Extended Weighted Pushdown Systems

Akash Lal, Thomas Reps, and Gogul Balakrishnan

University of Wisconsin, Madison, Wisconsin 53706
{akash, reps, bgogul}@cs.wisc.edu

Abstract. Recent work on weighted-pushdown systems shows how to generalize interprocedural-dataflow analysis to answer "stack-qualified queries", which answer the question "what dataflow values hold at a program node for a particular set of calling contexts?" The generalization, however, does not account for precise handling of local variables. Extended-weighted-pushdown systems address this issue, and provide answers to stack-qualified queries in the presence of local variables as well.

1 Introduction

An important static-analysis technique is dataflow analysis, which concerns itself with calculating, for each program point, information about the set of states that can occur at that point. For a given abstract domain, the ideal value to compute is the meet-over-all-paths (MOP) value. Kam and Ullman [10] gave a coincidence theorem that provides a sufficient condition for when this value can be calculated for single-procedure programs. Later, Sharir and Pnueli [23] generalized the theorem for multiple-procedure programs, but did not consider local variables. Knoop and Steffen [12] then further extended the theorem to include local variables by modeling the run-time stack of a program. Alternative techniques for handling local variables have been proposed in [17, 19], but these lose certain relationships between local and global variables.

The MOP value over-approximates the set of all possible states that occur at a program point (for all possible calling contexts). Recent work on weighted-pushdown systems (WPDSs) [18] shows how to generalize interprocedural-dataflow analysis to answer "stack-qualified queries" that calculate an over-approximation to the states that can occur at a program point for a given regular set of calling contexts. However, as with Sharir and Pnueli's coincidence theorem, it is not clear if WPDSs can handle local variables accurately. In this paper, we extend the WPDS model to the Extended-WPDS (EWPDS) model, which can accurately encode interprocedural-dataflow analysis on programs with local variables and answer stack-qualified queries on them. The EWPDS model can be seen as generalizing WPDSs in much the same way that Knoop and Steffen generalized Sharir and Pnueli's coincidence theorem.[1]

The contributions of this paper can be summarized as follows:

- We give a way of handling local variables in an extension of the WPDS model. The advantage of using (E)WPDSs is that they give a way of calculating dataflow values that hold at a program node for a particular calling context (or set of calling contexts). They can also provide a set of "witness" program execution paths that justify a reported dataflow value.
- We show that the EWPDS model is powerful enough to capture Knoop and Steffen's coincidence theorem. In particular, this means that we can calculate

[1] Recently, with S. Schwoon, we have shown that the computational power of WPDSs is the same as that of EWPDSs. We do not present this result in this paper due to space constraints, but it involves simulating the program run-time stack as a dataflow value.

the MOP value (referred to as the interprocedural-meet-over-all-valid-paths, or IMOVP value, for multiple-procedure programs with local variables) for any distributive dataflow-analysis problem for which the domain of transfer functions has no infinite descending chains. For monotonic problems that are not distributive, we can safely approximate the IMOVP value. In addition to this, EWPDSs support stack-qualified IMOVP queries.
– We have extended the WPDS++ library [11] to support EWPDSs and used it to calculate affine relationships that hold between registers in x86 code [2].

A further result was too lengthy to be included in this paper, but illustrates the value of our approach: we have shown that the IMOVP result of [13] for single-level pointer analysis is an instance of our framework.[2] This immediately gives us something new: a way of answering stack-qualified aliasing problems.

The rest of the paper is organized as follows: §2 provides background on WPDSs and explains the EWPDS model; §3 presents algorithms to solve reachability queries in EWPDSs. In §4, we show how to compute the IMOVP value using an EWPDS; §5 presents experimental results; and §6 describes related work.

2 The EXTENDED-WPDS Model

2.1 Pushdown Systems

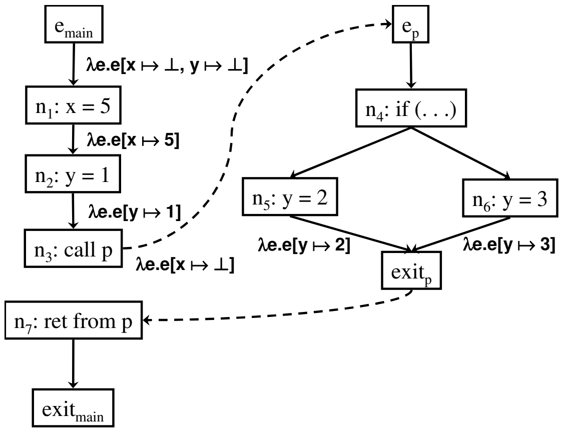
Definition 1. *A* **pushdown system** *is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is the set of states or control locations, Γ is the set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A* **configuration** *of $\mathcal{P}$ is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', u \rangle$ where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow_{\mathcal{P}}$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', u \rangle$ then $\langle p, \gamma u' \rangle \Rightarrow_{\mathcal{P}} \langle p', u u' \rangle$ for all $u' \in \Gamma^*$. The subscript $\mathcal{P}$ on the transition relation is omitted when it is clear from the context. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. For a set of configurations C, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, which are just backward and forward reachability under the transition relation $\Rightarrow$.*

We restrict the pushdown rules to have at most two stack symbols on the right-hand side. This means that for every rule $r \in \Delta$ of the form $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', u \rangle$, we have $|u| \leq 2$. This restriction does not decrease the power of pushdown systems because by increasing the number of stack symbols by a constant factor, an arbitrary pushdown system can be converted into one that satisfies this restriction [20]. Moreover, pushdown systems with at most two stack symbols on the right-hand side of each rule are sufficient for modeling control flow in programs. We use $\Delta_i \subseteq \Delta$ to denote the set of all rules with i stack symbols on the right-hand side.

It is instructive to see how a program's control flow can be modeled because even though the EWPDS model can work with any pushdown system, it is geared towards performing dataflow analysis in programs. The construction we present here is also followed in [18]. Let $(\mathcal{N}, \mathcal{E})$ be an interprocedural control flow graph where each *call* node is split into two nodes: one is the source of an interprocedural edge to the callee's

[2] Multi-level pointer analysis problems (the kind that occur in C, C++, and Java programs) can be safely approximated as single-level pointer-analysis problems [14].

entry node and the second is the target of an edge from the callee's exit node. $\mathcal{N}$ is the set of nodes in this graph and $\mathcal{E}$ is the set of control-flow edges. Fig. 1(a) shows an example of an interprocedural control-flow graph; Fig. 1(b) shows the pushdown system that models it. The PDS has a single state p, one stack symbol for each node in $\mathcal{N}$, and one rule for each edge in $\mathcal{E}$. We use Δ_1 rules to model intraprocedural edges, Δ_2 rules (also called *push* rules) for *call* edges, and Δ_0 rules (also called *pop* rules) for *return* edges. It is easy to see that a valid path in the program corresponds to a path in the pushdown system's transition system and vice versa.



(a)

(1) $\langle p, e_{main} \rangle \hookrightarrow \langle p, n_1 \rangle$
(2) $\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$
(3) $\langle p, n_2 \rangle \hookrightarrow \langle p, n_3 \rangle$
(4) $\langle p, n_3 \rangle \hookrightarrow \langle p, e_p\ n_7 \rangle$
(5) $\langle p, n_7 \rangle \hookrightarrow \langle p, exit_{main} \rangle$
(6) $\langle p, exit_{main} \rangle \hookrightarrow \langle p, \varepsilon \rangle$
(7) $\langle p, e_p \rangle \hookrightarrow \langle p, n_4 \rangle$
(8) $\langle p, n_4 \rangle \hookrightarrow \langle p, n_5 \rangle$
(9) $\langle p, n_4 \rangle \hookrightarrow \langle p, n_6 \rangle$
(10) $\langle p, n_5 \rangle \hookrightarrow \langle p, exit_p \rangle$
(11) $\langle p, n_6 \rangle \hookrightarrow \langle p, exit_p \rangle$
(12) $\langle p, exit_p \rangle \hookrightarrow \langle p, \varepsilon \rangle$

(b)

Fig. 1. (a) An interprocedural control flow graph. The e and *exit* nodes represent entry and exit points of procedures, respectively. x is a local variable of $main$ and y is a global variable. Dashed edges represent interprocedural control flow. Edge labels correspond to dataflow facts and are explained in §2.3. (b) A pushdown system that models the control flow of the graph shown in (a).

The number of configurations of a pushdown system is unbounded, so we use a finite automaton to describe a set of configurations.

Definition 2. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a pushdown system. A* $\mathcal{P}$**-automaton** *is a finite automaton $(Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, $\rightarrow \subseteq Q \times \Gamma \times Q$ is the transition relation, P is the set of initial states, and F is the set of final states of the automaton. We say that a configuration $\langle p, u \rangle$ is accepted by a $\mathcal{P}$-automaton if the automaton can accept u when it is started in state p (written as $p \xrightarrow{u}{}^* q$, where $q \in F$). A set of configurations is called* **regular** *if some $\mathcal{P}$-automaton accepts it.*

An important result is that for a regular set of configurations C, both $post^*(C)$ and $pre^*(C)$ are also regular sets of configurations [20, 3, 8].

2.2 Weighted Pushdown Systems

A weighted pushdown system is obtained by supplementing a pushdown system with a weight domain that is a bounded idempotent semiring [18, 4].

Definition 3. *A* **bounded idempotent semiring** *is a quintuple $(D, \oplus, \otimes, 0, 1)$, where D is a set whose elements are called* **weights***, 0 and 1 are elements of D, and $\oplus$ (the combine operation) and $\otimes$ (the extend operation) are binary operators on D such that*

1. $(D, \oplus)$ *is a commutative monoid with* 0 *as its neutral element, and where* $\oplus$ *is idempotent (i.e., for all* $a \in D$, $a \oplus a = a$*).*
2. $(D, \otimes)$ *is a monoid with the neutral element* 1.
3. $\otimes$ *distributes over* $\oplus$*, i.e., for all* $a, b, c \in D$ *we have*
   $$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ .$$
4. 0 *is an annihilator with respect to* $\otimes$*, i.e., for all* $a \in D$, $a \otimes 0 = 0 = 0 \otimes a$.
5. *In the partial order* $\sqsubseteq$ *defined by:* $\forall a, b \in D,\ a \sqsubseteq b$ *iff* $a \oplus b = a$*, there are no infinite descending chains.*

Definition 4. *A* **weighted pushdown system** *is a triple* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *where* $\mathcal{P} = (P, \Gamma, \Delta)$ *is a pushdown system,* $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ *is a bounded idempotent semiring and* $f : \Delta \to D$ *is a map that assigns a weight to each pushdown rule.*

Let $\sigma \in \Delta^*$ be a sequence of rules. Using f, we can associate a value to σ, i.e., if $\sigma = [r_1, \ldots, r_k]$, then we define $v(\sigma) \stackrel{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, for any two configurations c and c' of $\mathcal{P}$, we let $path(c, c')$ denote the set of all rule sequences $[r_1, \ldots, r_k]$ that transform c into c'. Weighted pushdown systems are geared towards solving the following two reachability problems.

Definition 5. *Let* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ *be a weighted pushdown system, where* $\mathcal{P} = (P, \Gamma, \Delta)$*, and let* $C \subseteq P \times \Gamma^*$ *be a regular set of configurations. The* **generalized pushdown predecessor** (GPP) **problem** *is to find for each* $c \in P \times \Gamma^*$*:*
$$\delta(c) \stackrel{\text{def}}{=} \bigoplus \{\, v(\sigma) \mid \sigma \in path(c, c'), c' \in C \,\}.$$
The **generalized pushdown successor** (GPS) **problem** *is to find for each* $c \in P \times \Gamma^*$*:*
$$\delta(c) \stackrel{\text{def}}{=} \bigoplus \{\, v(\sigma) \mid \sigma \in path(c', c), c' \in C \,\}.$$

2.3 Extended Weighted Pushdown Systems

The reachability problems defined in the previous section compute the value of a rule sequence by taking an extend of the weights of each of the rules in the sequence. However, when weighted pushdown systems are used for dataflow analysis of programs [18] then the rule sequences, in general, represent interprocedural paths in a program. To summarize the weight of such paths, we would have to maintain information about local variables of all unfinished procedures that appear on the path.

We lift weighted pushdown systems to handle local variables in much the same way that Knoop and Steffen [12] lifted conventional dataflow analysis to handle local variables. We allow for local variables to be stored at call sites and then use special merging functions to appropriately combine them with the value returned by a procedure. For a semiring $\mathcal{S}$ on domain D, define a *merging function* as follows:

Definition 6. *A function* $g : D \times D \to D$ *is a* **merging function** *with respect to a bounded idempotent semiring* $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ *if it satisfies the following properties.*
1. **Strictness.** *For all* $a \in D$, $g(0, a) = g(a, 0) = 0$.
2. **Distributivity.** *The function distributes over* $\oplus$*. For all* $a, b, c \in D$,
   $$g(a \oplus b, c) = g(a, c) \oplus g(b, c) \text{ and } g(a, b \oplus c) = g(a, b) \oplus g(a, c)$$
3. **Path Extension.**[3] *For all* $a, b, c \in D$, $g(a \otimes b, c) = a \otimes g(b, c)$.

[3] This property can be too restrictive in some cases; App. A discusses how this property may be dispensed with.

Definition 7. *An* **extended weighted pushdown system** *is a quadruple $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ where $(\mathcal{P}, \mathcal{S}, f)$ is a weighted pushdown system and $g : \Delta_2 \to \mathcal{G}$ assigns a merging function to each rule in Δ_2, where $\mathcal{G}$ is the set of all merging functions on the semiring $\mathcal{S}$. We will write g_r as a shorthand for $g(r)$.*

Note that a push rule has both a weight and a merging function associated with it. The merging functions are used to combine the effects of a called procedure with those made by the calling procedure just before the call. As an example, Figure 1 shows an interprocedural control flow graph and the pushdown system that can be used to represent it. We can perform constant propagation (with uninterpreted expressions) on the graph by assigning a weight to each pushdown rule. Let V be the set of all variables in a program and $(\mathbb{Z}_\perp, \sqsubseteq, \sqcap)$ with $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$ be the standard constant-propagation semilattice: $\perp \sqsubseteq c$ for all $c \in \mathbb{Z}$ and $\sqcap$ is the greatest-lower-bound operation in this partial order. $\perp$ stands for "not-a-constant". The weight semiring is $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ where $D = (Env \to Env)$ is the set of all environment transformers with an environment being a mapping for all variables: $Env = (V \to \mathbb{Z}_\perp) \cup \{\top\}$. We use $\top$ to denote an infeasible environment. Furthermore, we restrict the set D to contain only $\top$-strict transformers, i.e., for all $d \in D$, $d(\top) = \top$. We can extend the meet operation to environments by taking meet componentwise.

$$env_1 \sqcap env_2 = \begin{cases} env_1 & \text{if } env_2 = \top \\ env_2 & \text{if } env_1 = \top \\ \lambda v.(env_1(v) \sqcap env_2(v)) & \text{otherwise} \end{cases}$$

The semiring operations and constants are defined as follows:

$$0 = \lambda e.\top \qquad w_1 \oplus w_2 = \lambda e.(w_1(e) \sqcap w_2(e))$$
$$1 = \lambda e.e \qquad w_1 \otimes w_2 = w_2 \circ w_1$$

The weights for the PDS that models the program in Fig. 1 are shown as edge labels. A weight of the form $\lambda e.e[x \mapsto 5]$ returns an environment that agrees with the argument, except that x is bound to 5. The environment $\top$ cannot be updated, and thus $(\lambda e.e[x \mapsto 5])\top = \top$.

The merging function for call site n_3 will receive two environment transformers: one that summarizes the effect of the caller from its entry point to the call site (e_{main} to n_3) and one that summarizes the effect of the called procedure (e_p to $exit_p$). It then has produce the transformer that summarizes the effect of the caller from its entry point to the return site (e_{main} to n_7). We define it as follows:

$$g(w_1, w_2) \;=\; \textbf{if } (w_1 = 0 \text{ or } w_2 = 0) \textbf{ then } 0$$
$$\textbf{else } \lambda e.e[x \mapsto w_1(e)(x), y \mapsto (w_1 \otimes w_2)(e)(y)]$$

It copies over the value of the local variable x from the call site, and gets the value of y from the called procedure. Because the merging function has access to the environment transformer just before the call, we do not have to pass the value of local variable x into procedure p. Hence the call stops tracking the value of x using the weight $\lambda e.e[x \mapsto \perp]$.

To formalize this, we redefine the generalized pushdown predecessor and successor problem by changing the definition of the value of a rule sequence. If $\sigma \in \Delta^*$ with $\sigma = [r_1, r_2, \cdots, r_k]$ then let $(r\ \sigma)$ denote the sequence $[r, r_1, \cdots, r_k]$. Also, let $[\ ]$ denote the empty sequence. Consider the context-free grammar shown in Fig. 2.

σ_s is simply R_1^*. σ_b represents a *balanced* sequence of rules that have matched calls (R_2) and returns (R_0) with any number of rules from Δ_1 inbetween. σ_i is just $(R_2 \mid \sigma_b)^+$ in regular-language terminology, and represents sequences that increase

$$
\begin{array}{lll}
R_0 \rightarrow r \ \ (r \in \Delta_0) & \sigma_s \rightarrow [\,] \mid R_1 \mid \sigma_s\ \sigma_s & \sigma_i \rightarrow R_2 \mid \sigma_b \mid \sigma_i\ \sigma_i \\
R_1 \rightarrow r \ \ (r \in \Delta_1) & \sigma_b \rightarrow \sigma_s \mid \sigma_b\ \sigma_b & \sigma_d \rightarrow R_0 \mid \sigma_b \mid \sigma_d\ \sigma_d \\
R_2 \rightarrow r \ \ (r \in \Delta_2) & \quad\ \mid\ R_2\ \sigma_b\ R_0 & \sigma_a \rightarrow \sigma_d\ \sigma_i
\end{array}
$$

Fig. 2. Grammar used for parsing rule sequences. The start symbol of the grammar is σ_a.

stack height. σ_d is $(R_0 \mid \sigma_b)^+$ and represents sequences that decrease stack height. σ_a can derive any rule sequence. We use this grammar to define the value of a rule sequence.

Definition 8. *Given an* EWPDS $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$*, we define the* **value** *of a sequence of rules* $\sigma \in \Delta^*$ *by first parsing the sequence according to the above grammar and then giving a meaning to each production rule.*

1. $v(r) = f(r)$
2. $v([\,]) = 1$
3. $v(\sigma_s\ \sigma_s) = v(\sigma_s) \otimes v(\sigma_s)$
4. $v(\sigma_b\ \sigma_b) = v(\sigma_b) \otimes v(\sigma_b)$
5. $v(R_2\ \sigma_b\ R_0) = g_{R_2}(1, v(\sigma_b) \otimes v(R_0))$
6. $v(\sigma_d\ \sigma_d) = v(\sigma_d) \otimes v(\sigma_d)$
7. $v(\sigma_i\ \sigma_i) = v(\sigma_i) \otimes v(\sigma_i)$
8. $v(\sigma_d\ \sigma_i) = v(\sigma_d) \otimes v(\sigma_i)$

Here we have used g_{R_2} *as a shorthand for* g_r *where* r *is the terminal derived by* R_2*.*

The main thing to note in the above definition is the application of merging functions on balanced sequences. Because the grammar presented in Fig. 2 is ambiguous, there might be many parsings of the same rule sequence, but all of them would produce the same value because the extend operation is associative and there is a unique way to balance R_2s with R_0s.

The generalized pushdown problems GPP and GPS for EWPDS are the same as those for WPDS except for the changed definition of the value of a rule sequence. If we let each merging function be $g_r(w_1, w_2) = w_1 \otimes f(r) \otimes w_2$, then the EWPDS reduces to a WPDS. This justifies calling our model an *extended* weighted pushdown system.

3 Solving Reachability Problems in EWPDSs

In this section, we present algorithms to solve the generalized reachability problems for EWPDSs. Throughout this section, let $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ be an EWPDS where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system and $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ is the weight domain. Let C be a fixed regular set of configurations that is recognized by a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ such that $\mathcal{A}$ has no transition leading to an initial state. Note that any automaton can be converted to an equivalent one that has no transition into an initial state by duplicating the initial states. We also assume that $\mathcal{A}$ has no ε-transitions.

As in the case of weighted pushdown systems, we construct an annotated automaton from which $\delta(c)$ can be read off efficiently. This automaton is the same as the automaton constructed for simple pushdown reachability [20], except for the annotations. We will not show the calculation of witness annotations because they are obtained in exactly the same way as for weighted pushdown systems [18]. This is because witnesses record the paths that justify a weight and not how the values of those paths were calculated.

3.1 Solving GPP

To solve GPP, we take as input the $\mathcal{P}$-automaton $\mathcal{A}$ that describes the set of configurations on which we want to query the EWPDS. As output, we create an automaton $\mathcal{A}_{pre^*}$ with weights as annotations on transitions, and then read off the values of $\delta(c)$ from the automaton. The algorithm is based on the saturation rule shown below. Starting with the automaton $\mathcal{A}$, we keep applying this rule until it can no longer be applied. Termination is guaranteed because there are a finite number of transitions and the height of the weight domain is bounded as well. For each transition in the automaton being created, we store the weight on it using function l. The saturation rule is the same as that for predecessor reachability in ordinary pushdown systems, except for the weights, and is different from the one for weighted pushdown systems only in the last case, where a merging function is applied.

- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$, then update the annotation on $t = (p, \gamma, p')$ to $l(t) := l(t) \oplus f(r)$. We assume $l(t) = 0$ if the transition t did not exist before.
- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and there is a transition $t = (p', \gamma', q)$, then update the annotation on $t' = (p, \gamma, q)$ to $l(t') := l(t') \oplus (f(r) \otimes l(t))$.
- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and there are transitions $t = (p', \gamma', q_1)$ and $t' = (q_1, \gamma'', q_2)$, then update the annotation on $t'' = (p, \gamma, q_2)$ to
$$l(t'') := l(t'') \oplus \begin{cases} f(r) \otimes l(t) \otimes l(t') & \text{if } q_1 \notin P \\ g_r(1, l(t)) \otimes l(t') & \text{otherwise} \end{cases}$$

For convenience, we will write a transition $t = (p, \gamma, q)$ in $\mathcal{A}_{pre^*}$ with $l(t) = w$ as $p \xrightarrow[w]{\gamma} q$. Define the value of a path $q_1 \xrightarrow[w_1]{\gamma_1} q_2 \cdots \xrightarrow[w_n]{\gamma_n} q_{n+1}$ to be $w_1 \otimes w_2 \cdots \otimes w_n$. The following theorem shows how $\delta(c)$ is calculated.

Theorem 1. *For a configuration $c = \langle p, \gamma_1\gamma_2 \cdots \gamma_n \rangle$, $\delta(c)$ is the combine of the values of all accepting paths $p \xrightarrow{\gamma_1} q_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_n} q_n,\ q_n \in F$ in $\mathcal{A}_{pre^*}$.*

We can calculate $\delta(c)$ efficiently using an algorithm similar to the simulation algorithm for NFAs (cf. [1, Algorithm 3.4]).

3.2 Solving GPS

For this section, we shall assume that we can have at most one rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ for each combination of p',γ', and γ''. This involves no loss of generality because we can replace a rule $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ with two rules: (a) $r' = \langle p, \gamma \rangle \hookrightarrow \langle p_r, \gamma'\gamma'' \rangle$ with weight $f(r)$ and merging function g_r, and (b) $r'' = \langle p_r, \gamma' \rangle \hookrightarrow \langle p', \gamma' \rangle$ with weight 1, where p_r is a new state. This replacement does not change the reachability problem's answers. Let *lookup*(p', γ', γ'') be a function that returns the unique push rule associated with a triple (p', γ', γ'') if there is one.

Before presenting the algorithm, let us consider an operational definition of the value of a rule sequence. The importance of this alternative definition is that it shows the correspondence with the call semantics of a program. For each interprocedural path in a program, we define a stack of weights that contains a weight for each unfinished call in the path. Elements of the stack are from the set $D \times D \times \Delta_2$ (recall that Δ_2 was defined as the set of all push rules in Δ), where (w_1, w_2, r) signifies that (i) a call was made using rule r, (ii) the weight at the time of the call was w_1, and (iii) w_2 was the weight on the call rule.

Let *STACK* $= D.(D \times D \times \Delta_2)^*$ be the set of all nonempty stacks where the topmost element is from D and the rest are from $(D \times D \times \Delta_2)$. We will write an element $(w_1, w_2, r) \in D \times D \times \Delta_2$ as $(w_1, w_2)_r$. For each rule $r \in \Delta$ of the form

$\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, $u \in \Gamma^*$, we will associate a function $[\![r]\!]$: *STACK* $\rightarrow$ *STACK*. Let $S \in (D \times D \times \Delta_2)^*$.

- If r has one symbol on the right-hand side ($|u| = 1$), then accumulate its weight on the top of the stack: $[\![r]\!]\ (w_1\ S) = ((w_1 \otimes f(r))\ S)$
- If r has two symbols on the right-hand side ($|u| = 2$), then save the weight of the push rule as well as the push rule itself on the stack and start a fresh entry on the top of the stack: $[\![r]\!]\ (w_1\ S) = (1\ (w_1, f(r))_r\ S)$
- If r has no symbols on the right-hand side ($|u| = 0$), then apply the appropriate merging function if there is something pushed on the stack. Otherwise, r represents an unbalanced pop rule and simply accumulate its weight on the stack. Note that we drop the weight of the push rule when we apply the merging function in accordance with case 5 of Defn. 8.

$$[\![r]\!]\ (w_1\ (w_2, w_3)_{r_1}\ S) = ((g_{r_1}(w_2, w_1 \otimes f(r))\ S) \qquad (1)$$
$$[\![r]\!]\ (w_1) = (w_1 \otimes f(r))$$

For a sequence of rules $\sigma = [r_1, r_2, \cdots, r_n]$, define $[\![\sigma]\!] = [\![[r_2, \cdots, r_n]]\!] \circ [\![r_1]\!]$. Let *flatten* : *STACK* $\rightarrow D$ be an operation that computes a weight from a stack as follows:

$$\mathit{flatten}(w_1\ S) = \mathit{flatten}'(S) \otimes w_1 \qquad \begin{aligned} \mathit{flatten}'((\,)) &= 1 \\ \mathit{flatten}'((w_1, w_2)_r\ S) &= \mathit{flatten}'(S) \otimes (w_1 \otimes w_2) \end{aligned}$$

Example 1. Consider the rule sequence σ that takes the program in Fig. 1 from e_{main} to $exit_p$ via node n_5. If we apply $[\![\sigma]\!]$ to a stack containing just 1, we get a stack of height 2 as follows: $[\![\sigma]\!](1) = ((\lambda e.e[y \mapsto 2])\ (\lambda e.e[x \mapsto 5, y \mapsto 1], \lambda e.e[x \mapsto \bot])_r)$, where r is the push rule that calls procedure *p* (Rule 4 in Fig. 1(b)). The top of the stack is the weight computed inside *p* (Rules $7, 8, 10$), and the bottom of the stack contains a pair of weights: the first component is the weight computed in *main* just before the call (Rules $1, 2, 3$); the second component is just the weight on the call rule r. If we apply the *flatten* operation on this stack, we get the weight $\lambda e.e[x \mapsto \bot, y \mapsto 2]$ which is exactly the value $v(\sigma)$. When we apply the pop rule r' (Rule 12) to this stack, we get:

$$\begin{aligned} [\![\sigma\ r']\!](1) &= [\![r']\!] \circ [\![\sigma]\!](1) \\ &= (g_r(\lambda e.e[x \mapsto 5, y \mapsto 1], \lambda e.e[y \mapsto 2])) \\ &= (\lambda e.e[x \mapsto 5, y \mapsto 2]) \end{aligned}$$

Again, applying *flatten* on this stack gives us $v(\sigma\ r')$. The following lemma formalizes the equivalence between $[\![\sigma]\!]$ and $v(\sigma)$.

Lemma 1. *For any valid sequence of rules σ ($\sigma \in path(c, c')$ for some configurations c and c'), $[\![\sigma]\!]\ (1) = S$ such that flatten$(S) = v(\sigma)$.*

Corollary 1. *For a configuration c, let $\delta_S(c) \subseteq$ STACK be defined as follows:*

$$\delta_S(c) = \{[\![\sigma]\!](1) \mid \sigma \in paths(c', c),\ c' \in C\}.$$

Let C be the set of configurations described by the $\mathcal{P}$-automaton $\mathcal{A}$. Then

$$\delta(c) = \oplus\{\mathit{flatten}(S) \mid S \in \delta_S(c)\}.$$

The above corollary shows that $\delta_S(c)$ has enough information to compute $\delta(c)$ directly. To solve the pushdown successor problem, we take the input $\mathcal{P}$-automaton $\mathcal{A}$ that describes a set of configurations and create an annotated $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ (one that has weights as annotations on transitions) from which we can read off the value of $\delta(c)$

for any configuration c. The algorithm is again based on a saturation rule. For each transition in the automaton being created, we have a function l that stores the weight on the transition. Based on the above operational definition of the value of a path, we would create $\mathcal{A}_{post^*}$ on pairs of weights, that is, over the semiring $(D \times D, \oplus, \otimes, (0,0), (1,1))$ where $\oplus$ and $\otimes$ are defined component wise. Also, we introduce a new state for each push rule. So the states of $\mathcal{A}_{post^*}$ are $Q \cup \{q_{p',\gamma'} \mid \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta\}$. Let Q' be the set of new states added. The saturation rule is shown in Fig. 3. To see what the saturation rule is doing, consider a path in $\mathcal{A}_{post^*}$: $\tau = q_1 \xrightarrow{\gamma_1} q_2 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_n} q_{n+1}$. As an invariant of our algorithm, we would have $q_1 \in (P \cup Q')$; $q_2, \cdots, q_k \in Q'$; and $q_{k+1}, \cdots, q_{n+1} \in (Q - P)$ for some $0 \leq k \leq n+1$. This is because of the fact that we never create transitions from a state in P to a state in P, or from a state in Q' to a state in P, or from a state in $Q - P$ to a state in $P \cup Q'$. Now define a new transition label $l'(t)$ as follows: $l'(p, \gamma, q) = \mathit{lookup}(p', \gamma', \gamma)$ if $p \equiv q_{p',\gamma'}$.
Then the path τ describes the *STACK* $\mathit{vpath}(\tau) = (l_1(t_1) \;\; l(t_2)_{l'(t_2)} \cdots \;\; l(t_k)_{l'(t_k)})$ where $t_i = (q_i, \gamma_i, q_{i+1})$ and $l_1(t)$ is the first component projected out of the weight-pair $l(t)$. This means that each path in $\mathcal{A}_{post^*}$ represents a *STACK* and all the saturation algorithm does is to make the automaton rich enough to encode all *STACK*s in $\delta_S(c)$ for all configurations c. The first and third cases of the saturation rule can be seen as applying $[\![r]\!]$ for rules with one and two stack symbols on the right-hand side, respectively. Applying the fourth case immediately after the second case can be seen as applying $[\![r]\!]$ for pop rules.

- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and there is a transition $t = (p, \gamma, q)$ with annotation $l(t)$, then update the annotation on transition $t' = (p', \gamma', q)$ to $l(t') := l(t') \oplus (l(t) \otimes (f(r), 1))$. We assume $l(t') = (0, 0)$ if the transition did not exist before.
- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ and there is a transition $t = (p, \gamma, q)$ with annotation $l(t)$, then update the annotation on transition $t' = (p', \varepsilon, q)$ to $l(t') := l(t') \oplus (l(t) \otimes (f(r), 1))$.
- If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and there is a transition $t = (p, \gamma, q)$ with annotation $l(t)$ then let $t' = (p', \gamma', q_{p',\gamma'})$, $t'' = (q_{p',\gamma'}, \gamma'', q)$ and update annotations on them.
$$\begin{aligned} l(t') &:= l(t') \oplus (1,1) \\ l(t'') &:= l(t'') \oplus (l(t) \otimes (1, f(r))) \end{aligned}$$
- If there are transitions $t = (p, \varepsilon, q)$ and $t' = (q, \gamma', q')$ with annotations $l(t) = (w_1, w_2)$ and $l(t') = (w_3, w_4)$ then update the annotation on the transition $t'' = (p, \gamma', q')$ to $l(t'') := l(t'') \oplus w$ where w is defined as follows:
$$w = \begin{cases} (g_{\mathit{lookup}(p',\gamma',\gamma'')}(w_3, w_1), 1) & \text{if } q \equiv q_{p',\gamma'} \\ l(t') \otimes l(t) & \text{otherwise} \end{cases}$$

Fig. 3. Saturation rule for constructing $\mathcal{A}_{post^*}$ from $\mathcal{A}$.

Theorem 2. *For a configuration $c = \langle p, u \rangle$, we have,*
$$\delta(c) = \oplus\{\mathit{flatten}(\mathit{vpath}(\sigma_t)) \mid \sigma_t \in paths(p, u, q_f), q_f \in F\}$$
where $paths(p, u, q_f)$ denotes the set of all paths of transitions in $\mathcal{A}_{post^*}$ that go from p to q_f on input u, i.e., $p \xrightarrow{u}{}^* q_f$.

An easy way of computing the combine in the above theorem is to replace annotation $l(t)$ on each transition t with $l_1(t) \otimes l_2(t)$, the extend of the two weight components of $l(t)$, and then use standard NFA simulation algorithms (cf. [1, Algorithm 3.4]) as we would use for $\mathcal{A}_{pre^*}$.

4 Interprocedural Meet Over All Paths

In this section, we show how extended weighted pushdown systems can be used to compute the interprocedural-meet-over-all-paths (IMOVP) solution for a given dataflow analysis problem. We will first define the IMOVP strategy as described in [12] and then show how to solve it using an EWPDS.

We are given a meet semilattice $(\mathcal{C}, \sqcap)$ describing dataflow facts and the interprocedural-control-flow graph of a program $(\mathcal{N}, \mathcal{E})$ where $\mathcal{N}_C, \mathcal{N}_R \subseteq \mathcal{N}$ are the call and return nodes, respectively. We are also given a semantic transformer for each node in the program: $[\![\ ]\!] : \mathcal{N} \to (\mathcal{C} \to \mathcal{C})$, which represents (i.e., over-approximates) the effect of executing a statement in the program. Let $\mathit{STK} = \mathcal{C}^+$ be the set of all nonempty stacks with elements from $\mathcal{C}$. *STK* is used as an abstract representation of the run-time stack of a program. Define the following operations on stacks.

newstack : $\mathcal{C} \to \mathit{STK}$	creates a new stack with a single element
push : $\mathit{STK} \times \mathcal{C} \to \mathit{STK}$	pushes a new element on top of the stack
pop : $\mathit{STK} \to \mathit{STK}$	removes the top most element of the stack
top : $\mathit{STK} \to \mathcal{C}$	returns the top most element of the stack

We can now describe the interprocedural semantic transformer for each program node: $[\![\ ]\!]^* : \mathcal{N} \to (\mathit{STK} \to \mathit{STK})$. For $stk \in \mathit{STK}$,

$$[\![n]\!]^*(stk) = \begin{cases} \mathit{push}(\mathit{pop}(stk), [\![n]\!](\mathit{top}(stk))) & \text{if } n \in \mathcal{N} - (\mathcal{N}_C \cup \mathcal{N}_R) \\ \mathit{push}(stk, [\![n]\!](\mathit{top}(stk))) & \text{if } n \in \mathcal{N}_C \\ \mathit{push}(\mathit{pop}(\mathit{pop}(stk)), \mathcal{R}_n(\mathit{top}(\mathit{pop}(stk)), [\![n]\!](\mathit{top}(stk)))) & \text{if } n \in \mathcal{N}_r \end{cases}$$

where $\mathcal{R}_n : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a merging function like we have in EWPDSs. It is applied on dataflow value computed by the called procedure ($[\![n]\!](\mathit{top}(stk))$) and the value computed by the caller at the time of the call ($\mathit{top}(\mathit{pop}(stk))$). This definition assumes that a dataflow fact in $\mathcal{C}$ contains all information that is required by a procedure so that each transformer has to look at only the top of the stack passed to it – except for return nodes, where we look at the top two elements of the stack. Now, define a path transformer as follows. If $p = [n_1\ n_2 \cdots n_k]$ is a valid interprocedural path in the program then $[\![p]\!]^* = [\![[n_2 \cdots n_k]]\!]^* \circ [\![n_1]\!]^*$. This leads to the following definition.

Definition 9. *[12] If $s \in \mathcal{N}$ is the starting node of a program, then for $c_0 \in \mathcal{C}$ and $n \in \mathcal{N}$, the interprocedural-meet-over-all-paths value is defined as follows:*

$$\text{IMOVP}_{c_0}(n) = \sqcap\{[\![p]\!]^*(\mathit{newstack}(c_0)) \mid p \in \mathbf{IP}(s, n)\}$$

where $\mathbf{IP}(s, n)$ represents the set of all valid interprocedural paths from s to n and meet of stacks is just the meet of their topmost values: $stk_1 \sqcap stk_2 = \mathit{top}(stk_1) \sqcap \mathit{top}(stk_2)$.

We now construct an EWPDS $\mathcal{W}_e = (\mathcal{P}, \mathcal{S}, f, g)$ to compute this value when $\mathcal{C}$ has no infinite descending chains, all semantic transformers $[\![n]\!]$ are distributive, and all merging relations $\mathcal{R}_n$ are distributive in each of their arguments. Define a semiring $\mathcal{S} = (D, \oplus, \otimes, 0, 1)$ as $D = [\mathcal{C} \to \mathcal{C}] \cup \{0\}$, which consists of the set of all distributive functions on $\mathcal{C}$ and a special function 0. For $a, b \in D$,

$$a \oplus b = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ (a \sqcap b) & \text{otherwise} \end{cases} \qquad a \otimes b = \begin{cases} 0 & \text{if } a = 0 \text{ or } b = 0 \\ (b \circ a) & \text{otherwise} \end{cases} \qquad 1 = \lambda c.c$$

The pushdown system $\mathcal{P}$ is $(\{q\}, \mathcal{N}, \Delta)$ where Δ is constructed by including a rule for each edge in $\mathcal{E}$. First, let $\mathcal{E}_{\mathit{intra}} \subseteq \mathcal{E}$ be the intraprocedural edges and $\mathcal{E}_{\mathit{inter}} \subseteq \mathcal{E}$ be the interprocedural (call and return) edges. Then include the following rules in Δ.

1. For $(n,m) \in \mathcal{E}_{\mathit{intra}}$, include the rule $r = \langle q, n\rangle \hookrightarrow \langle q, m\rangle$ with $f(r) = [\![n]\!]$.
2. For $n \in \mathcal{N}_C$, $(n,m) \in \mathcal{E}_{\mathit{inter}}$ with $n_R \in \mathcal{N}_R$ being the return site for the call at n, include the rule $r = \langle q, n\rangle \hookrightarrow \langle q, m\, n_R\rangle$ with $f(r) = [\![n]\!]$ and
$$g_r(a,b) = \lambda c.\mathcal{R}_n(a(c), (a \otimes [\![n]\!] \otimes b \otimes [\![n_R]\!])(c)).$$
3. For $n \in \mathcal{N}$, if it is an exit node of a procedure, include the rule $r = \langle q, n\rangle \hookrightarrow \langle q, \varepsilon\rangle$ with $f(r) = [\![n]\!]$.

A small technical detail here is that the merging functions defined above need not satisfy the path-extension property given in Defn. 6. In App. A, we give an alternative definition of how to assign a weight to a rule sequence such that the path-extension property is no longer a limitation. This leads us to the following theorem.

Theorem 3. *Let $\mathcal{A}$ be a $\mathcal{P}$-automaton that accepts just the configuration $\langle q, s\rangle$, where s is the starting point of the program and let $\mathcal{A}_{post^*}$ be the automaton obtained by using the saturation rule shown in Fig. 3 on $\mathcal{A}$. Then if $c_0 \in \mathcal{C}$, $n \in \mathcal{N}$, $\delta(c)$ is read off $\mathcal{A}_{post^*}$ in accordance with Thm. 2, we have,*
$$\mathrm{IMOVP}_{c_0}(n) = [\oplus\{\delta(\langle q, n\, u\rangle \mid u \in \Gamma^*\}](c_0).$$
If $L \subseteq \Gamma^*$ is a regular language of stack configurations then $\mathrm{IMOVP}_{c_0}(n, L)$*, which is the* IMOVP *value restricted to only those paths that end in configurations described by L, can be calculated as follows:*
$$\mathrm{IMOVP}_{c_0}(n, L) = [\oplus\{\delta(\langle q, n\, u\rangle \mid u \in L\}](c_0).$$

In case the semantic transformers $[\![.]\!]$ and $\mathcal{R}_n$ are not distributive but only monotonic, then the two combines in Thm. 3 safely approximate $\mathrm{IMOVP}_{c_0}(n)$ and $\mathrm{IMOVP}_{c_0}(n, L)$, respectively. We do not present the proof in this paper, but the essential idea carries over from solving monotonic dataflow problems in WPDSs [18].

5 Experimental Results

In [2], Balakrishnan and Reps present an algorithm to analyze memory accesses in x86 code. Its goal is to determine an over-approximation of the set of values/memory-addresses that each register and memory location holds at each program point. The core dataflow-analysis algorithm used, called value-set analysis (VSA), is not relational, i.e., it does not keep track of the relationships that hold among registers and memory locations. However, when interpreting conditional branches, specifically those that implement loops, it is important to know such relationships. Hence, a separate affine-relation analysis (ARA) is performed to recover affine relations that hold among the registers at conditional branch points; those affine relations are then used to interpret conditional branches during VSA. ARA recovers affine relations involving registers only, because recovering affine relations involving memory locations would require points-to information, which is not available until the end of VSA. ARA is implemented using the affine-relation domain from [16] as a weight domain. It is based on machine arithmetic, i.e., arithmetic module 2^{32}, and is able to take care of overflow.

Before each call instruction, a subset of the registers is saved on the stack, either by the caller or the callee, and restored at the return. Such registers are called the *caller-save* and *callee-save* registers. Because ARA only keeps track of information involving registers, when ARA is implemented using a WPDS, all affine relations involving caller-save and callee-save registers are lost at a call. We used an EWPDS to preserve them across calls by treating caller-save and callee-save registers as local variables at a call; i.e., the values of caller-save and callee-save registers after the call are set to the values before the call and the values of other registers are set to the values at the exit node of the callee.

The results are shown in Tab. 1. The column labeled ‘Branches with useful information’ refers to the number of branch points at which ARA recovered at least one affine relation. The last column shows the number of branch points at which ARA implemented via an EWPDS recovered more affine relations when compared to ARA implemented via a WPDS. Tab. 1 shows that the information recovered by EWPDS is better in 30% to 63% of the branch points that had useful information. The EWPDS version is somewhat slower, but uses less space; this is probably due to the fact that the dataflow transformer from [16] for ‘spoiling’ the affine relations that involve a given register uses twice the space of a transformer that preserves such relations.

					Memory (MB)		Time (s)		Branches with useful information		
Prog	Insts	Procs	Branches	Calls	WPDS	EWPDS	WPDS	EWPDS	WPDS	EWPDS	Improvement
mplayer2	58452	608	4608	2481	27	6	8	9	137	192	57 (42%)
print	96096	955	8028	4013	61	19	20	23	601	889	313 (52%)
attrib	96375	956	8076	4000	40	8	12	13	306	380	93 (30%)
tracert	101149	1008	8501	4271	70	22	24	27	659	1021	387 (59%)
finger	101814	1032	8505	4324	70	23	24	30	627	999	397 (63%)
lpr	131721	1347	10641	5636	102	36	36	46	1076	1692	655 (61%)
rsh	132355	1369	10658	5743	104	36	37	45	1073	1661	616 (57%)
javac	135978	1397	10899	5854	118	43	44	58	1376	2001	666 (48%)
ftp	150264	1588	12099	6833	121	42	43	61	1364	2008	675 (49%)
winhlp32	179488	1911	15296	7845	156	58	62	98	2105	2990	918 (44%)
regsvr32	297648	3416	23035	13265	279	117	145	193	3418	5226	1879 (55%)
notepad	421044	4922	32608	20018	328	124	147	390	3882	5793	1988 (51%)
cmd	482919	5595	37989	24008	369	144	175	444	4656	6856	2337 (50%)

Table 1. Comparison of ARA results implemented using EWPDS versus WPDS.

6 Related Work

Some libraries/tools based on model-checking pushdown systems for dataflow analysis are MOPED [7, 21], WPDS [18], and WPDS++ [11]. Weighted pushdown systems have been used for finding uninitialized variables, live variables, linear constant propagation, and the detection of affine relationships. In each of these cases, local variables are handled by introducing special paths in the transition system of the PDS that models the program. These paths skip call sites to avoid passing local variables to the callee. This leads to imprecision by breaking existing relationships between local and global variables. Besides dataflow analysis, WPDSs have also been used for generalized authorization problems [22].

MOPED has been used for performing relational dataflow analysis, but only for finite abstract domains. Its basic approach is to embed the abstract transformer of each program statement into the rules of the pushdown system that models the program. This contrasts with WPDSs, where the abstract transformer is a separate weight associated with a pushdown rule. MOPED associates global variables with states of the PDS and local variables with its stack symbols. Then the stack of the PDS simulates the run-time stack of the program and maintains a different copy of the local variables for each procedure invocation. A simple pushdown reachability query can be used to compute the required dataflow facts. The disadvantage of that approach is that it cannot handle infinite-size abstract domains because then associating an abstract transformer with a pushdown rule would create an infinite number of pushdown rules. An EWPDS is capable of performing an analysis on infinite-size abstract domains as well. The domain used for copy-constant propagation in §2.3 is one such example.

Besides dataflow analysis, model-checking of pushdown systems has also been used for verifying security properties in programs [6, 9, 5]. Like WPDSs, we can use EWPDS for this purpose, but with added precision that comes due to the presence of merging functions.

A result we have not presented in this paper is that EWPDSs can be used for single-level pointer analysis, which enables us to answer stack-qualified aliasing queries. Stack-qualified aliasing has been studied before by Whaley and Lam [24]. However, for recursive programs, they collapse the strongly connected components in the call graph. We do not make any such approximation, and can also answer aliasing queries with respect to a language of stack configurations instead of just a single stack configuration.

The idea behind the transition from a WPDS to an EWPDS is that we attach extra meaning to each run of the pushdown system. We look at a run as a *tree* of matching calls and returns that push and pop values on the run-time stack of the program. This treatment of a program run has also been explored by Müller-Olm and Seidl [15] in an interprocedural dataflow-analysis algorithm to identify the set of all affine relationships. They explicitly match calls and returns to avoid passing relations involving local variables to different procedures. This allowed us to to directly translate their work into an EWPDS, which we have used for the experiments in §5.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Int. Conf. on Comp. Construct.*, 2004.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*, pages 135–150. Springer-Verlag, 1997.
4. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Symp. on Princ. of Prog. Lang.*, pages 62–73, 2003.
5. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.*, November 2002.
6. J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *TACAS*, pages 306–339, 2001.
7. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. CAV'01*, LNCS 2102, pages 324–336. Springer-Verlag, 2001.
8. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.
9. T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.
10. J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7(3):305–318, 1977.
11. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004.

12. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct.*, pages 125–140, 1992.
13. W. Landi and B. Ryder. Pointer-induced aliasing: A problem classification. In *Symp. on Princ. of Prog. Lang.*, pages 93–103, 1991.
14. W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Conf. on Prog. Lang. Design and Impl.*, pages 235–248, 1992.
15. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *Symp. on Princ. of Prog. Lang.*, 2004.
16. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.
17. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, pages 49–61, 1995.
18. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Static Analysis Symp.*, pages 189–213, 2003.
19. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.
20. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
21. S. Schwoon. Moped, 2002. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
22. S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In *Comp. Sec. Found. Workshop*, Wash., DC, 2003. IEEE Comp. Soc.
23. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
24. J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conf. on Prog. Lang. Design and Impl.*, pages 131–144, 2004.

A Relaxing Merging Function Requirements

This appendix discusses what happens when merging functions do not satisfy the third property in Defn. 6. The pre^* algorithm of §3.1 (used for creating $\mathcal{A}_{pre^*}$) would still compute the correct values for $\delta(c)$ because it parses rule sequences using the grammar from Defn. 8, but the $post^*$ algorithm of §3.2 (used for creating $\mathcal{A}_{post^*}$) would not work because it utilizes a different grammar and relies on the path-extension property to compute the correct value. Instead of trying to modify the $post^*$ algorithm, we will introduce an alternative definition of the value of a rule sequence that is suited for the cases when merging functions do not satisfy the path-extension property. The definition involves changing the productions and valuations of balanced sequences as follows:

$$\begin{array}{lll} \sigma_{b'} \rightarrow [\,] & v(\sigma_{b'}\ \sigma_{b'}) & = v(\sigma_{b'}) \otimes v(\sigma_{b'}) \\ \quad\ \mid\ \sigma_b\ R_2\ \sigma_b\ R_0 & v(\sigma_b\ R_2\ \sigma_b\ R_0) & = g_{R_2}(v(\sigma_b), v(\sigma_b) \otimes v(R_0)) \\ \sigma_b\ \rightarrow \sigma_{b'}\ \sigma_s & v(\sigma_{b'}\ \sigma_s) & = v(\sigma_{b'}) \otimes v(\sigma_s) \end{array} \qquad (2)$$

The value of a rule sequence as defined above is the same as the value defined by Defn. 8 when merging functions satisfy the path-extension property. In the absence of the property, we need to make sure that merging functions are applied to the weight computed in the caller just before the call and the weight computed by the callee. We enforce this using Eqn. (2). The *STACK* values that are calculated for rule sequences in §3.2 also does the same in Eqn. (1)[Pg. 8]. This means that Lem. 1 still holds and the $post^*$ algorithm correctly solves this more general version of GPS. However, the pre^* algorithm is closely based on Defn. 8 and does not solve the generalized version of GPP based on the above alternative definition.