Work in Progress: A Unified Algebraic Framework Of Program Analyses

MARTIN RINARD, MIT EECS and MIT CSAIL
HENNY SIPMA, Aarno Labs
THOMAS BOURGEAT, MIT EECS and MIT CSAIL

Program analyses have traditionally been formulated as point solutions to specific program analysis problems. We present a comprehensive, unified framework that places program analyses within an algebraic lattice structure based on the program traces that each analysis identifies. In this framework each program analysis is characterized by the set of program traces that it identifies, with program analyses ordered by reverse subset inclusion $\supseteq$ over sets of identified program traces. With this order, the collection of program analyses comprises a lattice with least upper bound $\cap$ and greatest lower bound $\cup$.

1 INTRODUCTION

Program analysis is a classic field in computer science. Initially developed for optimizing compilers [1], program analysis has since found wide applications in areas including computer security [16], program verification [18], error detection [7], software engineering, and human-computer interaction.

Despite this history, program analysis today presents a fragmented landscape characterized by point solutions — analyses developed to solve specific problems, with no unifying theory that would enable us to obtain a more comprehensive understanding of the space of program analyses or the relationships between them. Such a theory would help us navigate the space of program analyses to discover previously overlooked analyses, to choose analyses to satisfy specific analysis goals, and to systematically combine analyses within a clean algebraic reasoning framework.

1.1 An Algebraic Framework for Program Analyses

We present a new, unified framework of program analyses. In this framework, each analysis is characterized by a set of program execution traces, with each set of traces corresponding to a program property that the analysis extracts. This property is often made explicit in a *trace predicate* in the form of a logical formula that, given a candidate trace t, returns true if the trace is in the set of traces for the analysis and false otherwise.

Traces provide a unified algebraic framework for comparing and combining both sound and unsound program analyses. Sets of traces ordered by reverse subset inclusion ($\supseteq$) comprise a lattice with least upper bound set intersection ($\cap$) and greatest lower bound set union ($\cup$). This underlying lattice induces a corresponding lattice over program analyses. The lattice order immediately delivers a universal mechanism for comparing analyses, with less precise analyses including more traces than more precise analyses.

Viewing analyses as sets of traces also makes it possible to productively integrate constructs not previously perceived as program analyses into the framework. For example, a set of traces generated by a program fuzzer can be seen as an analysis. Many program analyses assume that certain kinds of errors (for example, out of bounds accesses or accesses via null pointers) do not occur. It is possible to identify these assumptions as sets of traces and integrate them into the framework.

In this paper we primarily consider two kinds of analyses: 1) sound dataflow analyses, which overapproximate the set of program traces and 2) techniques such as fuzzing and concolic execution that explore the set of program traces and which we view as unsound analyses that underapproximate the set of program traces. We illustrate how to specify monolithic dataflow analyses that extract multiple kinds of analyses as the conjunction (least upper bound) of multiple sound microanalyses. We also consider unsound analyses that dijunctively combine traces from multiple fuzzing or concolic executions. These unsound analyses can be seen as generalizing the information from the explored executions.

1.2 Sound and Unsound Analyses

Intuitively, a sound analysis produces an analysis result that holds for all possible executions of the analyzed program. More precisely, an analysis is *sound* for a program P if its set of traces includes all of the traces that the program may exhibit when it executes. Soundness proofs for analyses with sets of traces defined by trace predicates typically involve proving that the trace predicate includes all possible program traces. Prominent examples of sound analyses include standard dataflow analyses and exhaustive testing to enumerate all traces.

Intuitively, an unsound analysis produces an analysis result that may not hold for some possible executions of the analyzed program. Examples of unsound analyses include fuzzing, concolic execution, and bounded model checking. Examples of constructs that can be interpreted as unsound analyses include assertions and execution integrity assumptions such as the absence of memory errors, null pointer dereference errors, or API misuse errors.

We note that the framework cleanly highlights the tension between soundness and precision — soundness requires the analysis to include at least all of the traces of the analyzed program, driving the analysis to include more traces, while precision drives the analysis to include as fewer traces.

The tension between soundness and unsoundness can sometimes be productively addressed by program transformations that systematically transform the program to eliminate undesirable traces not included in the unsound analysis. Examples include failure-oblivious computing and recovery shepherding.

The least upper bound of a sound analysis and an unsound analysis is a *conditionally sound analysis* — an analysis that is sound given the property specified by the unsound analysis. This approach can cleanly and precisely specify, within a unified framework, the assumptions under which an analysis provides a sound characterization of the potential behavior of the program.

1.3 Program Analysis and Secure Input Parsing

Input parsing code is an important source of security vulnerabilities [14]. One common practice is interleaving parsing and validation code with input processing code (shotgun parsing) [6]. A key problem with this common practice is that (potentially adversarial) inputs that trigger program errors or target vulnerabilities should be rejected without processing [6, 17]. But starting input processing before input parsing and validation completes can make it difficult to impossible to reject such inputs before processing — by the time program determines that it should reject the input, it has already been at least partially processed.

One way to attack this problem is to process each input as a transaction, with the input processing effects taking place only when the entire input has been successfully processed [17]. Another approach is to structure the program so that input parsing and validation completes before input processing starts [5]. While domain specific languages for specifying parsers [15] and parser combinator libraries exist that support this coding practice [5], many developers prefer to manually code their input parsers. In any case, it may be beneficial to structure the computation in three

phases as follows. A key aspect of this structure is that if the input parsing or validation phases reject the input, the program behaves as if the input never existed:

- **Input Parsing Phase:** This phase reads the input and populates the data structures that hold the various parts of the input. It rejects inputs that are not syntactically correct.
- **Input Validation Phase:** This phase reads the data structures that hold the parsed data to check that it is possible to fully process the input. It not, this phase rejects the input. It may also store relevant input data into different data structures that the input processing phase reads.
- **Input Processing Phase:** This phase processes the parsed and validated input. It is the only phase that performs any effects that persist beyond the computation that the input triggers.

Each of these three phases is characterized by specific computational patterns that can be verified by appropriate program analyses that target the code for each phase. Analyses for the input parsing phase, for example, may verify that the parsing code writes only data structures that are designed to hold input data. If the input format can be specified by a deterministic context-free grammar or regular grammar, the analysis may verify that the structure of the code corresponds to the structure of the grammar. Analyses for the input validation phase may verify that the code only reads data structures that are designed to hold input data, never reads input data, and writes no data at all. Such analyses may also verify that specific safety checks are applied to specific data structure components. Analyses for the input processing phase may verify that the code never reads additional input and ensures that all data passed to critical functions was appropriately checked in the input validation phase.

The relevant analyses verify a range of properties, from general properties that essentially all applications should exhibit through to application-specific properties determined by the specific function that the application implements. Such analyses are therefore often profitably specified as combinations of a range of more basic analyses, which is one of the specific goals of our framework.

2 FRAMEWORK

We work with programs P that contain labeled commands of the form $l : c \in P$, where $l \in L$, $c \in C$. $\textbf{labels}(P) = \{l \mid \ l : c \in P\}$ is the set of labels in P. Labels are unique — no two labeled commands in P have the same label l. We use the notation $l : c \in P$ to denote that the command at label l in program P is c and the notation $l : c \notin P$ to denote that the command at label l in program P is not c.

2.1 Program Execution Traces

An executing program operates on states $\sigma \in \Sigma$. Execution starts at $\langle l_0, \sigma_0 \rangle$, where $l_0 = \textbf{first}(P)$ is the label of the first command to execute and σ_0 is the initial state. Each execution generates a trace $t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle$ of label/state pairs $\langle l_i, \sigma_i \rangle$. Execution terminates if it encounters an l : **halt** command. We require that terminated traces do not further execute, i.e., if $t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle$ and l_i : **halt** $\in P$, then $i = n$. The set of potential program traces $\mathcal{T}$ is therefore $\mathcal{T} = \{t \in (L \times \Sigma)* \mid l_i : \textbf{halt} \in P \text{ implies } i = n \text{ where } t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle\}$. $\textbf{done}(\langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle)$ is true if l_n : **halt** and false otherwise. $\textbf{traces}(P)$ is the (prefix closed) set of all (partial or done) traces from all executions of P.

2.2 Traces Sets and Trace Predicates

Each program analysis A corresponds to a set of traces. This set is often defined by *trace predicates*, in which case, given a program P, A has corresponding trace predicates τ_A^P that define the set of

traces that the analysis A identifies. Here $\tau_A^P(t)$ is true if t is one of the traces that the analysis A identifies for program P and false otherwise. $\textbf{traces}(A, P) = \textbf{traces}(\tau_A^P) = \left\{t \in \mathcal{T} \mid \tau_A^P(t)\right\}$ is the *traces induced by the analysis of P by A*. We conceptually identify trace predicates with the sets of traces they define when it is convenient:

- $\tau_1 \subseteq \tau_2 \equiv \forall t \in \mathcal{T}, \tau_1(t)$ implies $\tau_2(t)$
- $\tau_1 \sqcup \tau_2 \equiv \forall t \in \mathcal{T}, \tau_1(t)$ and $\tau_2(t)$
- $\tau_1 \sqcap \tau_2 \equiv \forall t \in \mathcal{T}, \tau_1(t)$ or $\tau_2(t)$
- $\tau_1 - \tau_2 \equiv \forall t \in \mathcal{T}, \tau_1(t)$ and (not $\tau_2(t)$).

The subsets of the set of all traces $\mathcal{T}$ form a lattice with order reverse subset inclusion $\supseteq$, least upper bound $\cap$ (noted $\sqcup$), and greatest lower bound $\cup$(noted $\sqcap$)[1]. Conceptually, the fewer traces that a trace predicate τ identifies, the more precise the trace predicate τ. So $\tau(t) = \text{true}$ ($\tau = \mathcal{T}$) is the least precise trace predicate and $\tau(t) = \text{false}$ ($\tau = \emptyset$) is the most precise trace predicate. A trace predicate τ is sound for a program P if $\tau \subseteq \textbf{traces}(P)$.

Trace predicates are often defined using a data structure D_A^P, produced by the analysis A, that contains the information derived when A analyzes P. We omit A and/or P when obvious from context.

We require trace predicates to define prefix-closed sets of traces. They therefore must be monotonic with respect to the operation of extending the trace, i.e. $\tau(\langle l_0, \sigma_0\rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n\rangle \rightarrow \langle l_{n+1}, \sigma_{n+1}\rangle)$ implies $\tau(\langle l_0, \sigma_0\rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n\rangle)$.

Conceptually, program analyses often determine or verify that a program exhibits a desired property. In this case the trace predicates can often be seen as rejecting traces that violate the property (i.e., returning false for such traces). Given a trace and a property, we consider the following alternatives:

- **Violation:** The trace definitely violates the property. In this case the trace predicate should reject the trace. To ensure prefix closure, the trace predicate must also reject all extensions of the trace.
- **No Violation:** The trace definitely does not violate the property. In this case the trace predicate should accept the trace and all extensions of the trace.
- **Potential Violation:** The trace does not violate the property, but some extension of the trace may violate the property. In this case the trace predicate should accept the trace, but may reject some extensions of the trace.

We identify three kinds of trace predicates: *early predicates* that transition from true to false only at nonterminated traces, *late predicates* that check properties that can only be determined once the program has terminated and are therefore false only for terminated traces, and *mixed predicates* that are intersections of early and late predicates.

Conceptually, a safety property [3] states that something bad does not happen. Because trace predicates operate on finite traces and are monotonic, in our framework all trace predicates specify safety properties.

Trace predicates deliver a finite representation of potentially infinite set of finite traces of unbounded length. One goal is to obtain a computationally tractable representation suitable for effective reasoning about the relationships between different analysis represented via trace predicates. We anticipate that the specific form of each trace predicate will depend on the analysis, relevant relationships between different analyses, and the specific reasoning contexts in which the trace predicate will be used.

[1]Note that the lattice is the *reverse* standard lattice, so the least upper bound is an intersection, instead of an union

2.3 Program Analysis Lattice

We use trace predicates to define a program analysis lattice as follows. We define the lattice order $A_1 \sqsubseteq A_2$ if $\mathbf{traces}(A_1, P) \supseteq \mathbf{traces}(A_2, P)$ for all P. The least upper bound $A_1 \sqcup A_2$ is the analysis A with trace predicates $\tau_A^P = \tau_{A_1}^P \cap \tau_{A_2}^P$ and the greatest lower bound $A_1 \sqcap A_2$ is the analysis A with trace predicates $\tau_A^P = \tau_{A_1}^P \cup \tau_{A_2}^P$.

Given two arbitrary program analyses A_1 and A_2, one may wonder how to construct an analysis $A = A_1 \sqcup A_2$ or $A = A_1 \sqcap A_2$. Given a program P, one may run $A_1(P)$ and $A_2(P)$ to obtain data structures $D_{A_1}^P$ and $D_{A_2}^P$. Then $A_1 \sqcup A_2$ is the analysis that produces the information in $D_{A_1}^P$ and $D_{A_2}^P$, while $A_1 \sqcap A_2$ is the analysis that produces the information in $D_{A_1}^P$ or $D_{A_2}^P$. Of course, it may be possible to produce the same information with a different analysis algorithm or represent the same information with a different data structure.

Conceptually, the fewer traces that the trace predicates for an analysis identifies, the more information the analysis provides about the program. So the analysis A with trace predicates $\tau_A^P(t) = \text{true}$ is the least informative analysis and the analysis A with trace predicates $\tau_A^P(t) = \text{false}$ is the most informative analysis. An analysis A is conservative if $\mathbf{traces}(P) \subseteq \mathbf{traces}(\tau_A^P)$ for all P. If an analysis is conservative, the trace predicates τ_A^P hold for all executions of P.

This lattice on program analyses is reminiscent of what can be obtained through the lense of abstract interpretation when comparing different analyses to a trace semantics. However, our construction as two main advantages over that more traditional one.

First, our framework can consider unsound analysis, and compositions of sound and unsound analysis, as we pointed out in subsection 1.2.

Second, many traditional analyses operate on information that is not usually part of the state in standard trace semantics. For example, for a live variables analysis, the set of live variables at each point of a program is not part of the state for traditional trace semantics. While it is possible to enrich the state of the base semantics as done in [9, 13], and so recover the ability to compare different analysis through abstract interpretation, one need to adapt the base trace semantics by hand to the specific analysis at hand, and then build the different abstraction functions. In contrast, we do not require to change the state of the base semantics, we carry the domain specifics information of each analysis completely within the predicate checker.

3 EXAMPLE DATAFLOW ANALYSES

We next present the data structures and corresponding trace predicates for a range of exemplary dataflow analyses. All of these analyses are designed to analyze programs written in a language with a set of variables $v \in V$, a set of expressions $e \in E$, and variable assignment commands of the form $v = e$. We use the notation $l : v = e \in P$ to denote that the variable assignment command $v = e$ appears at label l in program P. When we do not care about a specific component of a command c we write $\_$. For example, $l : v = \_ \notin P$ indicates that the command at label l in P is not an assignment to the variable v. $\mathbf{vars}(c)$ is the set of variables v that the command c reads. The values of the variables are drawn from a set of values N.

Each analysis produces a data structure D that stores, for each label l in the analyzed program P, an analysis result at the program point either before the command at l executes (we denote this program point $\bullet l$) or after the command at l executes (we denote this program point $l\bullet$). The trace predicates for the analyses are expressed using these data structures.

Correct implementations of all of these exemplary analyses use the standard dataflow analysis approach to produce sound analysis results that correctly reflect all possible program executions.

The analyses A are therefore conservative. The fact that the trace predicates τ_A^P identify a superset of the actual program traces is typically proved by induction on the length of the program traces.

3.1 Variable Value Analysis

Variable value analysis VV computes, for each program point, the values of variables that are analysis-time constants at that point. The data structure D_{VV} stores, for each program point $\bullet l$, the set of variables that the analysis has determined have a specific value $n \in N$ (so that $D_{VV}(\bullet l)(v)$ is the analysis-time constant value n that the analysis has computed for the variable $v \in \mathbf{dom}\ D_{VV}(\bullet l)$ at the program point just before the command at label l executes). The trace predicate $\tau_{VV}(t)$ checks that the values that the analysis computes are consistent with the values in the trace:

$$\tau_{VV}(t) := \forall\, \langle l, \sigma\rangle \in t,\ v \in \mathbf{dom}\ D_{VV}(\bullet l) \Rightarrow D_{VV}(\bullet l)(v) = \sigma(v) \tag{1}$$

This trace predicate is an early trace predicate.

3.2 Reaching Definitions Analysis

Reaching definitions analysis computes, for each program point, the set of definitions (i.e., the assignments $l : v = \_ \in P$) that reach that point (i.e., execute before that point in some execution without an intervening definition of the same variable). The data structure $D_{RD}(\bullet l)$ stores the set of reaching definitions at the program point $\bullet l$. The trace predicate checks that a definition reaches every succeeding program point in the trace until there is another definition of the same variable:

$$\begin{aligned}\tau_{RD}(t) := \forall 0 \le i < j \le n, (l_i : v = \_ \in P) \wedge (\forall i < k \le j,\ l_k : v = \_ \notin P)\\ \Rightarrow l_i \in D_{RD}(\bullet l_j)\end{aligned} \tag{2}$$

where $t = \langle l_0, \sigma_0\rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n\rangle$. This trace predicate is an early trace predicate.

3.3 Live Variables Analysis

Live variables analysis computes, for each program point, the set of variables that are live at that point — i.e., the set of variables that are read before they are written in some subsequent execution. The data structure $D_{LV}(l\bullet)$ stores the set of live variables at the program point $l\bullet$. The trace predicate checks that if a variable is not live after a command executes, there must be another definition of the variable before the trace executes another command that reads the variable:

$$\begin{aligned}\tau_{LV}(t) = \forall 0 \le i \le n,\ v \notin D_{LV}(l_i\bullet) \text{ implies}\\ \forall i < j \le n,\ l_j : c \in P \text{ and } v \in \mathbf{vars}(c) \text{ implies } \exists i < k < j,\ l_k : v = \_ \in P\end{aligned} \tag{3}$$

where $t = \langle l_0, \sigma_0\rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n\rangle$. This trace predicate is an early trace predicate.

3.4 Very Busy Expressions Analysis

Very busy expressions analysis computes, for each program point, which expressions are evaluated along all control flow paths from that point before any of the variables in the expression are redefined. The trace predicate is an intersection of an early trace predicate and a late trace predicate. The early trace predicate τ_E checks that if an expression e is very busy after a command executes, the expression is evaluated before any of the variables v in the expression e are redefined:

$$\begin{aligned}\tau_E(t) = \forall 0 \le i \le n, e \in D_{VB}(l_i\bullet) \text{ and } v \in \mathbf{vars}(e) \text{ implies}\\ (\forall i < j \le n, l_j : v = \_ \in P \text{ implies } \exists i < k \le j, l_k : c \in P \text{ and } e \in \mathbf{subs}(c))\end{aligned} \tag{4}$$

where $t = \langle l_0, \sigma_0\rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n\rangle$.

The late predicate τ_L checks that if an expression e is very busy after a command executes, the expression is evaluated before the execution terminates. A late predicate is required here because it

is, in general, not possible to tell that a nonterminated trace will not evaluate an expression before it terminates — it is always possible to extend any nonterminated trace that does not evaluate the expression with a command that does evaluate the expression. We use the notation $\mathbf{subs}(e)$ to denote the set of subexpressions evaluated during the evaluation of e.

$$\tau_L(t) = \forall 0 \leq i \leq n, e \in D_{VB}(l_i\bullet), l_n : \mathbf{halt} \in P \text{ implies } \exists i < k \leq n, l_k : c \in P \text{ and } e \in \mathbf{subs}(c) \quad (5)$$

where $t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle$.

A late predicate τ_L specifies that all very busy expressions must be evaluated before the program halts. A corresponding property for infinite executions would require that all very busy expressions would be eventually be evaluated (at some indeterminate time in the future). We note that this corresponding property is a liveness property, in the sense of [3], and is therefore not expressible as a trace predicate in our framework.

We make several observations here. First, dataflow analyses typically reason about all paths in the control flow graph of the program, whether those paths are actually realizable or not. The standard definition of very busy expressions would therefore determine that an expression that is evaluated on a control flow path out of an infinite loop is still very busy before the loop (assuming none of the intervening statements write one of the variables in the expression), even though no execution of the program actually exits the loop to evaluate the expression. The trace predicates that we present here accurately express this property.

It is possible to envision a very busy expressions analysis that determines that an expression is very busy only if every execution, including nonterminating executions, evaluates the expression. For an analysis to make this determination in the presence of potentially infinite executions, it would have to reason about termination. We anticipate that this termination reasoning would be reflected in a trace predicate that rejects nonterminating traces. Intersecting this trace predicate with the very busy expressions trace predicate above would precisely express the property that every execution evaluates the expression (because there are no nonterminating executions).

3.5 Discussion

It is instructive to consider the properties that the trace predicates (and their corresponding analyses) identify. Because the data structures D that the analysis produces are largely decoupled from the control flow of the analyzed program P, the trace predicates include many traces with command execution sequences that P will never execute and are even completely incompatible with the control flow graph of P. The variable value analysis, for example, allows all traces whose states conform to the specified values from the analysis, whether or not these traces are even remotely related to any causality in the program execution. Consider, for example, the following program:

```
1: x = 4;
2: y = 5;
3: z = x + y;
4: halt;
```

The variable value analysis for this program produces the following data structure, which records the analyzed values of each variable before each command in the program:

```
{ 1: [], 2: [x->4], 3: [x->4, y->5], 4: [x->4, y->5, z->9] }
```

The trace predicate simply requires that, if the data structure records a value for a variable before a command, the variable has that value in the corresponding state. Here are some examples of traces that satisfy the trace predicate:

```
<1, [x->0, y->0, z-> 0]> -> <4, [x->4, y->5, z->9]> -> <4, [x->4, y->5, z->9]>
<4, [x->4, y->5, z->9]> -> <3, [x->4, y->5, z->7]> -> <2, [x->4, y-> 5, z->9]>
```

The first trace simply skips commands 2 and 3 and has two occurrences of command 4. The second trace corresponds to executing the program backwards and includes values for z that occur in no execution of the program. The trace predicates for the other dataflow analyses exhibit similar properties.

This aspect of our framework is deliberate. It highlights the benefits of a clean separation between the analysis algorithm and the information that it produces. It directly reflects the fact that many analyses overapproximate the traces they identify. This overidentification can be caused by abstractions designed to enable efficient analysis, by analysis-time uncertainty about the flow of control, or simply because the scope of the information that the analysis is designed to produce.

We note that even though a dataflow analysis algorithm may analyze the control flow graph of the program to produce its results, the information that it produces typically identifies a much larger set of traces (so not just traces compatible with the control flow graph of the program). This can be seen, for example, in the example traces presented above. This phenomenon can be seen as emphasizing the separation between the analysis algorithm (the way the analysis produces its results) and the actual information that the algorithm produces.

Here the algebraic structure of the program analysis lattice can guide the refinement of the analysis results. One typical pattern intersects one or more dataflow analyses with a control flow analysis (Section 4) to obtain, for example a set of traces that is consistent with the control flow graph and conforms to the information produced by the dataflow analysis.

4 CONTROL FLOW ANALYSES

Control flow analyses extract information about the flow of control through the program. They typically construct, then analyze, a control flow graph of the program.

4.1 Control Flow Graph Analysis

Control flow graph analysis extracts the control flow graph of the program. Here we consider an analysis that operates at the level of the individual commands in the program. The analysis produces a data structure D_{CG} in which $D_{CG}(l)$ stores the labels of the successors of the command at label l in the control flow graph. The trace predicate checks that all executions are consistent with this successor information.

$$\tau_{CG}(t) = \forall 0 \leq i < n,\ l_{i+1} \in D_{CG}(l_i) \tag{6}$$

where $t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle$.

4.2 Post-Dominator Analysis

A node l_2 in the control flow graph post-dominates another node l_1, before exiting in l_3, if all paths that goes through l_3 is such that, if it went through l_1 before going through l_3, then it must have gone through l_2 in between l_1 and l_3.

Similarly to the dominator analysis, the post-dominator analysis produces a data structure D_{PDA} in which $D_{PDA}(l)$ stores the labels of the commands that post-dominate l with respect to a constant exit l_{exit}. The trace predicate checks that the trace is consistent with the post-dominator information, similar to the dominator trace predicate check.

$$\tau_{PDA}(t) = \forall 0 \leq i \leq e \leq n, l_e = l_{\text{exit}}, \forall l \in D_{PDA}(l_i),\ \exists i \leq k < e,\ l_k = l \tag{7}$$

where $t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle$.

4.3 Unrealizable Transition Analysis

This analysis identifies variable values that affect the flow of control. It produces a data structure D_{UT} in which D_{UT} stores tuples $\langle l, v, n, l' \rangle$ where $l' \in \mathbf{succ}(l)$ is a label, v is a variable, and n is a value of the variable. Here the tuple indicates that if $v = n$ at l, then control does not flow from l to l' — any transition of the form $\langle l, \sigma \rangle \rightarrow \langle l', \sigma' \rangle$ is unrealizable and does not occur in any execution of the program P under analysis, for example because $l : c \in P$ is a conditional statement that takes a different path when $v = n$. The trace predicate checks that the trace is consistent with this information:

$$\tau_{UT}(t) = \forall 0 \leq i < n,\ (\exists v,\ \langle l_i, v, \sigma_i(v), l \rangle \in D_{UT}) \text{ implies } l_{i+1} \neq l \tag{8}$$

where $t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle$.

4.4 Loop Exit Analysis

This analysis identifies loops, induction variables, and loop exit conditions. It produces a data structure D_{LE} which stores tuples $\langle l, v, n, l' \rangle$ where $l : c \in P$ is the command that implements the loop exit condition, $l' \in \mathbf{succ}(l)$ is the label of the next command that executes when the loop exits, and v is an induction variable that is always at least 0 and increases on every loop iteration, with the loop exiting when $n \leq v$. We express the trace predicate as the intersection of a loop initialization predicate τ_{init}, a loop exit predicate τ_{exit} and loop progress predicate τ_{prog} that requires the induction variable v to increase monotonically with each loop iteration, with $l : c \in P$ executing between executions of the loop.

$$\tau_{init}(t) = \forall 0 \leq i \leq n,\ \langle l_i, v, n, l \rangle \in D_{LE} \text{ implies } 0 \leq \sigma_i(v) \tag{9}$$

$$\tau_{exit}(t) = \forall 0 \leq i < n,\ \langle l_i, v, n, l \rangle \in D_{LE} \text{ and } n \leq \sigma_i(v) \text{ implies } l_{i+1} = l \tag{10}$$

$$\tau_{prog}(t) = \forall 0 \leq i < j \leq n,\ l_i == l_j \text{ and } \langle l_i, v, n, l \rangle \in D_{LE} \text{ implies } \sigma_i(v) < \sigma_j(v) \text{ or } \exists i < k < j, l_k = l \tag{11}$$

where $t = \langle l_0, \sigma_0 \rangle \rightarrow \cdots \rightarrow \langle l_n, \sigma_n \rangle$.

The resulting loop exit analysis predicate is $\tau_{LE} = \tau_{init} \sqcup \tau_{exit} \sqcup \tau_{prog}$. Given a tuple $\langle l, v, n, l' \rangle$, this trace predicate rejects all traces in which the loop at l executes more than n iterations, i.e., in which the label l occurs more than n times in the trace without an intervening occurrence of the loop exit label l'. For this predicate to be sound, the analysis must prove that all executions of the loop execute at most n times in all executions of the program P under analysis.

5 ANALYSIS FACTORING AND MICROANALYSES

The framework we present in this paper promotes factored analyses — analyses specified as the intersection/least upper bound of smaller building block microanalyses. We next outline two factored analyses.

5.1 Commands That Never Execute

This analysis reasons about the values of variables to discover commands that never execute. Such analyses are often expressed as an integrated monolith that combines the different sources of information into a single whole. We instead advocate specifying the analysis as $A_{VV} \sqcup A_{UT} \sqcup A_{CG}$.

Conceptually, the analysis first computes the values of variables with analysis-time constant values (A_{VV}). It then removes any traces with unrealizable transitions (A_{UT}). This operation still leaves traces that simply skip the unrealizable transition but contain unrealizable commands that never execute, for example because they are part of the control flow path that the unrealizable

transition enters. The last step applies the control flow graph analysis to remove any traces that violate the transitions in the control flow graph. This operation eliminates traces with unrealizable commands — these traces are not compatible with the control flow analysis because the transition required to enter the path they are on is missing.

This example illustrates how the framework promotes the development of a toolbox of general microanalyses that can be flexibly composed into larger analyses. We anticipate, for example, that the control flow graph analysis will prove to be a useful building block microanalysis because it can effectively complement analyses (such as the unrealizable transition analysis) that prune single edges, with the control flow graph analysis extending the pruning to complete subpaths within the control flow graph.

5.2 Very Busy Expressions That Always Evaluate

The standard very busy expressions analysis can identify that an expression is very busy even if (because of an infinite loop), no execution of the program actually evaluates the expression. Here we propose combining the loop exit analysis, control flow analysis, and standard very busy expressions analysis to obtain an analysis that identifies an expression as very busy only if it is evaluated in all executions. The analysis is specified as $A_{LE} \sqcup A_{VB} \sqcup A_{CG}$, where A_{LE} works with a data structure D_{LE} that contains a tuple $\langle l, v, n, l' \rangle$ for every loop in the program.

The analysis A_{LE} therefore ensures that all executions of the analyzed program P terminate. If all executions terminate, then A_{VB} ensures that all very busy expressions are evaluated in all executions if the executions conform to the control flow graph of P. And A_{CG} ensures that all traces conform to the control flow graph.

6 FUZZING

Fuzzing as initially conceived tests programs by generating random inputs. Fuzzing has since evolved to include a variety of techniques designed to enhance its effectiveness and more efficiently target various potential error locations. Fuzzing has proved to be surprisingly effective at surfacing errors in programs and has since become one of the most widely deployed testing and security vulnerability detection techniques.

We model fuzzing in our framework by simply disjunctively collecting the set of traces exercised during fuzzing. We note that this set grows as the fuzzer generates inputs, becoming a sound analysis in the limit as it approaches generating all possible inputs. This basic perspective also holds for concolic execution, which we model as simply the traces that the concolic execution generates. It is possible to accelerate the approach to soundness by generalizing information from the multiple traces by imposing additional structure in the form of an inferred logical formula that all of the traces satisfy.

These techniques deliver an unsound analysis that can be seen as characterizing a slice of the program's potential executions. A comparison of the trace sets from a sound analysis with the trace sets from the unsound analysis can provide an upper and lower bound on the actual sets of traces that the program can actually exhibit. These bounds can, in turn, deliver insight into how accurately the sound and unsound analyses characterize the set of program traces.

7 ALTERNATIVE TRACE PREDICATE FORMULATION

We next present an alternative formulation of trace predicates in which the trace predicates are formalized as algorithms (here written in pseudo-Haskell) that take a trace as an argument, then return true or false. This operational view of trace predicates can be seen as conceptually similar to the formulation of liveness and safety properties via Büchi automata [2].

7.1 Reaching Definitions Analysis

```
updateReaching label command reachingDefinitions =
    case command of
        Set v _expr -> DataMap.insert v (DataSet.singleton label) reachingDefinitions
        _ -> reachingDefinitions

reachingDefVar analysisResult v =
    fromMaybe DataSet.empty (DataMap.lookup v analysisResult)

traceChecker :: DataMap.Map String (DataSet.Set Label) ->
    DataMap.Map String (DataSet.Set Label) ->
    Trace a -> Bool
traceChecker analysisResult reachingDefinitions (((label,command),_currentState):t) =
    let validCurrentState = DataMap.foldrWithKey (\key setReaching acc -> acc &&
            DataSet.isSubsetOf
                setReaching
                (reachingDefVar (analysisResult label) key)) True reachingDefinitions in
    let newReachingDefinitions = updateReaching label command reachingDefinitions in
        validCurrentState && traceChecker analysisResult newReachingDefinitions t
traceChecker analysisResult reachingDefinitions [] =
    True
```

7.2 Live Variables Analysis

```
updateLive :: p -> Command -> DataSet.Set String -> DataSet.Set String
updateLive label command liveVariables =
    case command of
        Set v _expr -> DataSet.delete v . DataSet.union liveVariables $ footprint _expr
        _ -> DataSet.union liveVariables $ footprintCommand command

traceCheckerReverse :: DataSet.Set String -> DataSet.Set String -> Trace a -> Bool
traceCheckerReverse analysisResult liveVariables (((label,command),_currentState):t) =
    let validCurrentState = DataSet.isSubsetOf liveVariables (analysisResult label) in
    let newLiveDefinitions = updateLive label command liveVariables in
        validCurrentState && traceCheckerReverse analysisResult newLiveDefinitions t
traceCheckerReverse analysisResult liveVariables [] =
    True

traceChecker analysisResult liveVariables =
     traceCheckerReverse analysisResult liveVariables . reverse
```

8 CONCLUSION

We propose a unified framework of program analyses based on sets of traces. This approach makes it possible to compare the precision of different analyses and to algebraically combine analyses both conjunctively and disjunctively. One advantage include a uniform framework for all analyses, including both sound and unsound analyses, as well as things that have not been considered analyses at all. Another advantage is the ability to combine analyses in an algebraic framework

and factor complex analyses into smaller microanalyses. Finally, we anticipate that this framework will help practitioners understand the potential space of program analyses at their disposal and match these analyses to the needs of their application.

8.1 Taint Tracking Analysis

We propose an analysis to guarantee that the value set in a variable at a program point cannot influence the value of other variables at various program point.

The analysis produces a data structure D_{IFC} which stores for each program point l, for each variable v, the set of variables guaranteed to not have influenced the value in v.

```
type dIfc = DataMap.Map Label (DataMap.Map String DataSet.Set String) -- Result of the analysis

data dep = Dependencies{ -- dynamic data structure that accumulate the actual dependencies
     variables :: DataMap.Map String (DataSet.Set String),
     -- for each variable , describe what is the current reaching definition , and the variables that tainted th
     pc :: DataSet.Set String
     -- variables that taint the current program point
     }

updateDep :: p -> Command -> dep -> dep
-- Function that update dependencies with standard taint tracking style

traceChecker :: dIfc -> dep -> Trace a -> Bool
traceChecker analysisResult dependencies ((( label ,command),_currentState):t) =
    let validCurrentState =
            DataMap.foldrWithKey (\var untaintedVars acc -> acc &&
                    DataSet. disjoint
                    -- the variables taints at this point of the trace are agreeing
                    -- with the prescription of the analysis
                    -- i.e. no guaranteed untainted variable is tainted
                        untaintedVars
                        (DataSet.union
                            (pc dependencies)
                            (fromMaybe DataSet.empty
                                (lookup var ( variables dependencies )))))
                True
                (fromMaybe DataMap.empty (lookup label analysisResult)) in
    let newdep = updateDep label command dependencies in
        validCurrentState && traceChecker analysisResult newdep t
traceChecker analysisResult dependencies [] =
    True
```

9 RELATED WORK

There has been an enormous amount of research in dataflow analyses [11], later further formalized in the framework of abstract interpretation [8]. Throughout the history of the field, the focus has been on individual analyses of increasing sophisticationa and complexity, with scalability a ever-present concern.

While researchers and practitioners have encountered the need to combine analyses to realize their information extraction goals, this combination has largely been performed in an ad-hoc way driven by the specific needs of the program analyses problem at hand. That is, the focus was on how to compose various program analyses to solve a specific problem rather than deriving a general algebraic framework for classifying, decomposing, and characterizing relationships between multiple analyses, including both sound and unsound analysis.

Consider for example: [4] presents Astree's take on composing abstract domains. Astree is a well-known framework to analyse the absence of undefined behavior in C programs. It was originally intended to analyse programs solving control tasks from the embedded world, generated from a program written in a synchronous language. Astree supports a wide family of parametrizable abstract domains, most of them capturing relations of various precisions between program numeric variables. This extensibility allows Astree to raise very few false alarms on the test programs that consists of an order of hundred thousand lines of C. [10] elaborate on the same tool, also pointing out that Astree does not actually require its abstract domains to form a Galois connection. Instead Astree only requires concretization functions. Abstract domains are used as a way to pack trace fragments together in a compact way, equipped with an abstract step function.

In [12], the authors propose a mechanism to reduce the cost of computing the result of several entangled dataflow analyses without the loss of precision potentially incurred when computing the results of the different dataflow analyses independently. The key idea is to allow the different entangled analyses to iteratively replace and refine the program they are analyzing. In this methodology, the refined program is a convenient way for an analysis to inform the other analyses of the information it extracted. For example a constant propagation analysis would not only determine which variables are constant at specific program points, but also transform the program to make the fact that the value is constant visible to other analyses, for example by removing branches on constants.

10 ACKNOWLEDGEMENTS

This resarch was supported by the DARPA AMP program under contract N6600120C4025. The presented concepts and framework were inspired by conversations with Dr. Sergey Bratus.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Bowen Alpern, Bowen Alpera, Fred B. Schneider, and Fred B. Schneider. 1986. Recognizing Safety and Liveness. *Distributed Computing* 2 (1986), 117–126.

[3] Bowen Alpern and Fred B. Schneider. 1985. Defining liveness. *Inform. Process. Lett.* 21, 4 (1985), 181–185. https://doi.org/10.1016/0020-0190(85)90056-0

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. *SIGPLAN Not.* 38, 5 (May 2003), 196–207. https://doi.org/10.1145/780822.781153

[5] Sergey Bratus, J. Adam Crain, Sven M. Hallberg, Daniel P. Hirsch, Meredith L. Patterson, Maxwell Koo, and Sean W. Smith. 2016. Implementing a vertically hardened DNP3 control stack for power applications. In *Proceedings of the 2nd Annual Industrial Control System Security Workshop, ICSS 2016, Los Angeles, CA, USA, December 6, 2016.* ACM, 45–53.

[6] Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon Momot, Meredith L. Patterson, and Anna Shubina. 2017. Curing the Vulnerable Parser: Design Patterns for Secure Input Handling. *login Usenix Mag.* 42, 1 (2017).

[7] Ravi Chugh, Jan W Voung, Ranjit Jhala, and Sorin Lerner. 2008. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 316–326.

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on*

Principles of Programming Languages, Los Angeles, California, USA, January 1977, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252.

[9] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547.

[10] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and David Monniaux. [n.d.]. Combination of abstractions in the Astrée static analyzer. In *In Proc. of the 11th Annual Asian Computing Science Conference (ASIAN'06*. Springer, 272–300.

[11] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*. ACM Press, 194–206.

[12] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing Dataflow Analyses and Transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 270–282. https://doi.org/10.1145/503272.503298

[13] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. 2005. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. ACM, 364–377.

[14] Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*. IEEE Computer Society, 45–52.

[15] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 425–436.

[16] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society, 186–199. https://doi.org/10.1109/CSF.2010.20

[17] Jiasi Shen and Martin Rinard. 2017. Robust programs with filtered iterators. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Benoît Combemale, Marjan Mernik, and Bernhard Rumpe (Eds.). ACM, 244–255.

[18] Karen Zee, Viktor Kuncak, and Martin C. Rinard. 2008. Full functional verification of linked data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. ACM, 349–361.