Verifying the Safety of User Pointer Dereferences

Suhabe Bugrara
Department of Computer Science
Stanford University
Stanford, CA 94305
suhabe@stanford.edu

Alex Aiken
Department of Computer Science
Stanford University
Stanford, CA 94305
aiken@stanford.edu

Abstract

Operating systems divide virtual memory addresses into kernel space *and* user space. *The interface of a modern operating system consists of a set of system call procedures that may take pointer arguments called* user pointers. *It is* safe *to dereference a user pointer if and only if it points into user space. If the operating system dereferences a user pointer that does not point into user space, then a malicious user application could gain control of the operating system, reveal sensitive data from kernel space, or crash the machine. Because the operating system cannot trust user processes, the operating system must check that the user pointer points to user space before dereferencing it. In this paper, we present a scalable and precise static analysis capable of verifying the absence of unchecked user pointer dereferences. We evaluate an implementation of our analysis on the entire Linux operating system with over 6.2 million lines of code with false alarms reported on only 0.05% of dereference sites.*

1 Introduction

Operating systems divide virtual memory addresses into *kernel space* and *user space*. The interface of a modern operating system consists of a set of system call procedures that may take pointer arguments called *user* pointers. It is *safe* to dereference a user pointer if and only if it points into user space. If the operating system dereferences a user pointer that does not point into user space, then a malicious user application could gain control of the operating system, reveal sensitive data from kernel space, or crash the machine [12]. Because the operating system cannot trust user processes, the operating system must check that the user pointer points to user space before dereferencing it.

```
1:  void syscall(int** u) {
2:        int aok; int* cmd;
3:
4:        aok := access_ok(u);
5:
6:        if (aok != 0)
7:              cmd := get(u);
8:        else
9:              cmd := 0;
10: }

11: int* get(int** y) {
12:       int* x;
13:       x := *y;
14:       return x;
15: }
```

Figure 1. Example 1

Figure 1 gives an example of how a user pointer is checked before being dereferenced. The example consists of two procedures: `syscall` and `get`. The procedure `access_ok`, whose definition is not provided in the figure, returns a non-zero value if and only if its pointer parameter points into user space. Procedure `syscall` is a system call available to user applications. Consequently, its pointer parameter `u` is a user pointer. Line 4 applies `access_ok` to `u` to check whether `u` points into user space. Subsequently, line 7 calls `get` with `u` under the condition that the return value of the call to `access_ok` is non-zero which implies that the check on line 4 succeeded. On line 13, procedure `get` dereferences its pointer parameter.

In this paper, we present a scalable and precise static analysis capable of automatically verifying the absence of unchecked user pointer dereferences. That is, the analysis provides a formal guarantee that these security vulnerabilities do not exist given some standard assumptions, which are discussed later. The unchecked user pointer dereferences property is an example of a *finite state safety property*, an important class of specifications extensively studied in previous work [2, 4, 6–8, 10–13, 15]. Intuitively, a finite-state property associates one of a finite set of states with a value at each point in a program. In particular, the finite-state characterization of the unchecked user pointer dereferences property uses the set of states {`user, unchecked, unsafe`} described in Section 5.

We have implemented our analysis and evaluated its precision and scalability on the entire Linux 2.6.17.1 operating system. Our implementation reports false alarms on only 0.05% of dereference sites and uses only 2 annotations for scalability. We believe our static analysis is the first automatic verifier to demonstrate this level of scalability while maintaining soundness and precision.

The key to scalability in our system is the compositional manner in which a program is analyzed. Each procedure P is analyzed independently and only information about P's *summary*, which captures P's behavior with respect to the finite-state property, is communicated to other procedures that call P. Analyzing a procedure independently makes it possible to scale the analysis to millions of lines of code and still use expensive techniques to maintain necessary precision. In particular, our analysis is context-sensitive, flow-sensitive, field-sensitive, and intraprocedurally path-sensitive. Our experience suggests that this level of precision is needed to keep the number of false alarms low. Procedure summaries abstract the intraprocedural analysis but cause relatively little loss of information in practice, as programmers already naturally abstract at procedure boundaries. Our combination of precision and scalability makes near verification of interesting safety properties feasible on large, complex systems.

Our soundness claim makes several standard assumptions about the program being analyzed. In particular, our analysis is not guaranteed to find errors that may arise from unsafe memory operations in C such as buffer overruns, concurrency, and inline assembly. In addition, our system fails to converge completely on a few procedures, which means that any errors that depend on those procedures may not be reported. In our experiment, our system failed to converge completely on 0.17% of all procedures in Linux as described in Section 6. Furthermore, our system builds upon an existing alias analysis [9] that fails to converge completely on 10% of all procedures. However, manual inspection of the alias analysis results suggests that the alias analysis nevertheless conservatively overapproximates the set of concrete aliases for every procedure. Aside from these caveats, our system fully verifies the absence of unchecked user pointer dereferences in the Linux kernel.

1.1 Contributions

This paper makes the following contributions.

- We verify the absence of unchecked user pointer dereferences given some standard assumptions.
- We present experimental results that evaluate the scalability and precision of our analysis on a large, complex, and important program.
- We detail a general framework for summary-based, path-sensitive static analyses of finite-state safety properties.

The rest of this paper is organized as follows. Section 2 covers related work. Section 3 defines the language used in our examples and algorithms. Section 4 briefly explains the *memory model* that handles pointer aliasing. Section 5 details the *safety analysis* for finite-state properties. Finally, Section 6 presents our experience and experimental results.

2 Related Work

We briefly describe closely related work on scalable verification of finite-state properties in general and the unchecked user pointer dereferences property in particular.

MECA [16] is a path-insensitive bug finding tool that statically finds operating system security vulnerabilities such as unchecked user pointer dereferences. MECA is not sound, which means that it does not provide a formal guarantee that vulnerabilities of this kind do not exist. In particular, it does not handle aliasing conservatively and breaks recursive call chains arbitrarily. Using about 45 manual annotations to suppress false alarms, MECA found 44 unchecked user pointer dereferences and reported 8 false alarms on Linux 2.5.63.

Sparse [14] is a bug finding tool developed by the Linux community that statically finds unchecked user pointer dereferences. Linux 2.6.17.1 has more than 9,000 annotations on variables and other constructs believed to be user pointers. Sparse is not sound and does not check the correctness of these thousands of programmer-supplied annotations.

CQual [8] is a type-qualifier inference system that has been used to verify the absence of unchecked user pointer dereferences of individual modules in earlier versions of Linux [12]. Solving the monolithic set of constraints that CQual produces degenerates into a whole program analysis that requires keeping the entire program in main memory, thus limiting its scalability to about 300,000 lines of code. The experiments in [12] are not directly comparable to ours because their reports are clustered to avoid redundancy. For example, the single source causing 111 sink warnings in our system would count as one report by their methodology. Even so, their analysis of Linux 2.2.23 produced 227 raw warnings, or one warning for every 1300 lines of code. We achieve an order of magnitude improvement in warning density due to the fact that our system is flow-sensitive and intraprocedurally path-sensitive which are features that CQual lacks. Several sources of false alarms listed in [12] do not appear in our experiments, because our analysis is precise enough to handle them soundly. Interestingly, function pointers are not a major cause of false alarms in [12], although they cause over 1/3 of the false alarms in our experiments. This is actually consistent with our experience that accuracy in analyzing function pointers is not particularly important until one analyzes the entire operating system including all device drivers.

ESP [4] is a path-sensitive dataflow analysis system used to verify *probing*, the Windows version of unchecked user pointer dereferences [6]. Their experiment reports warnings on 30 of the 500 user pointer sources (6%) on a program consisting of about 1 million lines of code, all of which were false alarms. Contrastingly, our experiment reports warnings on 11 of the 627 user pointer sources (1.8%) on a program consisting of about 6.2 million lines of code. Because ESP's *value-flow* analysis is path-sensitive (including interprocedurally) and performs *strong updates* (a feature we have not discussed, but which our underlying memory model also handles), ESP's expressive power is more comparable to our's than CQual's. One significant technical difference is that ESP encodes path sensitivity by tracking sets of dataflow facts, one set for each path. In contrast, we associate a boolean constraint, or *guard*, with each fact describing a set of paths. In our representation it is easy to reason about multiple paths simultaneously, as the guards directly encode all paths where a fact holds.

SLAM [2] and BLAST [11] are software model checkers based on predicate abstraction. Both systems are able to analyze systems with hundreds of thousands of lines of code. Astree [3] is a static analysis tool that has been used to verify automatically-generated safety-critical software in a restricted subset of C. A notable aspect of Astree is that it performs full verification of the absence of any undefined runtime behavior without making any assumptions about the analyzed program.

3 Language

We briefly define a simple imperative language used to present our analysis in this paper. This language is restrictive, which enables us to present our techniques with minimal extraneous detail. However, our implementation handles the entire C language. A program is a set of *procedures*.

$$
\begin{aligned}
\textit{procedure} &::= \textit{type}\ \texttt{P}\ (\overrightarrow{\textit{type}\ \texttt{v}})\ \{\textit{statement}\}\\
\textit{statement} &::= \textit{type}\ \texttt{v} \mid \texttt{return v} \mid \texttt{v := } \mathbb{Z}\\
&\quad \mid \texttt{v}_1\texttt{ := *v}_2 \mid \texttt{v}_1\texttt{ := \&v}_2\\
&\quad \mid \texttt{v}_1\texttt{ := Q }(\overrightarrow{\texttt{v}_2})\\
&\quad \mid \textit{statement}\ \texttt{;}\ \textit{statement}\\
&\quad \mid \texttt{if (v != 0)}\ \textit{statement}\\
&\quad\ \ \ \texttt{else}\ \textit{statement}\\
\textit{type} &::= \texttt{void} \mid \texttt{int} \mid \texttt{int*} \mid \texttt{int**}
\end{aligned}
$$

A procedure has a return type, formal parameters, and a statement. The statement forms are self explanatory. Our analysis always takes place in some procedure, thus we superscript variables, sets, and functions with the name of the procedure with which they are associated. The superscript is omitted when the procedure is clear from context. Let $Proc$ be the set of procedures in the program. Then, $Rvalue^P$ is the set of right hand side expressions, $Lvalue^P$ is the set of left hand side expressions, and Var^P is the set of variables $\texttt{v}$ of procedure $P \in Proc$. Figure 1 shows the running example used in the rest of the paper written in this language.

4 Memory Model

This section describes the *memory model* used by the safety analysis. The memory model of a procedure gives an abstract description of the portion of the heap relevant to the procedure. Interested readers are referred to [9] for details of how the memory model is computed.

The memory model for a procedure P consists of a distinct set of *abstract locations* Loc^P (usually called just *locations*). A function $varloc^P \in Var^P \rightarrow Loc^P$ assigns each variable $\texttt{v} \in Var^P$ to a location $l \in Loc^P$. The set of *guards*, $Guard^P$, consists of propositional formulas over a set of primitive predicates on locations and boolean variables.

EXAMPLE. In procedure `syscall` in Figure 1, suppose the memory model assigns the location $l_{\mathtt{aok}}$ the variable `aok` so $\mathit{varloc}^{\mathtt{syscall}}(\mathtt{aok}) = l_{\mathtt{aok}}$. Then, the conditional `aok != 0` that appears on line 6 is represented by the guard $(l_{\mathtt{aok}} \neq 0) \in \mathit{Guard}^{\mathtt{syscall}}$. □

4.1 Guarded Points-to Graphs

A *guarded points-to graph* $\rho \in \mathit{PointsTo}^P = \mathit{Loc}^P \times \mathit{Loc}^P \to \mathit{Guard}^P$ is a function that associates a guard $\varphi \in \mathit{Guard}^P$ with each pair of locations $l_i, l_j \in \mathit{Loc}^P$, representing the condition under which l_i may point-to l_j. The *rvalue evaluation* function $\mathit{rval}^P \in \mathit{PointsTo}^P \to (\mathit{Rvalue}^P \times \mathit{Loc}^P) \to \mathit{Guard}^P$ gives the guard under which an *rvalue* may point-to a given location. An *rvalue* is an expression that appears on the right-hand side of an assignment. Similarly, the *lvalue evaluation* function $\mathit{lval}^P \in \mathit{PointsTo}^P \to (\mathit{Lvalue}^P \times \mathit{Loc}^P) \to \mathit{Guard}^P$ gives the guard under which an *lvalue* is *represented* by a given location. An *lvalue* is an expression that appears on the left-hand side of an assignment.

EXAMPLE. In procedure `get` in Figure 1, assume the guarded points-to graph $\rho_{\mathtt{11}}$ encodes the points-to relationships at entry on line 11. Also let $l, l_{\mathtt{y}}, l_{\mathtt{*y}}, l_{\mathtt{**y}} \in \mathit{Loc}^P$ where $\mathit{varloc}(\mathtt{y}) = l_{\mathtt{y}}$ and $\mathit{varloc}(\mathtt{x}) = l_{\mathtt{x}}$. The guard under which the lvalue `*y` is represented by the location $l_{\mathtt{*y}}$ with respect to the points-to graph $\rho_{\mathtt{11}}$ is $\mathit{lval}(\rho_{\mathtt{11}})(\mathtt{*y}, l_{\mathtt{*y}})$, which is equivalent to $\mathit{rval}(\rho_{\mathtt{11}})(\mathtt{y}, l_{\mathtt{*y}})$, the guard under which the rvalue `y` points-to $l_{\mathtt{*y}}$ with respect to $\rho_{\mathtt{11}}$, which is equivalent to $\rho_{\mathtt{11}}(l_{\mathtt{y}}, l_{\mathtt{*y}})$. Similarly, the guard under which the lvalue `**y` is represented by the location $l_{\mathtt{**y}}$ is $\mathit{lval}(\rho_{\mathtt{11}})(\mathtt{**y}, l_{\mathtt{**y}})$, which is equivalent to $\mathit{rval}(\rho_{\mathtt{11}})(\mathtt{*y}, l_{\mathtt{**y}})$, the guard under which the rvalue `*y` points-to $l_{\mathtt{**y}}$, which is equivalent to $\bigvee_l [\rho_{\mathtt{11}}(l_{\mathtt{y}}, l) \wedge \rho_{\mathtt{11}}(l, l_{\mathtt{**y}})]$, which is the disjunction of all guards under which $l_{\mathtt{y}}$ points-to some location l and l points-to $l_{\mathtt{**y}}$. □

4.2 Location Instantiation

The abstract locations of two different procedures are disjoint: $\mathit{Loc}^P \cap \mathit{Loc}^Q = \emptyset$ if $P \neq Q$. A separate mapping shows when two abstract locations from different procedures represent the same set of concrete locations. Suppose P calls Q at a call statement `s` in P and let $\rho \in \mathit{PointsTo}^P$ be the points-to graph encoding the points-to information at `s`. The *location instantiation* function $\mathcal{I}^P_{\mathit{Loc}} \in \mathit{PointsTo}^P \to (\mathit{Loc}^Q \times \mathit{Loc}^P) \to \mathit{Guard}^P$ gives the guard under which the abstract locations $l_1^Q \in \mathit{Loc}^Q$ and $l_2^P \in \mathit{Loc}^P$ represent the same set of concrete locations. We say the callee location l_1^Q *instantiates* to the caller location l_2^P at `s` if they represent the same set of concrete locations.

EXAMPLE. In the example in Figure 1, procedure `syscall` calls procedure `get` on line 7 with actual parameter `u` and formal parameter `y`. Let $\rho_{\mathtt{11}} \in \mathit{PointsTo}^{\mathtt{get}}$ be the points-to graph at entry to procedure `get` on line 11 and $\rho_{\mathtt{7}} \in \mathit{PointsTo}^{\mathtt{syscall}}$ be the points-to graph at the call statement on line 7. Also let $l_{\mathtt{u}}, l_{\mathtt{*u}} \in \mathit{Loc}^{\mathtt{syscall}}$ and $l_{\mathtt{y}}, l_{\mathtt{*y}} \in \mathit{Loc}^{\mathtt{get}}$, where $\mathit{varloc}^{\mathtt{syscall}}(\mathtt{u}) = l_{\mathtt{u}}$ and $\mathit{varloc}^{\mathtt{get}}(\mathtt{y}) = l_{\mathtt{y}}$. The locations $l_{\mathtt{u}}$ and $l_{\mathtt{y}}$ never correspond to the same set of concrete locations because the concrete locations represented by $l_{\mathtt{u}}$ are allocated on the stack of `syscall` while the concrete locations represented by $l_{\mathtt{y}}$ are allocated on the stack of `get`. Thus, $\mathcal{I}^{\mathtt{syscall}}_{\mathit{Loc}}(\rho_{\mathtt{7}})(l_{\mathtt{y}}, l_{\mathtt{u}}) = \mathit{false}$. However, $l_{\mathtt{*u}}$ and $l_{\mathtt{*y}}$ may correspond to the same set of concrete locations because of the implicit pointer copy that occurs from $l_{\mathtt{u}}$ to $l_{\mathtt{y}}$ at the call statement. Thus, $\mathcal{I}^{\mathtt{syscall}}_{\mathit{Loc}}(\rho_{\mathtt{7}})(l_{\mathtt{*y}}, l_{\mathtt{*u}}) = \rho_{\mathtt{7}}(l_{\mathtt{u}}, l_{\mathtt{*u}})$ when $\rho_{\mathtt{11}}(l_{\mathtt{y}}, l_{\mathtt{*y}}) = \mathit{true}$. □

4.3 Judgments

This paper refers to the memory model by using judgments of the form

$$\phi, \rho \vdash_{\mathit{mem}} \mathtt{s} : \phi', \rho', \psi$$

where ϕ (resp. ϕ') is the guard under which execution reaches the entry (resp. exit) point of statement `s`, ρ (resp. ρ') is the points-to graph at the entry (resp. exit) point of `s`, and ψ is the guard under which transfer of control occurs across `s`.

EXAMPLE. In the example in Figure 1, consider the assignment statement `x = *y` on line 13. The guards under which execution reaches the entry and exit points of the statement are both true, and the guard under which control transfer across the statement is also true. The points-to graph $\rho_{\mathtt{13}}$ on entry to the statement has $\rho_{\mathtt{13}}(l_{\mathtt{y}}, l_{\mathtt{*y}}) = \mathit{true}$, $\rho_{\mathtt{13}}(l_{\mathtt{*y}}, l_{\mathtt{**y}}) = \mathit{true}$. The points-to graph $\rho_{\mathtt{14}}$ on exit to the statement has $\rho_{\mathtt{14}}(l_{\mathtt{y}}, l_{\mathtt{*y}}) = \mathit{true}$, $\rho_{\mathtt{14}}(l_{\mathtt{*y}}, l_{\mathtt{**y}}) = \mathit{true}$, $\rho_{\mathtt{14}}(l_{\mathtt{x}}, l_{\mathtt{**y}}) = \mathit{true}$. Consequently, the judgment $\mathit{true}, \rho_{\mathtt{13}} \vdash_{\mathit{mem}}$ `x = *y` $: \mathit{true}, \rho_{\mathtt{14}}, \mathit{true}$ is valid. □

5 Safety

This section describes the safety analysis. Section 5.1 defines *state environments*. Section 5.2 describes the fixed point iteration over the program. Section 5.3 explains how the analysis generates the preliminary summary of a procedure. Sections 5.4 to 5.6 describe the initial state environment used on entry to procedures. Section 5.7 gives the abstraction function that the analysis applies to preliminary summaries to produce final summaries. Sections 5.8 and 5.9 detail the intraprocedural and interprocedural components of the analysis, respectively. Section 5.10 describes a path-sensitive *must-modify* analysis that substantially reduces the number of false alarms.

Our analysis has many components which are detailed in this section. The most important points, however, are:

1. Our analysis is intraprocedurally path-sensitive. We associate with each fact a guard describing the conditions under which that fact holds.

2. Our analysis is compositional. We compute a summary of each procedure P that succinctly describes the behavior of P. The callers of P refer only to P's summary. Summaries are polymorphic in that they are parametrized by whether particular locations are `user` or not.

The combination of (1) and (2) leads to a number of difficulties in scaling a precise user pointer dereference analysis to a program the size of Linux. First, tracking all the possibilities for whether a location can be `user` or not for every location in a procedure turns out to be very expensive. As a result, we use an additional analysis described in Section 5.6 that determines which locations can never be `user`, which turns out to be most locations, greatly improving the scalability of the analysis.

Second, a similar problem arises in deciding how to make use of the path-sensitive information computed for a procedure, which is far too expensive to use directly as the summary of a procedure. We simplify, or *abstract*, this information as described in Section 5.7.

Third, having both polymorphic summaries and path sensitivity introduces subtleties in the mapping between actual and formal parameters at procedure calls as described in Section 5.9.

Finally, it turns out that it is not only important to know what locations a pointer may point to, but also which pointers must be updated by a procedure. For example, it is important to know if a callee is guaranteed to overwrite a user pointer with a kernel pointer. Our solution for this problem is described in Section 5.10.

5.1 State Environments

Let State^P be a set of *abstract states* of a procedure P. A *state environment* $\Gamma^P \in \mathit{StateEnv}^P = \mathit{State}^P \rightarrow \mathit{Guard}^P$ associates each state with the guard under which the program is in that state. There is a natural partial order on state environments: $\Gamma_1 \sqsubseteq \Gamma_2$ if $\forall q(\Gamma_1(q) \Rightarrow \Gamma_2(q))$. The *least upper bound* operator is defined by $(\Gamma_1 \sqcup \Gamma_2)(q) = \Gamma_1(q) \vee \Gamma_2(q)$, the *greatest lower bound* operator by $(\Gamma_1 \sqcap \Gamma_2)(q) = \Gamma_1(q) \wedge \Gamma_2(q)$, *bottom* by $\bot(q) = \mathit{false}$, and *top* by $\top(q) = \mathit{true}$. Two state environments $\Gamma_1, \Gamma_2 \in \mathit{StateEnv}$ are *$\cong$-equivalent* if and only if $\Gamma_1 \sqsubseteq \Gamma_2$ and $\Gamma_2 \sqsubseteq \Gamma_1$.

The *state instantiation* function $\mathcal{I}^P_{\mathit{State}} \in \mathit{PointsTo}^P \rightarrow (\mathit{State}^Q \times \mathit{State}^P) \rightarrow \mathit{Guard}^P$ gives the guard under which a callee state $q^Q \in \mathit{State}^Q$ instantiates to caller state $q^P \in \mathit{State}^P$.

Consider the unchecked user pointer dereference property. The set of abstract states consists of location-typestate pairs $\mathit{State}^P = \mathit{Loc}^P \times \mathit{Typestate}$ where $\mathit{Typestate} = \{\texttt{user}, \texttt{unchecked}, \texttt{unsafe}\}$. The state $(l, \texttt{user}) \in \mathit{State}^P$ signifies that location $l \in \mathit{Loc}^P$ is a user location. Similarly, the state $(l, \texttt{unchecked})$ signifies that l has not been checked, and the state $(l, \texttt{unsafe})$ signifies that l is a user location that has been the target of a dereference while unchecked. Note that a location may be both `user` and `unchecked`—these states are not mutually exclusive. The state instantiation function for this property is defined by

$$\mathcal{I}^P_{\mathit{State}}(\rho)((l_1^Q, t), (l_2^P, t)) = \mathcal{I}^P_{\mathit{Loc}}(\rho)(l_1^Q, l_2^P).$$

5.2 Fixed Point Iteration

The finite-state safety analysis makes several passes over the program. In the ith pass, the analysis analyzes each procedure in isolation generating a *summary state environment* that encodes the behavior of the procedure with respect to the finite-state property and the memory model. If a procedure P calls a procedure Q, then the analysis uses the summary state environment of Q computed in the $(i-1)$st pass to compute the summary state environment of P in the ith pass. Consequently, the summary of P depends on the summary of Q. In a program with recursive procedures, cyclic dependencies arise which requires the analysis to reanalyze procedures until the summaries for all procedures stabilize, or reach a *fixed point*. That is, the analysis terminates after the nth pass when, for each procedure P, the state environment computed in the $(n-1)$st pass is equivalent to the state environment computed in the nth pass.

5.3 Summary Generation

This paper refers to the safety analysis by using judgments of the form

$$\phi, \rho, \Gamma \vdash^i_{\mathit{safety}} \mathsf{s} : \Gamma'$$

where ϕ is the guard under which execution reaches statement s, ρ is the points-to graph on entry to s, and Γ (resp. Γ') is the state environment on entry (resp. exit) to s. The superscript i on the turnstile signifies that the judgment holds in the ith pass of the safety analysis over the program.

The following judgment signifies that the summary state environment of procedure P in the 0th pass is initialized to $\bot$, which means that the analysis initially assumes none of the locations are in any of the predefined states on exit from P.

$$\vdash^0_{safety} \texttt{t}_1 \texttt{ P } (\overrightarrow{\texttt{t}_2 \texttt{ v}}) \texttt{ \{ s \}} : \bot$$

The following judgment describes how the summary state environment, Γ_{sum}, of procedure P is computed in the ith pass, where $i > 0$.

$$\frac{\begin{array}{c}\phi_{init}, \rho_{init}, \Gamma_{init} \vdash^i_{safety} \texttt{s} : \Gamma_{prelim} \\ \Gamma_{sum} = \alpha(\Gamma_{prelim})\end{array}}{\vdash^i_{safety} \texttt{t}_1 \texttt{ P } (\overrightarrow{\texttt{t}_2 \texttt{ v}}) \texttt{ \{ s \}} : \Gamma_{sum}}$$

The top antecedent says that the analysis first performs an intraprocedural analysis of the procedure body, s, with respect to an *initial state environment*, Γ_{init}, to compute a *preliminary summary state environment*, Γ_{prelim}. The bottom antecedent applies an *abstraction function* α to the preliminary summary state environment to compute the final summary state environment, Γ_{sum}, of the procedure. The choice for initial state environment is discussed in Section 5.4, and the abstraction function is described in Section 5.7. The intraprocedural analysis is explained in Section 5.8.

5.4 Initial State Environment

Recall from Section 5.3 that the summary state environment of a procedure is generated with respect to an initial state environment, which represents the contexts in which the procedure is called. Because a procedure may be called in many different contexts, the analysis uses a fresh boolean variable, called a *context variable*, to represent the guard of a particular state in any context, denoting that the guard of the state is unknown and unconstrained during summary generation. The function $xvar^P \in State^P \to XVar^P$ assigns a context variable in $XVar^P$ to each state. The *most general* initial state environment does not incorporate any information about the guards associated with states in contexts, thus allowing the analysis to generate polymorphic summaries that are applicable in any possible context, even those that do not appear in the program. The most general state environment is defined by $\Gamma_{init}(q) = xvar(q)$ for any state q.

EXAMPLE. In the example in Figure 1, consider the procedure get on line 11. Suppose the analysis makes the state-to-context-variable assignment $xvar(l_{\texttt{y}}, \texttt{user}) = v_1$, $xvar(l_{\texttt{y}}, \texttt{unchecked}) = v_2$, $xvar(l_{\texttt{y}}, \texttt{unsafe}) = v_3$, $xvar(l_{*\texttt{y}}, \texttt{user}) = v_4$, $xvar(l_{*\texttt{y}}, \texttt{unchecked}) = v_5$, $xvar(l_{*\texttt{y}}, \texttt{unsafe}) = v_6$, $xvar(l_{**\texttt{y}}, \texttt{user}) = v_7$, $xvar(l_{**\texttt{y}}, \texttt{unchecked}) = v_8$, $xvar(l_{**\texttt{y}}, \texttt{unsafe}) = v_9$. Then, the most general initial state environment would be defined by $\Gamma_{init}(q) = xvar(q)$ for any state q. $\Box$

Section 5.5 describes a special initial state environment used for system call procedures. Section 5.6 describes a refinement of the most general initial state environment crucial to scaling the analysis for unchecked user pointer dereferences to millions of lines of code.

5.5 System Call Initial State Environment

System calls are entry points of an operating system. Consequently, any system call formal parameter is a user pointer, and furthermore, any pointer reachable from a user pointer via a series of dereferences is also a user pointer. Let $Syscall$ be the set of system call procedures. The analysis uses a special initial state environment when analyzing a procedure $P \in Syscall$ that guards each $(l, \texttt{user})$ state with $true$, where l is a location reachable from a formal parameter u of P via a series of dereferences. Formally, the initial state environment Γ^P_{init} where $P \in Syscall$ is defined by

$$\begin{array}{l}\Gamma^P_{init}(l, \texttt{user}) = reachable(\rho^P_{init})(varloc^P(\texttt{u}), l) \\ \Gamma^P_{init}(l, \texttt{unchecked}) = true \\ \Gamma^P_{init}(l, \texttt{unsafe}) = false\end{array}$$

where $reachable \in PointsTo \to (Loc \times Loc) \to Guard$ gives the guard under which a location is reachable via a series of dereferences from another location. It is recursively defined by

$$reachable(\rho)(l_1, l_2) = \bigvee_l [reachable(\rho)(l_1, l) \wedge \rho(l, l_2)]$$

when $l_1 \neq l_2$. Any location l is reachable from itself, so $reachable(\rho)(l, l) = true$.

EXAMPLE. In the example in Figure 1, the system call procedure syscall uses an initial state environment that guards every user state for a location reachable from the user pointer u with $true$, and all other states with *false*: $\Gamma_{init}(l_{\star\mathtt{u}}, \mathtt{user}) = \mathit{true}$, $\Gamma_{init}(l_{\star\star\mathtt{u}}, \mathtt{user}) = \mathit{true}$, and $\Gamma_{init}(l, \mathtt{user}) = \mathit{false}$ for any other l. □

5.6 Refining the Initial State Environment

Recall from Section 5.3 that analyzing a procedure using the most general initial state environment generates a summary that may be used in any calling context including those that do not appear in the program. The cost of such a pure, compositional bottom-up analysis is exactly that it must account for the possibility of every possible calling environment which, depending on the application, may be prohibitively expensive.

For unchecked user pointer dereferences it turns out that only a fraction of pointers are actually user pointers, and restricting the set of pointers to track during the analysis of a procedure to only those that could potentially be `user` in some context substantially improves scalability. In particular, knowing whether the guard associated with a state $(l, \mathtt{user})$ is unsatisfiable in all contexts allows the analysis to avoid tracking l as it is never `user` in any calling context.

The function $\mathit{statecontext} \in (\mathit{Proc} \times \mathbb{N}) \rightarrow 2^{\mathit{State}}$ associates each procedure Q with the set of states that appear in some calling context with a satisfiable guard in pass i.

Formally, $q_1^Q \in \mathit{statecontext}(Q, i)$ if and only if the judgment

$$\phi^P, \rho^P, \Gamma^P_{call} \vdash^i_{safety} \mathtt{v}_1 \ \mathtt{:=}\ \mathtt{Q}\ (\overrightarrow{\mathtt{v}_2}) : \Gamma'^P$$

holds in a procedure P and

$$\bigvee_{q_2^P \in \mathit{State}^P} \Gamma^P_{call}(q_2^P) \wedge \mathcal{I}_{\mathit{State}}(\rho^P)(q_1^Q, q_2^P) \wedge \phi^P$$

is satisfiable. The satisfiability of this condition implies that the Q-state q_1^Q instantiates to some q_2^P state whose associated guard $\Gamma^P_{call}(q_2^P)$ in the calling context is satisfiable. Finally, we define a new *satisfiability initial state environment* ι_{sat} for procedure Q in the ith pass that incorporates the satisfiability of guards in the calling contexts:

$$\iota^i_{sat}(q_1^Q) = \begin{cases} \mathit{xvar}(q_1^Q) & q_1^Q \in \mathit{statecontext}(Q, i-1) \\ \mathit{false} & \mathit{otherwise} \end{cases}$$

EXAMPLE. In the example in Figure 1, let $\Gamma^{\mathtt{syscall}}_{call}$ be the context state environment computed on entry to the call statement to `get` on line 7 in some pass i. Using the system call state environment, the intraprocedural analysis, described in Section 5.8, determines that $\Gamma^{\mathtt{syscall}}_{call}(l_{\mathtt{u}}, \mathtt{user}) = \mathit{false}$, $\Gamma^{\mathtt{syscall}}_{call}(l_{\star\mathtt{u}}, \mathtt{user}) = \mathit{true}$ and $\Gamma^{\mathtt{syscall}}_{call}(l_{\star\star\mathtt{u}}, \mathtt{user}) = \mathit{true}$. Thus, the states $(l_{\star\mathtt{y}}, \mathtt{user})$, $(l_{\star\star\mathtt{y}}, \mathtt{unchecked})$, and $(l_{\star\star\mathtt{y}}, \mathtt{user})$ appear in $\mathit{statecontext}(\mathtt{get}, i)$ because $(l_{\star\mathtt{u}}, \mathtt{user})$ and $(l_{\star\star\mathtt{u}}, \mathtt{user})$ are associated with satisfiable guards in $\Gamma^{\mathtt{syscall}}_{call}$ and $l_{\star\mathtt{y}}$ (resp. $l_{\star\star\mathtt{y}}$) instantiates to $l_{\star\mathtt{u}}$ (resp. $l_{\star\star\mathtt{u}}$). Recall the example in Section 5.4 where the most general initial state environment for `get` was given. For purposes of scalability, we refine the initial state environment for `get` to incorporate information about the satisfiability of the guards in $\Gamma^{\mathtt{syscall}}_{call}$. Thus, $\Gamma^{\mathtt{get}}_{init}$ has $\Gamma^{\mathtt{get}}_{init}(l_{\mathtt{y}}, \mathtt{user}) = \mathit{false}$, $\Gamma^{\mathtt{get}}_{init}(l_{\star\mathtt{y}}, \mathtt{user}) = v_4$, $\Gamma^{\mathtt{get}}_{init}(l_{\star\mathtt{y}}, \mathtt{unchecked}) = v_5$, and $\Gamma^{\mathtt{get}}_{init}(l_{\star\star\mathtt{y}}, \mathtt{user}) = v_7$. Note how the analysis can conclude that $l_{\mathtt{y}}$ is not `user` in any context and thus can avoid tracking this pointer throughout the procedure. □

5.7 Summary Abstraction Function

Recall from Section 5.3 that the intraprocedural analysis of a procedure body generates a preliminary summary state environment, which encodes a very precise description of the behavior of the procedure. However, we have found that retaining this level of precision interprocedurally is prohibitively expensive and, thus, inhibits scalability. Consequently, the analysis performs a sound abstraction on the preliminary summary state environment to compute the final summary state environment using a *summary abstraction function*, $\alpha \in \mathit{StateEnv} \rightarrow \mathit{StateEnv}$. This abstraction step reduces the size of the summary, allowing the analysis to trade precision for scalability without sacrificing soundness. Formally, α is sound if and only if $\Gamma \sqsubseteq \alpha(\Gamma)$ for any $\Gamma \in \mathit{StateEnv}$.

Our particular choice for α in the analysis of the unchecked user pointer dereferences property is the *context variable abstraction*, α_{XVar}, which abstracts a state environment by the strongest state environment whose guards are only over context variables. Formally, $\Gamma' = \alpha_{XVar}(\Gamma)$ is characterized by $\Gamma \sqsubseteq \Gamma'$ and $(\Gamma \sqsubseteq \Gamma'' \sqsubseteq \Gamma') \Rightarrow (\Gamma'' \cong \Gamma')$ when $\mathit{atoms}(\Gamma') \subseteq \mathit{XVar}$ and $\mathit{atoms}(\Gamma'') \subseteq \mathit{XVar}$ for any Γ''. The function atoms gives the set of atomic predicates and boolean variables appearing in the guards used in a given state environment. Intuitively, the context variable abstraction removes any atomic predicates in the state environment guards except for context variables. Let ψ be $\Gamma(q)$. We compute $\alpha_{XVar}(\Gamma)(q)$ in the following manner:

1. Enumerate all disjunctions over the set of context variables that appear in ψ.

2. For each disjunction ϕ, check the validity of $\psi \implies \phi$.

3. Conjoin all the disjunctions that pass the above validity check to form $\alpha_{XVar}(\Gamma)(q)$.

The validity check on boolean constraints in step 2 requires the use a boolean satisfiability solver. This approach requires exponentially many calls to the solver, but because the number of context variables is typically small, it works well in practice. This choice of abstraction function makes our analysis interprocedurally path-*in*sensitive, that is, it cannot reason about branch conditions across procedure boundaries.

EXAMPLE. Let ψ be the guard $(v_a \wedge \psi_1) \vee (v_b \wedge \psi_2)$ where ψ_1 and ψ_2 are predicates not involving context variables. Then, the set of context variables in ψ is $\{v_a, v_b\}$. The disjunctions over $\{v_a, v_b\}$ are $(v_a \vee v_b)$, $(\neg v_a \vee v_b)$, $(v_a \vee \neg v_b)$, and $(\neg v_a \vee \neg v_b)$. The only disjunction that is implied by ψ is $(v_a \vee v_b)$. Thus, the context variable abstraction of ψ is simply $(v_a \vee v_b)$. $\square$

5.8 Intraprocedural Analysis

This section describes how the intraprocedural analysis handles various program constructs.

5.8.1 Statement Sequences

The rule for statement sequences is:

$$\frac{\begin{array}{c}\phi, \rho \vdash_{mem} \mathtt{s}_1 : \phi', \rho', \psi \\ \phi, \rho, \Gamma \vdash^{i}_{safety} \mathtt{s}_1 : \Gamma_1 \\ \phi', \rho', \Gamma_1 \vdash^{i}_{safety} \mathtt{s}_2 : \Gamma_2\end{array}}{\phi, \rho, \Gamma \vdash^{i}_{safety} \mathtt{s}_1 \mathtt{\ ;\ } \mathtt{s}_2 : \Gamma_2}$$

The first antecedent uses the memory model to compute the statement guard ϕ' and points-to graph ρ' after statement $\mathtt{s}_1$. The middle antecedent uses the safety analysis to compute the resulting state environment Γ_1 after executing $\mathtt{s}_1$. Similarly, the last antecedent computes the final state environment Γ_2.

5.8.2 Checking

Let `access_ok` be a procedure that returns a nonzero value if and only if `access_ok`'s user pointer argument points into user space. Consider the statement $\mathtt{v}_1$ `:= access_ok(`$\mathtt{v}_2$`)`. A location l is `unchecked` after the call if l is `unchecked` before the call and the call does not check l. Thus, if Γ is the state environment before the call, then l is `unchecked` after the call if $\Gamma(l, \mathtt{unchecked})$ and either $\mathtt{v}_2$ does not point to l or $\mathtt{v}_1 = 0$ after the call. Now, $\varphi_1 \equiv \neg(rval(\rho)(\mathtt{v}_2, l) \wedge \phi)$ is the guard under which $\mathtt{v}_2$ does not point to l, and $\varphi_2 \equiv \bigvee_{l'}[rval(\rho)(\mathtt{v}_1, l') \wedge (l' = 0)] \wedge \phi$ is the the guard under which $\mathtt{v}_1 = 0$, where ρ is the points-to graph at the call statement and ϕ is the guard under which the call statement executes. The rule for checking statements is:

$$\frac{\begin{array}{c} l \in Loc \\ \varphi_1 \equiv \neg(rval(\rho)(\mathtt{v}_2, l) \wedge \phi) \\ \varphi_2 \equiv \bigvee_{l'} (rval(\rho)(\mathtt{v}_1, l') \wedge (l' = 0)) \wedge \phi \\ \varphi \equiv (\varphi_1 \vee \varphi_2) \\ \Gamma' = \Gamma[(l, \mathtt{unchecked}) \mapsto \Gamma(l, \mathtt{unchecked}) \wedge \varphi]\end{array}}{\phi, \rho, \Gamma \vdash^{i}_{safety} \mathtt{v}_1 \mathtt{\ :=\ access\_ok(}\mathtt{v}_2\mathtt{)} : \Gamma'}$$

5.8.3 Branches

The rule for branch statements is:

$$\frac{\begin{array}{c}\phi_1, \rho_1 \vdash_{mem} \mathtt{s}_1 : \phi_1', \rho_1', \psi_1 \\ \phi_2, \rho_2 \vdash_{mem} \mathtt{s}_2 : \phi_2', \rho_2', \psi_2 \\ \phi_1, \rho_1, \Gamma \vdash^{i}_{safety} \mathtt{s}_1 : \Gamma_1 \\ \phi_2, \rho_2, \Gamma \vdash^{i}_{safety} \mathtt{s}_2 : \Gamma_2 \\ \Gamma' = refine(\Gamma_1, \psi_1) \sqcup refine(\Gamma_2, \psi_2)\end{array}}{\phi, \rho, \Gamma \vdash^{i}_{safety} \mathtt{if\ (v\ !=\ 0)\ \{s_1\}\ else\ \{s_2\}} : \Gamma'}$$

where $refine \in (StateEnv \times Guard) \rightarrow StateEnv$ be a function that refines a state environment Γ with a guard φ by conjoining the guard for each state in Γ with φ as follows:

$$refine(\Gamma, \varphi)(q) = \Gamma(q) \wedge \varphi.$$

The purpose of *refine* is to preserve the information of the two branches Γ_1 and Γ_2 in the combined environment Γ'.

5.8.4 Dereferences

The dereference of a pointer $\mathtt{v}_2$ is `unsafe` when $\mathtt{v}_2$ is an `unchecked`, `user` pointer. Consider the statement $\mathtt{v}_1$ `:= *`$\mathtt{v}_2$ that dereferences $\mathtt{v}_2$. The location l is `unsafe` after the dereference when either l is `unsafe` before the dereference or $\mathtt{v}_2$ points to l and l is `user` and l is `unchecked`. The location l is `unsafe` before the dereference under the guard $\Gamma(l, \mathtt{unsafe})$ where Γ is the state environment before the dereferencing statement. Now, $\varphi_1 \equiv rval(\rho)(\mathtt{v}_2, l) \wedge \phi$ is the guard under which $\mathtt{v}_2$ points-to l, and $\varphi_2 \equiv \Gamma(l, \mathtt{user}) \wedge \Gamma(l, \mathtt{unchecked})$ is the guard under which l is `user` and `unchecked`, where ρ is the points-to graph at the dereferencing statement and ϕ is the guard under which the dereferencing statement executes. The following inference rule describes how the analysis updates the state environment with the

```
1:  void syscall(int c) {
2:        int x;
3:        int y;
4:        int *p;
5:
6:        if (c != 0)
7:              p := &x;
8:        else
9:              p := &y;
10:
11:       process(p, p);
12: }

13: void process(int* q, int *r) {
14:       ...
15: }
```

Figure 2. Example 2

guard under which l is `unsafe`:

$$\frac{\begin{array}{c} l \in \mathit{Loc} \\ \varphi_1 \equiv \mathit{rval}(\rho)(\mathtt{v}_2, l) \wedge \phi \\ \varphi_2 \equiv \Gamma(l, \mathtt{user}) \wedge \Gamma(l, \mathtt{unchecked}) \\ \Gamma' = \Gamma[(l, \mathtt{unsafe}) \mapsto \Gamma(l, \mathtt{unsafe}) \vee (\varphi_1 \wedge \varphi_2)] \end{array}}{\phi, \rho, \Gamma \vdash^{i}_{\mathit{safety}} \mathtt{v}_1 \ \mathtt{:=}\ \mathtt{*v}_2 : \Gamma'}$$

5.9 Interprocedural Analysis

At a call statement, the analysis looks up the summary state environment of the callee generated in the previous pass and instantiates it with respect to the calling context:

$$\frac{\begin{array}{c} \vdash^{i-1}_{\mathit{safety}} \mathtt{t}_1\ \mathtt{Q}\ (\overrightarrow{\mathtt{t}_2\ \mathtt{v}})\ \{\ \mathtt{s}\ \} : \Gamma^{Q}_{\mathit{sum}} \\ \Gamma^{P}_{\mathit{sum}} = \mathcal{I}_{\mathit{StateEnv}}(\rho^P, \Gamma^{P}_{\mathit{call}})(\Gamma^{Q}_{\mathit{sum}}) \\ \Gamma^{P}_{\mathit{out}} = \mathit{refine}(\Gamma^{P}_{\mathit{sum}}, \phi^P) \end{array}}{\phi^P, \rho^P, \Gamma^{P}_{\mathit{call}} \vdash^{i}_{\mathit{safety}} \mathtt{v}_1\ \mathtt{:=}\ \mathtt{Q}\ (\overrightarrow{\mathtt{v}_2}) : \Gamma^{P}_{\mathit{out}}}$$

The first antecedent binds $\Gamma^{Q}_{\mathit{sum}}$ to the summary state environment of procedure Q generated in the $(i-1)$st pass. The second antecedent binds the state environment $\Gamma^{P}_{\mathit{sum}}$ to the *state environment instantiation* of $\Gamma^{Q}_{\mathit{sum}}$, and the last antecedent binds $\Gamma^{P}_{\mathit{out}}$ to the refinement of $\Gamma^{P}_{\mathit{sum}}$ with ϕ^P.

The state environment instantiation function $\mathcal{I}_{\mathit{StateEnv}} \in (\mathit{PointsTo}^P \times \mathit{StateEnv}^P) \rightarrow \mathit{StateEnv}^Q \rightarrow \mathit{StateEnv}^P$ instantiates a callee state environment with respect to the points-to graph and state environment at the call statement in the caller. The remainder of this section presents its definition in several steps. First, note that the instantiation of a callee state environment is complicated by the possibility that a callee location may instantiate to many caller locations, and many callee locations may instantiate to one caller location. The following example demonstrates how this many-many instantiation relation between callee-caller locations may arise.

EXAMPLE. Consider the program in Figure 2. The procedure `syscall` calls the procedure `process`. In `syscall` the stack locations for variables `c`, `x`, `y`, and `p` are represented by the abstract locations $l_{\mathtt{c}}$, $l_{\mathtt{x}}$, $l_{\mathtt{y}}$, and $l_{\mathtt{p}}$, respectively. In `process` the stack locations for variables `q` and `r` are represented by the abstract locations $l_{\mathtt{q}}$ and $l_{\mathtt{r}}$, respectively. The locations that represent the points-to targets of these variables on entry to the procedure are $l_{*\mathtt{q}}$ and $l_{*\mathtt{r}}$, respectively.

Let $\rho_{\mathtt{11}}$ be the points-to graph computed by the memory model on line 11 at the call to `process`. Then, $\rho_{\mathtt{11}}(l_{\mathtt{p}}, l_{\mathtt{x}}) = (l_{\mathtt{c}} \neq 0)$ and $\rho_{\mathtt{11}}(l_{\mathtt{p}}, l_{\mathtt{y}}) = (l_{\mathtt{c}} = 0)$ because of the guarded assignments on lines 7 and 9. Similarly, let $\rho_{\mathtt{13}}$ be the points-to graph computed by the memory model on line 14 on entry to `process`. Then, $\rho_{\mathtt{13}}(l_{\mathtt{q}}, l_{*\mathtt{q}}) = \neg b$, $\rho_{\mathtt{13}}(l_{\mathtt{q}}, l_{*\mathtt{r}}) = b$, and $\rho_{\mathtt{13}}(l_{\mathtt{r}}, l_{*\mathtt{r}}) = \mathit{true}$, where b is an unconstrained boolean variable. Observe that the memory model captures the entry aliasing among the parameters in `process` induced by the call statement on line 11 by introducing an edge from $l_{\mathtt{q}}$ to $l_{*\mathtt{r}}$ under the guard b.

Now, at the call statement on line 11, the location instantiation function introduced in Section 4.2 has

$$\begin{array}{c} \mathcal{I}_{\mathit{Loc}}(\rho_{\mathtt{11}})(l_{*\mathtt{q}}, l_{\mathtt{x}}) = (l_{\mathtt{c}} \neq 0) \\ \mathcal{I}_{\mathit{Loc}}(\rho_{\mathtt{11}})(l_{*\mathtt{q}}, l_{\mathtt{y}}) = (l_{\mathtt{c}} = 0) \\ \mathcal{I}_{\mathit{Loc}}(\rho_{\mathtt{11}})(l_{*\mathtt{r}}, l_{\mathtt{x}}) = (l_{\mathtt{c}} \neq 0) \\ \mathcal{I}_{\mathit{Loc}}(\rho_{\mathtt{11}})(l_{*\mathtt{r}}, l_{\mathtt{y}}) = (l_{\mathtt{c}} = 0) \end{array}$$

Observe that $l_{*\mathtt{q}}$ may instantiate to either $l_{\mathtt{x}}$ or $l_{\mathtt{y}}$, and that $l_{\mathtt{x}}$ may instantiate from either $l_{*\mathtt{q}}$ or $l_{*\mathtt{r}}$. □

Suppose procedure P calls procedure Q at a particular call statement and ψ^Q is a guard that appears in the summary state environment of Q. Let $\Gamma^{P}_{\mathit{call}}$ be the state environment computed at the call statement in P, which maps each P-state to the P-guard under which the program is in that state at the call statement.

The *guard instantiation* function $\mathcal{I}^{P}_{\mathit{Guard}} \in (\mathit{PointsTo}^P \times \mathit{StateEnv}^P) \rightarrow \mathit{Guard}^Q \rightarrow \mathit{Guard}^P$ instantiates a Q-guard to the corresponding P-guard by individually instantiating each of the context variables that appear in the Q-guard. Recall from Section 5.7 that the only atoms appearing in a summary state environment guard are context variables. Also recall from Section 5.4 that each state has a corresponding context variable according to $xvar$ that represents the guard under which the program is in that state on entry to the procedure.

In order to describe guard instantation, we introduce additional notation. The function $xvar^{-1} \in XVar \rightarrow State$ is the inverse function of $xvar$. The function sub takes two sets S_1 and S_2 and constructs all possible substitutions from S_1 to S_2 such that the domain of $\sigma \in sub(S_1, S_2)$ is S_1 and the range is a subset of S_2. The expression $\psi[\sigma]$ is the guard produced by substituting each atom v in ψ with $\sigma(v)$.

Guard instantation entails constructing a substitution σ that maps each Q-context variable to a P-guard and then applying σ to ψ^Q to construct the P-guard $\psi^Q[\sigma]$. Finding the P-guard $\sigma(c^Q)$ that corresponds to a Q-context variable c^Q consists of the following three steps:

1. Look up the Q-state q_1^Q that corresponds to c^Q using $xvar^Q$.
2. Instantiate q_1^Q to a P-state q_2^P.
3. Look up the P-guard that corresponds to q_2^P using Γ^P_{call}.

The first and third steps are straightforward because $xvar$ and Γ^P_{call} are injective functions. However, the second step requires more care because one Q-state may instantiate to many P-states and many Q-states may instantiate to one P-state as demonstrated by the example presented earlier in this section.

We handle the second step by considering all possible substitutions from $State^Q$ to $State^P$ denoted by $sub(State^Q, State^P)$. Consider a substitution $\tau \in sub(State^Q, State^P)$. We define an substitution guard instantiation function $\mathcal{I}_{SubApply}$ that instantiates ψ^Q with respect to τ:

$$\mathcal{I}_{SubApply}(\Gamma^P_{call}, \tau)(\psi^Q) = \psi^Q[\Gamma^P_{call} \circ \tau \circ (xvar^{-1})^Q]$$

The series of function compositions $\Gamma^P_{call} \circ \tau \circ (xvar^{-1})^Q$ carries out the three steps enumerated above but considers only one way to instantiate Q-states to P-states, namely τ. Observe that each substitution $\tau \in sub(State^Q, State^P)$ induces a guard $\mathcal{I}_{SubRefine}(\rho^P)(\tau)$ formed by conjoining, for each $q_1^Q \mapsto q_2^P \in \tau$, the guard under which q_1^Q instantiates to q_2^P:

$$\mathcal{I}_{SubRefine}(\rho^P)(\tau) = \bigwedge_{q_1^Q \mapsto q_2^P \in \tau} \mathcal{I}_{State}(\rho^P)(q_1^Q, q_2^P)$$

Now, we define the guard instantiation function $\mathcal{I}_{Guard}$ that considers all possible substitutions $\tau \in sub(State^Q, State^P)$:

$$\mathcal{I}_{Guard}(\rho^P, \Gamma^P_{call})(\psi^Q) =$$

$$\bigvee_{\tau} \left(\mathcal{I}_{SubApply}(\Gamma^P_{call}, \tau)(\psi^Q) \wedge \mathcal{I}_{SubRefine}(\rho^P)(\tau)\right)$$

Finally, we define of the state environment instantiation function $\mathcal{I}_{StateEnv}$. Let Γ^Q_{sum} be the summary state environment of Q, and let Γ^P_{sum} be its instantiation to P at the call statement. So, $\Gamma^P_{sum} = \mathcal{I}_{StateEnv}(\rho^P, \Gamma^P_{call})(\Gamma^Q_{sum})$. The P-guard associated with a P-state q_2^P in Γ^P_{sum} is formed by disjoining, for each Q-state $q_1^Q \in State^Q$, the guard under which q_1^Q instantiates to q_2^P, and the guard instantiation of $\Gamma^Q_{sum}(q_1^Q)$:

$$\mathcal{I}_{StateEnv}(\rho^P, \Gamma^P_{call})(\Gamma^Q_{sum})(q_2^P) =$$

$$\bigvee_{q_1^Q} \left(\mathcal{I}_{StateEnvQ}(\rho^P, \Gamma^P_{call})(\Gamma^Q_{sum})(q_1^Q, q_2^P)\right)$$

where

$$\mathcal{I}_{StateEnvQ}(\rho^P, \Gamma^P_{call})(\Gamma^Q_{sum})(q_1^Q, q_2^P) =$$

$$\mathcal{I}_{State}(\rho^P)(q_1^Q, q_2^P) \wedge \mathcal{I}_{Guard}(\rho^P, \Gamma^P_{call})(\Gamma^Q_{sum}(q_1^Q))$$

5.10 Path-Sensitive Must Modify Analysis

The underlying alias analysis described in [9] is an interprocedurally path-*in*sensitive may-alias analysis. Unfortunately, this level of precision is not sufficient to reduce the false alarms in our study to a reasonable number. Initially, more than four hundred false alarms were reported because the alias analysis was not precise enough to compute path-sensitive *must-modify* information from callees. A location is *must-modified* if, under some guard, the location must be updated with a value different than the one it contained on entry to the procedure.

Consider the procedures `sys_recvmsg` and `verify_iovec` reproduced in Figure 3. Now, `sys_recvmsg` copies a user pointer into the `msg_sys.msg_name` stack location, and then updates that location with a kernel pointer by invoking `verify_iovec`. Then, the procedure subsequently passes `msg_sys.msg_name` into `sock_recvmsg`. Here, `verify_iovec` conditionally must-modifies `m->msg_name` under the guard

```
long sys_recvmsg(..., struct msghdr __user *msg)
{
   struct msghdr msg_sys;
   int err;
   char addr[MAX_SOCK_ADDR];
   ...

   if (copy_from_user(&msg_sys,msg,
                      sizeof(struct msghdr)))
      return -EFAULT;
   ...
   err = verify_iovec(&msg_sys, ...,
                      addr, VERIFY_WRITE);
   if (err < 0)
      goto out_freeiov;
   ...
   err = sock_recvmsg(..., &msg_sys, ...);
   ...
   out_freeiov:
      return err;
}

int verify_iovec(struct msghdr *m, ...,
                 char *address, int mode)
{
   int err;

   if (m->msg_namelen) {
      if (mode == VERIFY_READ) {
          err = move_addr_to_kernel(m->msg_name,
                                    m->msg_namelen,
                                    address);
          if (err < 0) return err;
       }

       m->msg_name = address;
   } else {
       m->msg_name = NULL;
   }
   ...
}
```

Figure 3. From net/sys.c, net/core/iovec.c

$\psi \equiv \neg(\texttt{mode} = \texttt{VERIFY\_READ} \wedge \texttt{err} < 0)$. Because the original underlying alias analysis was not interprocedurally path-sensitive, it reported that `m->msg_name` was only may-modified under the guard $\psi' \equiv \mathit{false}$. While ψ' is a sound choice for the must-modify guard, as it underapproximates the exact guard ψ, using the more precise guard ψ is important because it prevents the analysis from concluding that `msg_sys.msg_name` is a user pointer on entry to `sock_recvmsg`.

Consequently, we augment the alias analysis with a must-modify analysis that tracks a guard ψ as the condition under which a location is must-modified by the procedure. Then, to keep the guard small in the interest of scalability, we use a special abstraction function (see Section 5.7) called the *correlation abstraction* function, α_{corr}, to compute a conservative underapproximation of ψ described in [5]. Tracking these more precise must-modify conditions in the alias analysis substantially reduces the number of false alarms.

6 Evaluation

This section evaluates an implementation of the unchecked user pointer dereferences analysis. The implementation uses the Saturn program analysis framework [1].

6.1 Setup

We ran our implementation over the entire Linux 2.6.17.1 distribution built for the x86 architecture. The distribution contains over 6.2 million lines of code with 91,543 procedures, 40,760 global variables, 14,794 composite types, and 35,317 initializers. Our implementation transforms the 33,886 loops into tail recursive procedures. The abstract syntax trees are stored in several databases totaling 1.7 GB in size.

We ran the Saturn alias analysis over the abstract syntax trees to compute the memory model for each procedure as described in Section 4, and then we ran our unchecked user pointer dereferences analysis over the abstract syntax trees and memory models. The unchecked user pointer dereferences analysis consists of two phases. The first phase determines which expressions in a procedure may evaluate to a user location, and the second phase determines which of those expressions identified in the first phase are not guarded by a check. Decomposing the analysis into two phases aids scalability by allowing the first phase to identify the minimum set of expressions that need to tracked by the second phase.

Because our analysis is compositional and each procedure is analyzed independently, we parallelized our implementation by distributing the analyses of individual procedures over a cluster consisting of 25 nodes where each node consists of 4 cores and 6 GB of memory. The implementation allotted 3 minutes to each procedure before timing out and moving on to the next procedure. The implementation times out on 154 procedures, or 0.17%. The total running time of the unchecked user pointer dereferences analysis is 3.5 hours.

6.2 Results

A user pointer *source* is a pointer parameter to a system call. A user pointer *sink* is a pointer dereference site. The Linux distribution we analyzed has 627 sources and 867,544 sinks. Our analysis discharges 616 out of the 627 user pointer sources (or 98.2% of sources) and 851,686 of the 852,092 user pointer sinks that do

```
1:  int sound_ioctl(..., uint cmd,
2:                  ulong /*user*/ arg) {
3:     if (_SIOC_DIR(cmd) != _SIOC_NONE &&
4:         _SIOC_DIR(cmd) != 0) {
5:
6:         if(_SIOC_DIR(cmd) & _SIOC_WRITE) {
7:            if (!access_ok(..., arg,...)) {
8:               return -EFAULT;
9:            }
10:        }
11:    }
12:    ...
13:    return sound_mixer_ioctl(..., cmd, arg);
14: }

15: int sound_mixer_ioctl(uint cmd,
16:                       void /*user*/ *arg) {
17:    ...
18:    return aci_mixer_ioctl(...,cmd, arg);
19: }
20:
21: int aci_mixer_ioctl(..., uint cmd,
22:                    void /*user*/ *arg) {
23:    switch(cmd)
24:       case SOUND_MIXER_WRITE_IGAIN:
25:          ...*arg...;
26:       ...
27: }
```

Figure 4. From sound/oss/soundcard.c

not appear in procedures that time out (or 99.95% of sinks). There were 11 warnings on user pointer sources (1 source warning for approximately 560,000 lines of code) and 406 warnings on user pointer sinks (1 sink warning for approximately 15,000 lines of code) all of which can be discharged by 22 additional, simple annotations. Almost all false alarms can be classified into two categories: lack of interprocedural path sensitivity and imprecision in analyzing function pointers. The annotations discharging the false alarms due to interprocedural path-insensitivity refine the guard associated with a state in a summary state environment with additional program predicates, while those discharging the false alarms due to function pointer imprecision refine the set of possible targets computed for function pointer call statements.

6.3 Interprocedural Path Insensitivity

The analysis presented in this paper is fully *intra*procedurally path-sensitive but *inter*procedurally path-*in*sensitive. Within a procedure the analysis reasons about all branch correlations, however, the context variable abstraction performed on the preliminary summary eliminates all path information in the final summary of the procedure, which prevents the analysis from correlating branches and return values across procedure boundaries. Interprocedural path sensitivity is used in a few places in Linux, causing the analysis to fail to discharge 5 user pointer sources and 265 user pointer sinks.

```
1:  struct { char *name; ...} map[] = ...,
2:     {[NFSCTL_GETFD] = {.name = ".getfd", ...},
3:     [NFSCTL_GETFS] = {.name = ".getfs", ...},};
4:
5:  long sys_nfsservctl(int cmd, ..., void *res) {
6:     ...
7:     struct file *file = do_open(map[cmd].name);
8:     ...
9:     int err = file->f_op->read(file, res, ...);
10:    ...
11: }
```

Figure 5. From fs/nfsctl.c

Consider the procedure `sound_ioctl` of `sound/oss/soundcard.c` from Linux 2.6.17.1 (see Figure 4) where the formal parameter `arg` is a user pointer passed from the system call `sys_ioctl`. Line 7 performs a check on the user pointer using the special checking primitive `access_ok` under the condition $\phi \equiv \phi_1 \wedge \phi_2$ where $\phi_1 \equiv$ `_SIOC_DIR(cmd) != _SIOC_NONE && _SIOC_DIR(cmd) != 0` is the conditional on lines 3 and 4, and $\phi_2 \equiv$ `_SIOC_DIR(cmd) & _SIOC_WRITE) != 0` is the conditional on line 6. Thus, before the call to `sound_mixer_ioctl` on line 13, `arg` is checked under the condition ϕ. Consequently, any subsequent dereference of `arg` must be guarded by a condition that implies ϕ. Line 18 in procedure `aci_mixer_ioctl` dereferences the user pointer `arg` under the condition `cmd == SOUND_MIXER_WRITE_IGAIN` which implies ϕ, and thus the user pointer is checked before it is dereferenced and therefore safe. Adding relevant guards to procedure summaries to express interprocedural path sensitivity would enable the analysis to prove this dereference safe.

6.4 Function Pointers

Four user pointer sources and 130 user pointer sinks could not be discharged because the set of targets for some function pointers inferred by the alias analysis is too coarse. Consider the function pointer invocation in procedure `sys_nfsservctl` of `fs/nfsctl.c`, shown in Figure 5. This single site is responsible for the analysis failing to discharge 1 user pointer source and 111 user pointer sinks. On line 1, the global array `map` maps integer constants to file names. On line 7, `sys_nfsservctl` performs a lookup into `map` for a file name and uses the file name to open a file represented by a `struct file` object. The `struct file` object has a field called `f_op` which points to a function pointer table of type `struct file_operations`

```
int notifier_call_chain(struct notifier_block **nl,
           unsigned long val, void *v) {
  int ret = NOTIFY_DONE;
  struct notifier_block *nb;

  nb = *nl;

  while (nb) {
    ret = nb->notifier_call(nb, val, v);
    ...
    nb = nb->next;
  }

  return ret;
}
```

Figure 6. From fs/nfsctl.c

one of whose entries is a field `read`. The memory model imprecisely reports that the targets of the function pointer `file->f_op->read` points to the targets of any `read` field from any instance of `struct file_operations` rather than only the instances that can actually be pointed to by the `file` returned by this call to `do_open`.

6.5 Manual Summaries

We manually summarized several commonly used assembly statements. In particular, we summarized each inline assembly statement to specify that the statement dereferences each of its operands. These dereference summaries allow the analysis to handle more conservatively some inline assembly statements such as `__memcpy` and `__copy_to_user` which dereference some of their operands but do not check whether these operands point into user space.

We also summarized several inline assembly statements and procedures designated by Linux developers as primitives that check whether a user pointer points into user space. These checker inline assembly statements include `__range_ok`, `get_user`, and `put_user`, and the checker procedures include `copy_from_user` and `copy_to_user`.

6.6 Manual Annotations

We used two annotations that soundly restrict which locations are tracked as `user` at particular program points. These two annotations increase the precision of the analysis which prevents the `user` state from propagating to many times more locations than necessary. Without these two annotations, the analysis fails to terminate in a reasonable amount of time because an excessive number of locations are tracked.

We placed one of these scalability annotations in `notifier_call_chain`, a generic procedure shown in Figure 6 whose first parameter `nl` is a linked list of function pointers and whose third parameter `v` is a `void*` pointer. The procedure iterates over `nl` and invokes each of its function pointers on `v`. The pointer `v` is a user pointer in some calling contexts but not others. Because our analysis does not track the correlation between the possible targets of the function pointers in `nl` and whether `v` is a user pointer, it concludes that `notifier_call_chain` passes a user pointer to all possible targets of function pointers in `nl`. We placed the other scalability annotation in procedure `HiSax_command`. This annotation refines the guard under which a particular location is tracked as being `user` with additional interprocedural, path-sensitive information.

7 Conclusion

We have presented a scalable and precise analysis for finite-state safety properties and reported on our experience in attempting to verify the absence of unchecked user pointer dereferences in the Linux operating system. We believe that our analysis can be adapted to verify other important security properties as well.

8 Acknowledgment

The authors would like to thank Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins for their comments on earlier drafts of this paper. This work was supported by NSF grants CCF-0430378 and NSF SA4899-10808PG-1 with additional support from DARPA.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An Overview of the Saturn Project. In *Proceeding of the 7th ACM Workshop on Program Analysis for Software Tools and Engineering*, New York, NY, USA, 2007. ACM Press.

[2] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. pages 85–108, 2002.

[4] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, New York, NY, USA, 2002. ACM Press.

[5] I. Dillig, T. Dillig, and A. Aiken. Static Error Detection Using Semantic Inconsistency Inference. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, volume 42, pages 435–445, New York, NY, USA, 2007. ACM Press.

[6] N. Dor, S. Adams, M. Das, and Z. Yang. Software Validation via Scalable Path-sensitive Value Flow Analysis. In *Proceedings of the ACM SIGSOFT 2004 International Symposium on Software Testing and Analysis*, pages 12–22, New York, NY, USA, 2004. ACM Press.

[7] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 133–144, New York, NY, USA, 2006. ACM Press.

[8] J. Foster, M. Fahndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[9] B. Hackett and A. Aiken. How is Aliasing Used in Systems Software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 69–80, New York, NY, USA, 2006. ACM Press.

[10] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, New York, NY, USA, 2002. ACM Press.

[11] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[12] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs with Type Inference. In *Proceedings of the 13th USENIX Security Symposium*, pages 119–134, 2004.

[13] R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. volume 12, pages 157–171, Piscataway, NJ, USA, 1986. IEEE Press.

[14] L. Torvalds. Sparse.

[15] Y. Xie and A. Aiken. Scalable Error Detection using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, New York, NY, USA, 2005. ACM Press.

[16] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: An Extensible, Expressive System and Language for Statically Checking Security Properties. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 321–334, New York, NY, USA, 2003. ACM Press.