x_n\rangle \in [\![E]\!]\}$ and $\max(\langle m_1, \ldots, m_n\rangle, [E]_\equiv)$ is defined analogously. The vector $\langle m_1, \ldots, m_n\rangle$ is written as $\sum_{i=1}^n m_i y_i$ for brevity. Note that $\min(\sum_{i=1}^n m_i y_i, [E]_\equiv) \in \mathbb{R} \cup \{-\infty\}$ whereas $\max(\sum_{i=1}^n m_i y_i, [E]_\equiv) \in \mathbb{R} \cup \{+\infty\}$. Projection is defined $\exists_{x_i}([E]_\equiv) = [E']_\equiv$ where $[\![E']\!] = \{\langle x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n\rangle \mid x \in \mathbb{R} \wedge \langle x_1, \ldots, x_n\rangle \in [\![E]\!]\}$. For brevity, let $\exists_{y_{i_1}, \ldots, y_{i_m}}([E]_\equiv) = \exists_{y_{i_m}}(\ldots \exists_{y_{i_1}}([E]_\equiv)\ldots)$. The variable set for $[E]_\equiv$ is defined $\mathrm{var}([E]_\equiv) = \cap\{\mathrm{var}(E') \mid E \equiv E'\}$ where $\mathrm{var}(E')$ is the set of variables (syntactically) occurring in E'. Let $false = [\{0 = 1\}]_\equiv$ so that $[\![false]\!] = \emptyset$.

Projection can be computed using the Fourier algorithm [3], min and max using Simplex [3], $\sqcap$ is straightforward to compute whereas $\sqcup$ can either be calculated using the point, ray and line representation [4] or by using constraint relaxation techniques [7]. Let $s = \sum_{i=1}^n m_i y_i$. Then entailment can be tested by: $P \models \{s < m\}$ iff $\max(s, P) < m$ (even though $\{s < m\}$ is strict); $P \models \{s \leq m\}$ iff $\max(s, P) \leq m$; $P \models \{s = m\}$ iff $P \models \{s \leq m\}$ and $P \models \{s \geq m\}$. Moreover, $P \models \{e_1, \ldots, e_k\}$ iff $P \models \{e_1\} \ldots P \models \{e_k\}$. Henceforth, whenever unambiguous, $[E]_\equiv$ will be written as E for brevity. Finally note that $Poly_Y$ does not satisfy the ascending chain condition, that is, chains $P_1 \models P_2 \models \ldots$ exist for which $\sqcup_{i>0} P_i$ does not exist.

Example 2. Let P_1 be an equilateral triangle, P_2 a hexagon, P_3 a dodecagon and so forth such that the vertices of P_i are contained within those of P_{i+1}. Then $P_1 \models P_2 \models P_3 \ldots$ is an ascending chain which converges onto a circle which itself is not a polyhedron. Thus widening is required to enforce termination in polyhedral fix-point calculations [4].

3.2 Polyhedral buffer domain

Let X_n, X_o and X_s denote sets of variables such that $x \in X$ iff $x_n \in X_n$, $x_o \in X_o$ and $x_s \in X_s$ and suppose that X, X_n, X_o and X_s are pair-wise disjoint. Let $PB_X = Poly_{X \cup X_n \cup X_o \cup X_s}$. The lattice $\langle PB_X, \models, \sqcap, \sqcup \rangle$ is used to describe numeric buffer properties salient to overrun analysis. In particular, suppose an environment ρ maps the variable $x \in X$ to the address $\rho(x) = b$ and a store σ maps b to a pointer triple $\sigma(b) = \langle a_b, a, a_e \rangle$. Then the variable $x_s \in X_s$ describes the number of locations between a_b and a_e (the size of the underlying buffer); $x_o \in X_o$ represents the offset of a relative to a_b (the position of x within the buffer); and $x_n \in X_n$ captures the first location between a_b and a_e whose contents is zero (the position of the null when the buffer is interpreted as a string). If $P \in PB_X$ describes the pair $\langle \rho, \sigma \rangle$, then P captures the numeric relationships between x_s, x_o and x_n. This idea is formalized below.

Definition 2. The polyhedral concretization map $\gamma_X^{PB} : PB_X \to \wp(Env \times Str)$ is defined $\gamma_X^{PB}(P) = \{\langle \rho, \sigma \rangle \mid \alpha_X^{\text{sc}}(\langle \rho, \sigma \rangle) \sqcap \alpha_X^{\text{bf}}(\langle \rho, \sigma \rangle) \models P\}$ where

$$\alpha_X^{\text{sc}}(\langle \rho, \sigma \rangle) = \{x = \sigma.\rho(x) \mid x \in X \wedge \sigma.\rho(x) \in \mathbb{N}\}$$

$$\alpha_X^{\text{bf}}(\langle \rho, \sigma \rangle) = \left\{ \begin{array}{l} x_o = a - a_b, \\ x_s = a_e - a_b, \\ x_n = b - a_b \end{array} \left| \begin{array}{l} x \in X \wedge \sigma.\rho(x) = \langle a_b, a, a_e \rangle \wedge \\ b = \min(\{a_e\} \cup \{n \in [a_b, a_e - 1] \mid \sigma(n) = 0\}) \end{array} \right. \right\}$$

If zero does not occur within the buffer then $\{n \in \mathbb{N} \mid a_b \leq n < a_e \wedge \sigma(n) = 0\} = \emptyset$ so the $\{a_e\}$ element is used to ensure the map is well-defined. It also records the definite absence of a zero (and thereby enables certain definite errors to be found). An abstraction map $\alpha_X^{PB} : \wp(Env \times Str) \to PB_X$ cannot be synthesized from γ_X^{PB} since PB_X is not complete. In particular, $\sqcup\{\alpha_X^{\text{sc}}(\langle \rho, \sigma \rangle) \sqcap \alpha_X^{\text{bf}}(\langle \rho, \sigma \rangle) | \langle \rho, \sigma \rangle \in M\}$ is not well defined for arbitrary $M \in \wp(Env \times Str)$. To put it another way, M does not necessarily have a best polyhedral abstraction.

Tracking the first zero (rather than, say, every zero) keeps the number of variables in the model low which aids simplicity and computational efficiency. However, there are unusual combinations of string operations where just tracing the first null can lead to a loss of precision and hence false warnings.

Example 3. Consider the following (synthetic) C program that zeros the buffer located by s, then and writes the buffer character by character.

```
char *s = malloc(10), t[4];
memset(s,0,10);
s[0]='O'; s[1]='k';
strcpy(t, s);
```

The call to `memset` will set $s_n = 0$, i.e. the first null position is 0. After the first write the analysis can only deduce $s_n \geq 1$ since the first null (if it exists) must occur to the right of the `'0'`. Likewise after the second write the analysis infers $s_n \geq 2$. The write to the 4 character buffer `t` is safe if $s_n \leq 3$. However $s_n \geq 2$ does not imply $s_n \leq 3$ and therefore the call to `strcpy` generates a spurious warning. Inserting `s[2]='a'; s[3]='y';` in front of the call to `strcpy` will yield a definite error since $s_n \geq 4$ implies that $s_n \leq 3$ cannot be satisfied. On the other hand, writing the same four characters to the same positions in reverse order only leads to a warning: $s_n = 0$ holds valid until `s[0]` is written which then updates the null position to $s_n \geq 1$.

3.3 Possible and definite buffer sharing domains

Let $PS_X = \wp(X^2)$ and $DS_X = \wp(X^2)$. The complete lattices $\langle PS_X, \subseteq, \cap, \cup\rangle$ and $\langle DS_X, \supseteq, \cup, \cap\rangle$ are used to express (pair-wise) possible buffer sharing and definite buffer sharing respectively.

Definition 3. The abstraction maps $\alpha_X^{PS} : \wp(Env \times Str) \to PS_X$ and $\alpha_X^{DS} : \wp(Env \times Str) \to DS_X$ and concretization maps $\gamma_X^{PS} : PS_X \to \wp(Env \times Str)$ and $\gamma_X^{DS} : DS_X \to \wp(Env \times Str)$ are defined as follows:

$$\alpha_X^{PS}(M) = \cup\{\alpha_X^{sh}(\langle\rho,\sigma\rangle) \mid \langle\rho,\sigma\rangle \in M\} \quad \alpha_X^{DS}(M) = \cap\{\alpha_X^{sh}(\langle\rho,\sigma\rangle) \mid \langle\rho,\sigma\rangle \in M\}$$
$$\gamma_X^{PS}(S) = \{\langle\rho,\sigma\rangle \mid \alpha_X^{sh}(\langle\rho,\sigma\rangle) \subseteq S\} \qquad \gamma_X^{DS}(D) = \{\langle\rho,\sigma\rangle \mid D \subseteq \alpha_X^{sh}(\langle\rho,\sigma\rangle)\}$$

where $\alpha_X^{sh}(\langle\rho,\sigma\rangle) = \{\langle x, y\rangle \in X^2 \mid \sigma(\rho(x)) = \langle a_b, a_x, a_e\rangle \wedge \sigma(\rho(y)) = \langle a_b, a_y, a_e\rangle\}$.

The intuition is that each $D \in DS_X$ captures the certain presence of sharing in that if $\langle x, y\rangle \in D$ then $\langle x, y\rangle \in \alpha_X^{sh}(\langle\rho,\sigma\rangle)$ for all $\langle\sigma,\rho\rangle \in \gamma_X^{DS}(D)$. Conversely, each $S \in PS_X$ describes the certain lack of sharing, that is, if $\langle x, y\rangle \notin S$ then $\langle x, y\rangle \notin \alpha_X^{sh}(\langle\rho,\sigma\rangle)$ for all $\langle\sigma,\rho\rangle \in \gamma_X^{PS}(S)$. This is the negative interpretation of S. The positive interpretation of S is that S describes the possible present of sharing, that is, if $\langle x, y\rangle \in S$ then there exists $\langle\sigma,\rho\rangle \in \gamma_X^{PS}(S)$ such that $\langle x, y\rangle \in \alpha_X^{sh}(\langle\rho,\sigma\rangle)$.

Proposition 1. α_X^{PS} and γ_X^{PS} are monotonic; $\alpha_X^{PS}(\gamma_X^{PS}(S)) \subseteq S$ for all $S \in PS_X$; and $M \subseteq \gamma_X^{PS}(\alpha_X^{PS}(M))$ for all $M \in \wp(Env \times Str)$.

Since $\wp(Env \times Str)$ and PS_X are complete lattices, it follows that the quadruple $\langle\wp(Env \times Str), \gamma_X^{PS}, PS_X, \alpha_X^{PS}\rangle$ is a Galois connection between $\wp(Env \times Str)$ and PS_X. Likewise $\langle\wp(Env \times Str), \gamma_X^{DS}, DS_X, \alpha_X^{DS}\rangle$ is also a Galois connection.

The overrun analysis presented in this paper pre-supposes possible and definite buffer sharing information. The inter-procedural flow-insensitive points-to analysis of [21] can be adapted to derive the former whereas intra-procedural analysis is likely to be sufficient for the latter. The sharing abstractions S^l and D^l are introduced to abstract away from a particular sharing analysis.

Definition 4. S^l and D^l are defined so that $\langle\rho,\sigma_2\rangle \in \gamma_X^{PS}(S^l) \cap \gamma_X^{DS}(D^l)$ if $\rho \vdash \langle x = \text{main}(), \sigma_1\rangle \to^* \langle L^l, \sigma_2\rangle$ where $\rho = [x \mapsto 0]$, $\sigma_1 = [0 \mapsto n]$ and $n \in \mathbb{N}$.

3.4 Domain interaction

A polyhedral abstraction can be used to refine a possible sharing abstraction whereas a definite sharing abstraction can improve a polyhedral abstraction.

Definition 5. The operators $\varrho_D : PB_X \to PB_X$ and $\varrho_P : PS_X \to PS_X$ are defined $\varrho_D(P) = P \sqcap \{x_n = y_n, x_s = y_s \mid \langle x, y\rangle \in D\}$ and

$$\varrho_P(S) = S \setminus \left\{ \langle x, y\rangle \in S \,\middle|\, \begin{array}{l} P \models x_n < y_n \vee P \models y_n < x_n \vee \\ P \models x_s < y_s \;\vee P \models y_s < x_s \end{array} \right\}$$

Proposition 2. $\varrho_D(P) \models P$, $\gamma_X^{DS}(D) \cap \gamma_X^{PB}(P) = \gamma_X^{DS}(D) \cap \gamma_X^{PB}(\varrho_D(P))$, $\varrho_P(S) \subseteq S$ and $\gamma_X^{PS}(S) \cap \gamma_X^{PB}(P) = \gamma_X^{PS}(\varrho_P(S)) \cap \gamma_X^{PB}(P)$.

4 Abstract semantics for String C

This section presents an analysis for detecting string buffer overruns; it is not intended to verify that a program conforms to the ANSI C standard – it is simply designed to alert the programmer to potential overruns.

4.1 Assignment

The starting point for the construction is a polyhedral operator for assignment.

Definition 6. Let $\circledcirc \in \{\leq, =, \geq\}$, $s = \sum_{j=1}^{n} m_j y_j$ and $t \notin \text{var}(P \sqcap \{x \circledcirc s\})$. Then destructive update, $\rhd$, and additive update, $\unrhd$, are respectively defined:

$$P \rhd x \circledcirc s = \exists_t(\exists_x(P \sqcap \{t \circledcirc s\}) \sqcap \{x = t\}) \qquad P \unrhd x \circledcirc s = P \sqcup (P \rhd x \circledcirc s)$$

If x does not occur in s, that is $x \notin \text{var}(\{t \circledcirc s\})$, then $\exists_x(P \sqcap \{t \circledcirc s\}) = \exists_x(P) \sqcap \{t \circledcirc s\}$ and hence $P \rhd x \circledcirc s = \exists_x(P) \sqcap \{x \circledcirc s\}$. This version of update requires few operations, is more efficient, and also suggests that $P \rhd x \circledcirc s$ can be used to simulate destructive update. In fact this is precisely the rôle of the $\rhd$ operator. The $\unrhd$ operator, additive update, is used to model a destructive update that may *or* may not have been applied. Safety follows because in the former case $P \rhd x \circledcirc s \models P \unrhd x \circledcirc s$ whereas in the latter case $P \models P \unrhd x \circledcirc s$.

Example 4. Let $P = \{x = z, x \leq y + 1\}$ and consider $P \rhd \{x = x + 1\}$. Observe that $t \notin \{x, y, z\} = \text{var}(P \sqcap \{x = x+1\})$. Since $\exists_x(\{x = z, x \leq y+1, t = x+1\}) = \{t = z + 1, t \leq y + 2\}$ and $\exists_t(\{t = z + 1, t \leq y + 2, x = t\}) = \{x = z + 1, x \leq y + 2\})$ it follows that $P \rhd \{x = x + 1\} = \{x = z + 1, x \leq y + 2\})$. Moreover, $P \unrhd \{x = x + 1\} = P \sqcup \{x = z + 1, x \leq y + 2\} = \{z \leq x \leq z + 1, x \leq y + 2\}$.

It is not unusual for a concrete operation to modify several attributes of a polyhedra together and thus it is useful to introduce a concept of parallel update. In particular, let $i \in \{1, \ldots, k\}$ and consider $e_i = \{x_i \circledcirc_i s_i\}$ where $\circledcirc_i \in \{\leq, =, \geq\}$ and $s_i = m_i + \Sigma_{j=1}^{n_i} m_{i,j} y_{i,j}$. Define $P \rhd \langle e_1, \ldots, e_k\rangle = ((P \rhd e_1) \ldots \rhd e_k)$ and likewise $P \unrhd \langle e_1, \ldots, e_k\rangle = ((P \unrhd e_1) \ldots \unrhd e_k)$. The following proposition explains how $P \rhd \langle e_1, \ldots, e_k\rangle$ and $P \unrhd \langle e_1, \ldots, e_k\rangle$ are independent of the evaluation order.

$$
\begin{array}{rl}
[\text{skip}'] & \langle \text{skip}, P\rangle \to' P \\
[\text{num}'] & \langle x = n, P\rangle \to' \exists_{x_n,x_o,x_s}(P) \rhd \{x = n\} \qquad \text{if } n \in \mathbb{N} \\
[\text{str}'] & \langle x = "n_1 \ldots n_m", P\rangle \to' \exists_x(P) \rhd \{x_o = 0, x_s = m+1, x_n = m\} \text{ if } n_i \in \mathbb{N} \\
[\text{var}'] & \langle x = y, P\rangle \to' \begin{cases} \exists_{x_n,x_o,x_s}(P) \rhd \{x = y\} & \text{if } y \in \text{sc}(P) \\ \exists_x(P) \rhd \{x_n = y_n, x_o = y_o, x_s = y_s\} & \text{else if } y \in \text{bf}(P) \\ \exists_{x,x_n,x_o,x_s}(P) & \text{otherwise} \end{cases} \\
& \langle x = y + z, P\rangle \to' \\
[\text{add}'] & \begin{cases} \exists_{x_n,x_o,x_s}(P) \rhd \{x = y + z\} & \text{if } y, z \in \text{sc}(P) \\ \exists_x(P) \rhd \{x_n = z_n, x_o = y + z_o, x_s = z_s\} & \text{else if } y \in \text{sc}(P) \wedge z \in \text{bf}(P) \\ \exists_x(P) \rhd \{x_n = y_n, x_o = y_o + z, x_s = y_s\} & \text{else if } y \in \text{bf}(P) \wedge z \in \text{sc}(P) \\ \exists_{x,x_n,x_o,x_s}(P) & \text{otherwise} \end{cases} \\
& \langle x = y - z, P\rangle \to' \\
[\text{sub}'] & \begin{cases} \exists_{x_n,x_o,x_s}(P) \rhd \{x = y - z\} & \text{if } y, z \in \text{sc}(P) \\ \exists_x(P) \rhd \{x_n = y_n, x_o = y_o - z, x_s = y_s\} & \text{else if } y \in \text{bf}(P) \wedge z \in \text{sc}(P) \\ \exists_{x_n,x_o,x_s}(P) \rhd \{x = y_o - z_o\} & \text{else if } y, z \in \text{bf}(P) \\ \exists_{x,x_n,x_o,x_s}(P) & \text{otherwise} \end{cases} \\
[\text{arr}_1'] & \langle x = y[z], P\rangle \to' \begin{cases} \text{err} & \text{if } P \models \{y_o + z < 0\} \\ \text{err} & \text{else if } P \models \{y_o + z \geq y_s\} \\ \text{warn} & \text{else if } P \not\models \{0 \leq y_o + z < y_s\} \\ \exists_{x_n,x_o,x_s}(P) \rhd \{x = 0\} & \text{else if } P \models \{y_o + z = y_n\} \\ \exists_{x_n,x_o,x_s}(P) \rhd \{x \geq 1\} & \text{else if } P \models \{y_o + z < y_n\} \\ \exists_{x,x_n,x_o,x_s}(P) & \text{otherwise} \end{cases} \\
[\text{arr}_2'] & \langle [y[z] = x]^l, P\rangle \to' \begin{cases} \text{err} & \text{if } P \models \{y_o + z < 0\} \\ \text{err} & \text{else if } P \models \{y_o + z \geq y_s\} \\ \text{warn} & \text{else if } P \not\models \{0 \leq y_o + z < y_s\} \\ P'' \trianglerighteq \{v_i = y_n \mid v_i \in V\} & \text{otherwise} \end{cases} \\
[\text{malloc}'] & \langle x = \text{malloc}(y), P\rangle \to' \exists_x(P) \rhd \{x_n \geq 0, x_o = 0, x_s = y\}
\end{array}
$$

Fig. 3. Abstract semantics for simple statements where $W = \{w_n \mid \langle w, y\rangle \in D^l\}$, $P' = \varrho_{D^l}(P)$, $V = \{v_n \mid \langle v, y\rangle \in \varrho_{P'}(S^l)\} \backslash W$ and $P'' = \varrho_{D^l}(\text{update}_{x,y,z}(\exists_{W \backslash \{y_n\}}(P')))$.

Proposition 3. *Let $\pi : \{1, \ldots, k\} \to \{1, \ldots, k\}$ be a permutation, $e_i = \{x_i \circledcirc_i s_i\}$, $s_i = m_i + \Sigma_{j=1}^{n_i} m_{i,j} y_{i,j}$ and $x_i \notin \text{var}(e_j)$ for all $i \neq j$. Then $P \rhd \langle e_1, \ldots, e_k\rangle = P \rhd \langle e_{\pi(1)}, \ldots, e_{\pi(k)}\rangle$ and $P \trianglerighteq \langle e_1, \ldots, e_k\rangle = P \trianglerighteq \langle e_{\pi(1)}, \ldots, e_{\pi(k)}\rangle$.*

For brevity, define $P \rhd \{e_1, \ldots, e_k\} = P \rhd \langle e_1, \ldots, e_k\rangle$ if $x_i \notin \text{var}(e_j)$ for all $i \neq j$. Proposition 3 ensures that $P \rhd \{e_1, \ldots, e_k\}$ is well-defined whereas Proposition 4 explains how $\trianglerighteq$ can be used to approximate $\rhd$ when a polyhedra can be updated with different combinations of equations (and possibly not at all).

Proposition 4. *$P \rhd E' \models P \trianglerighteq E$ for all $E' \subseteq E$.*

4.2 Non-array statements

To construct abstract versions of the non-array statements, the sc and bf maps are introduced to detect whether an object is a scalar or a pointer. The following proposition states an invariant of any correct analysis: a polyhedra cannot simultaneously record scalar and buffer attributes for a given variable. The lemma then asserts correctness for the abstract semantics of the non-array statements.

Definition 7. The maps $\mathrm{sc} : PB_X \to \wp(X)$ and $\mathrm{bf} : PB_X \to \wp(X)$ are defined $\mathrm{sc}(P) = X \cap \mathrm{var}(P)$ and $\mathrm{bf}(P) = \{x \in X \mid \{x_n, x_o, x_s\} \cap \mathrm{var}(P) \neq \emptyset\}$.

Proposition 5. If $\mathrm{sc}(P) \cap \mathrm{bf}(P) \neq \emptyset$ then $\gamma_X^{PB}(P) = \emptyset$.

Lemma 1. Suppose $L = [\mathrm{skip}]^l \mid \ldots \mid [x = y - z]^l$. Then if $\rho \vdash \langle L, \sigma_1\rangle \to \sigma_2$, $\langle \rho, \sigma_1\rangle \in \gamma_X^{PB}(P)$ and $\langle L, P_1\rangle {\to}' P_2$ it follows that $\langle \rho, \sigma_2\rangle \in \gamma_X^{PB}(P_2)$.

4.3 Array statements

To reason about array statements that can error, the store is extended to $EStr = Str \cup \{\mathrm{err}\}$ where err denotes an error state. Likewise, the polyhedral domain is extended to $EPB_X = PB_X \cup \{\bot, \mathrm{err}, \mathrm{warn}\}$. $\langle EPB_X, \preceq, \curlywedge, \curlyvee\rangle$ is a lattice where the ordering $\preceq$ is defined by $\bot \preceq P$ and $P \preceq \mathrm{warn}$ for all $P \in EPB_X$ whereas for all $P_i \in PB_X$, $P_1 \preceq P_2$ iff $P_1 \models P_2$. Moreover, $\curlywedge$ and $\curlyvee$ are given by:

$$P_1 \curlywedge P_2 = \begin{cases} P_1 \sqcap P_2 \text{ if } P_1, P_2 \in PB_X \\ \mathrm{err} \text{ else if } P_1 = P_2 = \mathrm{err} \\ P_1 \text{ else if } P_2 = \mathrm{warn} \\ P_2 \text{ else if } P_1 = \mathrm{warn} \\ \bot \text{ otherwise} \end{cases} \qquad P_1 \curlyvee P_2 = \begin{cases} P_1 \sqcup P_2 \text{ if } P_1, P_2 \in PB_X \\ \mathrm{err} \text{ else if } P_1 = P_2 = \mathrm{err} \\ P_1 \text{ else if } P_2 = \bot \\ P_2 \text{ else if } P_1 = \bot \\ \mathrm{warn} \text{ otherwise} \end{cases}$$

The following (extended) concretization map explains how the abstract objects err and warn represent definite errors and possible errors in the concrete setting.

Definition 8. The concretization map $\gamma_X^{EPB} : EPB_X \to \wp(Env \times EStr)$ is defined: $\gamma_X^{EPB}(\bot) = \emptyset$, $\gamma_X^{EPB}(\mathrm{err}) = Env \times \{\mathrm{err}\}$, $\gamma_X^{EPB}(\mathrm{warn}) = Env \times EStr$ and $\gamma_X^{EPB}(P) = \gamma_X^{PB}(P)$ if $P \in PB_X$.

The abstract semantics for array read generates an error (warning) if the read is definitely (possibly) outside the buffer. Array write is more subtle since a write action through one pointer can effect the null position attribute of all the pointers directed at the same buffer. The following operator details the way in which the null attribute is effected by the write.

Definition 9. The operator $\mathrm{update}_{x,y,z} : PB_X \to PB_X$ is defined:

$$\mathrm{update}_{x,y,z}(P) = \begin{cases} P_A \text{ if } P \models \{y_o + z > y_n\} \\ P_A \text{ else if } P \models \{x > 0, y_o + z < y_n\} \\ P_B \text{ else if } P \models \{x = 0, y_o + z < y_n\} \\ P_C \text{ else if } P \models \{y_o + z < y_n\} \\ P_D \text{ else if } P \models \{x > 0\} \wedge w_u = n_l = w_l \\ P_B \text{ else if } P \models \{x = 0\} \wedge w_u = n_l = w_l \\ P_E \text{ else if } P \models \{x > 0\} \\ P_A \text{ else if } P \models \{x = 0\} \wedge n_l \leq w_l \leq n_u < w_u \\ P_F \text{ else if } P \models \{x = 0\} \\ P_G \text{ otherwise} \end{cases} \qquad \begin{aligned} P_A &= P \\ P_B &= P \rhd (y_n = y_o + z) \\ P_C &= P \trianglerighteq (y_n = y_o + z) \\ P_D &= P \rhd (y_n \geq y_o + z + 1) \\ P_E &= P \trianglerighteq (y_n \geq y_n) \\ P_F &= P_C \sqcap \{y_n \leq \min\{n_u, w_u\}\} \\ P_G &= P \trianglerighteq (y_n \geq y_o + z) \\ w_l &= \lceil \min(y_o + z, P) \rceil \\ w_u &= \lfloor \max(y_o + z, P) \rfloor \\ n_l &= \lceil \min(y_n, P) \rceil \\ n_u &= \lfloor \max(y_n, P) \rfloor \end{aligned}$$

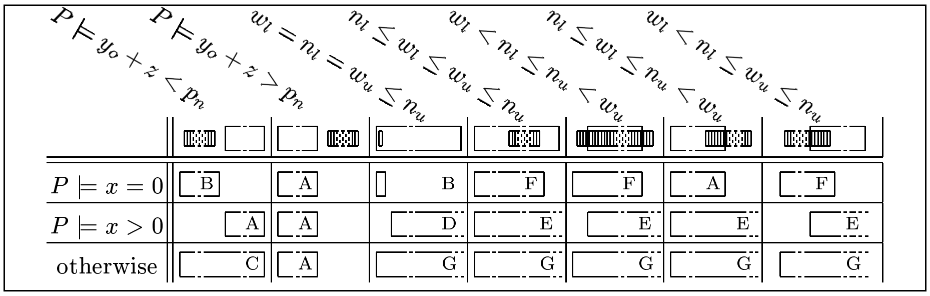


Fig. 4. Graphical representation of the update function.

The operator reduces to a case analysis of the write position relative to the null position. The various cases are depicted in Figure 4 in which the striped bars (hollow bars) represent a range that includes the write (first null) position. In case A the null position is not altered. In B, the null position is refined to coincide with the write position. In C, the write possibly resets the null, hence the additive update. In D, the first element in the range of possible null positions is overwritten with non-zero. This may overwrite the actual null, hence the range is extended without bound at the end and shortened by one at the beginning. In E, the additive update $P \trianglerighteq (y_n \geq y_n)$ extends the null position to the right to capture a possibly overwritten null. In F, case B is refined so that the null exceeds neither upper bound. Finally G provides a conservative approximation when the update value is not known.

The abstract semantics for array write first inspects D^l – the definite sharing abstraction for program point l – to collect together those program variables W that definitely share with y. The polyhedra P is refined using D^l to obtain P'. V is constructed from $\varrho_{P'}(S^l)$ – the possible sharing abstraction for program point l refined by P' – to obtain those variables which possibly share with y, but do not definitely share with y. Information on null positions of those variables of W (except y) is then removed from P'. The update is applied to revise the null attribute for y, and then this change is reflected to other variables of W. Additive update is used to propagate the update to the variables in V since they may or may not have been affected by the write. This step-wise construction enables correctness to be established, and given correct sharing abstractions D^l and S^l for point l (see Section 3.3), the following correctness result is obtained.

Theorem 1. Suppose $L = [\text{skip}]^l \mid \ldots \mid [x = \text{malloc}(y)]^l$. Then if $\rho \vdash \langle L, \sigma_1 \rangle \to \sigma_2$, $\langle \rho, \sigma_1 \rangle \in \gamma_X^{PB}(P)$ and $\langle L, P_1 \rangle \to' P_2$ it follows that $\langle \rho, \sigma_2 \rangle \in \gamma_X^{EPB}(P_2)$.

Note that if $P_2 = \text{err}$, then a definite overrun is detected (if l is ever reached); if $P_2 = \text{warn}$, then a possible overrun may occur at l; whereas if $P \in PB_X$ then no overrun can occur at l.

Example 5. The consecutive updates of Example 3 correspond to case D in the table. The reverse writing of the first three characters corresponds to fourth column and second row and thus to case E. The last write is again a D update.

$$
\begin{array}{rl}
[\mathrm{if}'_1] & \langle \text{if } x \text{ then } L_1 \text{ else } L_2, P\rangle \rightarrow' \langle L_1, P \sqcap \{1 \le x\}\rangle \\
[\mathrm{if}'_2] & \langle \text{if } x \text{ then } L_1 \text{ else } L_2, P\rangle \rightarrow' \langle L_1, P \sqcap \{x \le -1\}\rangle \\
[\mathrm{if}'_3] & \langle \text{if } x \text{ then } L_1 \text{ else } L_2, P\rangle \rightarrow' \langle L_2, P \sqcap \{x = 0\}\rangle \\
[\mathrm{while}'_1] & \langle \text{while } x\, L, P\rangle \rightarrow' \langle L;\ \text{while } x\, L, P \sqcap \{1 \le x\}\rangle \\
[\mathrm{while}'_2] & \langle \text{while } x\, L, P\rangle \rightarrow' \langle L;\ \text{while } x\, L, P \sqcap \{x \le -1\}\rangle \\
[\mathrm{while}'_3] & \langle \text{while } x\, L, P\rangle \rightarrow' P \sqcap \{x = 0\} \\
[\mathrm{seq}'_1] & \dfrac{\langle L_1, P_1\rangle \rightarrow' \langle L_2, P_2\rangle}{\langle L_1;\ L_3, P_1\rangle \rightarrow' \langle L_2;\ L_3, P_2\rangle} \\
[\mathrm{seq}'_2] & \dfrac{\langle L_1, P_1\rangle \rightarrow' P_2}{\langle L_1;\ L_2, P_1\rangle \rightarrow' \langle L_2, P_2\rangle} \\
[\mathrm{ret}'] & \langle \text{return};\ L,\ P\rangle \rightarrow' P \\
[\mathrm{call}'] & \dfrac{\langle \theta(z_1) = y_1;\ \ldots;\ \theta(z_n) = y_n;\ \theta(L), P_1\rangle \rightarrow'^{*} P_2}{\langle x = f(y_1, \ldots y_n), P_1\rangle \rightarrow' \langle x = \theta(f_r), P_2\rangle} \\
& \text{if } c(f) = \langle z_1, \ldots, z_n, L\rangle \wedge \mathrm{var}(\theta(c(f))) \cap \mathrm{var}(P_1) = \emptyset
\end{array}
$$

Fig. 5. Abstract semantics for control statements where θ denotes a renaming (bijective) substitution.

4.4 Control statements

Figure 5 details the abstract semantics for the control statements. Note how the if and while branches restrict the polyhedron, possibly collapsing it to *false*. The main safety result, Theorem 1, can be lifted to full String C by induction on the number of steps from main in a sequence of abstract reductions. Of course, an analysis will have to address finiteness issues, by applying widening [4], and efficiency issues, by combining function call and exit with projection so as to minimize the number of variables considered in the analysis of each function.

5 Related work

Apart from those static analyses already discussed [8, 12, 13, 23], most of proposals for detecting overruns are either based on lexical analysis [22], testing [11] or stack protection mechanisms [2, 10]. ITS4 [22] is a lexical analysis tool that searches for security problems using a database of potentially dangerous constructs. Lexical analysis is fast and calls to problematic library functions can be flagged. Such a limited approach, however, will fail to find problematic string buffer manipulation that is hand coded.

FIST [11] finds possible overruns by automatically perturbing the program states, for example, appending or truncating strings, or by mangling the stack. The developer or analyst selects buffers to check, and then FIST injects a state perturbation to generate a possible overrun. Manual analysis is required to determine whether the overrun buffer can actually occur.

StackGuard protects from stack smashing by aborting the program if the return address is over-written [10]. The StackGuard compiler, however, does not bar overruns and any overrun has the potential of side-effecting a variable and thereby altering access and privileges. Baratloo, Singh and Tsai [2] also describe a mechanism for checking the return address before the jump. Any run-time approach, however, will always incur an overhead (of 40% in the case of StackGuard [10]). Moreover, as pointed out in [12], these run-time systems effectively just turn a buffer overflow attack into a denial-of-service attack.

Finally, the AST ToolKit [6] has been used to detect errors in string manipulation, though details on this work are sparse [8].

6 Future work

One direction for future work will be to investigate the extent to which polyhedral sub-domains [16] impact on the precision of buffer overrun analysis; a programmer might be willing to trade extra warning messages for an analysis that scales smoothly to large applications. For brevity, the analysis omits how to recover from an error or warning state. Future work will thus investigate how to safely reshape the polyhedron to satisfy the requirements of an operation so as to avoid an unhelpful error cascade. Future work will also address how to extend the analysis to Unicode buffers and other problematic buffer operations. For example, pointer difference $i = s - t$ is well defined iff s and t point to the same buffer. Moreover, the analysis can be enriched to flag an error if s and t do not possibly share and raise a warning if s and t do not definitely share.

7 Conclusions

This paper has formalized the buffer overrun problem by following the methodology of abstract interpretation. First, an instrumented semantics for a C subset was presented that captures those properties relevant to overrun detection. Second, polyhedral and buffer sharing domains were proposed and then connected with the concrete semantics via concretization maps. Third, the instrumented semantics was abstracted to synthesize an analysis for detecting both definite and possible overruns. Fourth, correctness results were reported. The paper provides an excellent practical foundation for overrun analysis since it explains how buffer sharing and buffer overrun interact and how sharing analysis can be used to trim down the number of variables in a polyhedral buffer abstraction.

Acknowledgements We thank Florence Benoy, John Gallagher, Les Hatton and Jacob Howe for interesting discussions.

References

1. L. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, Datalogisk Institut Kobenhavns Universitet, 1994.
2. A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In *Ninth USENIX Security Symposium*, 2000.

3. V. Chandru and M.R. Rao. Linear programming. In *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
4. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *Proceedings of Principles of Programming Languages*, pages 84–97. ACM Press, 1978.
5. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Information Survivability Conference and Exposition*, volume II, pages 154–163. IEEE Press, 1998.
6. R. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Conference on Domain-Specific Languages*, pages 229–242. USENIX Association, 1997.
7. B. De Backer and H. Beringer. A CLP language handling disjunctions of linear constraints. In *International Conference on Logic Programming*, pages 550–563. MIT Press, 1993.
8. N. Dor, M. Rodeh, and M. Sagiv. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In *Static Analysis Symposium*, volume 2126 of *LNCS*, pages 194–212. Springer-Verlag, 2001.
9. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural analysis in the presence of function pointers. In *Programming Language Design and Implementation*, pages 242–256, June 1994.
10. C. Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, 1998.
11. A. Ghosh, T. O'Connor, and G. McGraw. An Automated Approach for Identifying Potential Vulnerabilities in Software. In *IEEE Symposium on Security and Privacy*, pages 104–114. IEEE Computer Society, 1998.
12. D. Larochelle and D. Evans. Statically Detecting likely Buffer Overflow Vulnerabilities. In *Tenth USENIX Security Symposium*. USENIX Association, 2001.
13. D. Larochelle and D. Evans. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.
14. B. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.
15. T. C. Miller and T. de Raadt. strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation. In *USENIX Annual Technical Conference*, 1999.
16. A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects*, volume 2053 of *LNCS*, pages 155–172, 2001.
17. A. One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7(49).
18. N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.
19. R. T. Rockafellar. *Convex Analysis*. Princeton University Press, 1970.
20. B. Snow. Panel Discussion on the Future of Security. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1999.
21. B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Principles of Programming Languages*, pages 32–41. ACM Press, 1996.
22. J. Viega, J. T. Bloch, T. Kohno, and G. McGraw. ITS4: A Static Vulnumerability Scanner for C and C++ Code. In *Sixteenth Annual Computer Security Applications Conference*, 2000.
23. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*. Internet Society, 2000.
24. D. Weise. Static Analysis of Mega-Programs. In *Static Analysis Symposium*, volume 1694 of *LNCS*, pages 300–302. Springer-Verlag, 1999.